THÈSE DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

M. Andi DREBES

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse:

Dynamic optimization of data-flow task-parallel applications for large-scale NUMA systems

soutenue le 25 juin 2015 devant le jury composé de:

M. Albert COHEN	Examinateur	INRIA / École Normale Supérieure
M. Benoît DUPONT DE DINECHIN	Examinateur	Kalray S.A.
Mme. Nathalie DRACH-TÉMAM	Directeur de thèse	Université Pierre et Marie Curie
Mme. Karine HEYDEMANN	Encadrant de thèse	Université Pierre et Marie Curie
M. Jean-François MÉHAUT	Rapporteur	Université Joseph Fourier / CEA
M. Raymond NAMYST	Examinateur	Université de Bordeaux
M. Nacho NAVARRO	Rapporteur	Universitat Politècnica de Catalunya / Barcelona Supercomputing Center
M. Antoniu POP	Invité	The University of Manchester
M. Pierre SENS	Examinateur	INRIA / Université Pierre et Marie Curie
M. Marc SHAPIRO	Invité	INRIA / Université Pierre et Marie Curie

In loving memory of Hans and Solange.

This page intentionally left blank.

Abstract

Within the last decade, microprocessor development reached a point at which higher clock rates and more complex micro-architectures became less energy-efficient, such that power consumption and energy density were pushed beyond reasonable limits. As a consequence, the industry has shifted to more energy efficient multi-core designs, integrating multiple processing units (cores) on a single chip. The number of cores is expected to grow exponentially and future systems are expected to integrate thousands of processing units. In order to provide sufficient memory bandwidth in these systems, main memory is physically distributed over multiple memory controllers with non-uniform access to memory (NUMA).

Past research has identified programming models based on fine-grained, dependent tasks as a key technique to unleash the parallel processing power of massively parallel general-purpose computing architectures. However, the execution of task-paralel programs on architectures with non-uniform memory access and the dynamic optimizations to mitigate NUMA effects have received only little interest. In this thesis, we explore the main factors on performance and data locality of task-parallel programs and propose a set of transparent, portable and fully automatic on-line mapping mechanisms for tasks to cores and data to memory controllers in order to improve data locality and performance. Placement decisions are based on information about point-to-point data dependences, readily available in the run-time systems of modern task-parallel programming frameworks. The experimental evaluation of these techniques is conducted on our implementation in the run-time of the OpenStream language and a set of high-performance scientific benchmarks. Finally, we designed and implemented Aftermath, a tool for performance analysis and debugging of task-parallel applications and run-times.

Résumé

Au milieu des années deux mille, le développement de microprocesseurs a atteint un point à partir duquel l'augmentation de la fréquence de fonctionnement et la complexification des micro-architectures devenaient moins efficaces en termes de consommation d'énergie, poussant ainsi la densité d'énergie au delà du raisonnable. Par conséquent, l'industrie a opté pour des architectures multi-cœurs intégrant plusieurs unités de calcul sur une même puce. Les sytèmes hautes performances d'aujourd'hui sont composés de centaines de cœurs et les systèmes futurs intègreront des milliers d'unités de calcul. Afin de fournir une bande passante mémoire suffisante dans ces systèmes, la mémoire vive est distribuée physiquement sur plusieurs contrôleurs mémoire avec un accès non-uniforme à la mémoire (NUMA).

Des travaux de recherche récents ont identifié les modèles de programmation à base de tâches dépendantes à granularité fine comme une approche clé pour exploiter la puissance de calcul des architectures généralistes massivement parallèles. Toutefois, peu de recherches ont été conduites sur l'optimisation dynamique des programmes parallèles à base de tâches afin de réduire l'impact négatif sur les performances résultant de la non-uniformité des accès à la mémoire. L'objectif de cette thèse est de déterminer les enjeux et les opportunités concernant l'exploitation efficace de machines many-core NUMA par des applications à base de tâches et de proposer des mécanismes efficaces, portables et entièrement automatiques pour le placement de tâches et de données, améliorant la localité des accès à la mémoire ainsi que les performances. Les décisions de placement sont basées sur l'exploitation des informations sur les dépendances entre tâches disponibles dans les run-times de langages de programmation à base de tâches modernes. Les évaluations expérimentales réalisées reposent sur notre implémentation dans le run-time du langage OpenStream et un ensemble de benchmarks scientifiques hautes performances. Enfin, nous avons développé et implémenté Aftermath, un outil d'analyse et de débogage de performances pour des applications à base de tâches et leurs run-times.

Contents

List of Figures

Listings

List of Algorithms

List of Tables

1 Introduction

Microprocessor development from the early 1970s until the mid-2000s was characterized by substantial increases of sequential performance with each new processor generation due to aggressive scaling of the clock frequency and micro-architectural improvements. Towards the mid-2000s this development reached a point at which higher clock rates and more complex micro-architectures became less energy-efficient, such that power consumption and energy density were pushed beyond reasonable limits. As an alternative, the industry has shifted to more energy efficient multi-core designs, integrating multiple processing units on a single chip. [43]

To satisfy the ever-increasing need for computing power, the focus now lies on increasing the parallel performance by integrating more cores per chip instead of developing more complex cores with higher sequential performance. Today's high performance computing systems range from multi-core systems with several cores to many-core systems composed of dozens or hundreds of general-purpose computing units. The trend to integrate more and more cores is expected to continue and future systems are expected to integrate thousands of cores [26].

Memory accesses in these architectures are a major concern for performance for two main reasons. First, the clock frequency of processors and DRAM have evolved at different speeds, resulting in a drastic performance gap between these components known as the memory wall [64]. From the perspective of a core, each access to main memory potentially stalls the core for many cycles until data eventually becomes available and can thus reduce performance considerably. Second, the integration of a large number of cores in a parallel system puts additional stress on the memory interface due to an increased amount of requests that can be issued per time unit. This might lead to contention on the memory controller, further increasing the latency of memory accesses and further decreasing performance.

To improve memory bandwidth and to avoid contention, many-core systems integrate multiple memory controllers and group cores with memory controllers into nodes that are connected through large-scale links. The local memory of a node can be accessed by the cores of the node without using the interconnect and local accesses are thus fast, while accesses to remote nodes are slow. Access to local and remote memory in such systems with non-uniform access to main memory (NUMA) is usually managed transparently through the hardware and main memory is accessible by software through a single, unified address space. However, to keep the latency of memory access low and to avoid contention on specific nodes, data and computations must be distributed across nodes using appropriate software techniques such that all accesses ideally target local memory and such that none of the nodes is targeted by a significantly higher amount of requests than the others.

While existing sequential applications required no or only few changes to benefit from improvements of sequential performance of systems with uniform memory access (UMA), the shift to parallel architectures and non-uniform memory access for mainstream computing represents a major challenge for software development and optimization with fundamental changes throughout the entire software stack [13, 27]. This involves:

- the parallelization of algorithms to take advantage of the processing power of multiple cores.
- the design of parallel programming models that allow to denote parallelism and whose execution models define how a parallel program is executed.
- the design of compilers that translate the specification of a parallel program to code that is executable on a parallel target architecture.
- the development of low-overhead execution environments that implement the execution models of parallel programming models.
- the development of efficient system software providing fine-grained control over the assignment of computations to cores and of data to nodes.

Due to the wide variety of available parallel hardware architectures and short release cycles of systems with higher core counts, parallel applications are expected to be portable across multiple systems and to be able to yield shorter execution times with each additional core. As parallel software is usually significantly more complex than sequential software, parallel programming models must provide means to improve the productivity and to reduce the implementation overhead related to parallelism.

Task-parallel programming models [22, 25, 70, 72, 36, 54, 33, 35, 28] are a recent trend to respond to these challenges. A key feature of these models is to abstract from details of the underlying architecture and system software and to reduce the specification of a parallel program to the definition of fine-grained tasks and dependences between them. While this concept greatly improves the productivity of the programmer, it leaves issues related to efficient interaction with system software, efficient exploitation of the hardware and performance portability to the implementation of the execution model. On many-core NUMA systems, this includes the optimization of memory accesses, i.e., keeping accesses to main memory local and distributing requests equally to all nodes. Providing efficient mechanisms for the placement of tasks on cores and the placement of data on nodes is indispensable for the implementation of task-parallel programming models of many-core systems in order to achieve high performance.

1.1 Objectives and contributions of this thesis

Past research has lead to a wide variety of approaches for optimized placement of computations and data on NUMA systems, ranging from static optimizations by the compiler to dynamic solutions operating at execution time. However, only little work has been done for task-parallel applications and run-times managing their execution.

The purpose of this thesis is to explore the challenges and opportunities regarding the efficient exploitation of many-core NUMA systems by task-parallel applications with a focus on accesses to main memory and to propose mechanisms for efficient task and data placement. The first major challenge in the development of such mechanisms is to identify and analyze the interactions between the application, the run-time, the hardware and the operating system that are relevant for data locality and performance of task-parallel programs. The second major challenge is to design mechanisms that are portable, fully-automatic, application-transparent and able to react to dynamic changes of the application. From this perspective, the thesis focuses on the identification and detailed analysis of:

1. characteristics of the run-time systems that are needed to support low-overhead implementations of mechanisms for task and data placement and to prevent the run-time system itself from becoming a bottleneck for performance.
2. characteristics of task-parallel programs that are relevant for data locality and performance and that have to be taken into account for the design of mechanisms for data and task

placement.

Based on the findings of this analysis we developed multiple automatic on-line techniques for efficient and portable data and task placement exploiting information on point-to-point data dependences readily available in modern task-parallel run-time systems at execution time.

The implementation of these mechanisms and their experimental evaluation led to several practical contributions. First, we developed a NUMA-aware run-time based on OpenStream, a state-of-the-art framework for task-parallel applications, that serves as the basis for our mechanisms for task and data placement. Second, we implemented these mechanisms and integrated them into the NUMA-aware run-time. To validate that our concepts apply to real-world task-parallel applications, we have implemented of a set of high performance, scientific OpenStream benchmarks and executed them using our run-time. Finally, we designed and implemented Aftermath, a tool for performance analysis and debugging of task-parallel applications and run-times. This allowed us to understand the interactions between the task-parallel application, the run-time system, the hardware and the operating system and to take these into account for the theoretical concepts for task and data placement.

1.2 Outline of this document

The outline of this thesis is the following. Chapter 2 presents the context of this thesis. The chapter provides a motivation for task-parallel languages as a programming model for many-core systems and defines the goals for efficient exploitation of the hardware through optimized mappings of computation to cores and data to memory controllers by a run-time system. A presentation of related work forms the basis of the problem statement provided at the end of the chapter.

Chapter 3 presents OpenStream, a data-flow extension for OpenMP that enables task parallel programming and that we have chosen for the implementation of the concepts proposed in this thesis. We present the syntax of OpenStream, its execution model and provide an outline for the compilation of OpenStream applications.

Chapter 4 focuses on the requirements for run-time systems to support efficient task and data placement on many-core NUMA systems. The chapter shows how memory is typically placed on nodes by the operating system and investigates the influence of this strategy on the locality of memory accesses in OpenStream programs. Based on these findings, we propose methods for low-overhead, NUMA-aware memory allocation and the determination of memory placement.

Chapter 5 introduces dynamic single assignment, a programming style that allows the run-time system to reliably and accurately determine the data that is accessed by a task before the task is executed. We point out the implications of this programming style on the memory footprint related to task creation and conclude that parallel task creation is beneficial for the memory footprint as well as for performance.

The experimental setup for the validation of the concepts presented in this thesis is given in Chapter 6. This includes a detailed description and characterization of all benchmarks, a description of baselines to which we compare our optimizations, the definition of the methodology for measurements and a description of the architectures of our test systems.

Chapter 7 presents our solutions for NUMA-aware task placement. We propose work-pushing, a technique that transfers tasks to cores associated to nodes that contain the memory regions accessed by the tasks and topology-aware work-stealing, a mechanism that steals tasks from an incrementally widening neighborhood of a core with respect to the memory hierarchy.

Efficient data placement is addressed in Chapter 8 presenting deferred allocation. In this technique, the allocation of the memory regions that receive the input data of a task is delayed from task creation to the moment when the set of tasks writing to the memory regions as well as the nodes on which these tasks execute are known. As in Chapter 7, we analyze the impact of this strategy on data locality and performance.

Chapter 9 treats our optimizations for broadcasts, passing the data of a single producer is to multiple consumers. We present broadcast tables and show that this optimization considerably

reduces the memory footprint and significantly increases performance of a broadcast-intensive linear algebra kernel.

Chapter 10 covers Aftermath, a tool for the visualization and analysis of execution traces that we have originally developed for performance debugging of our optimizations and the benchmarks presented in Chapter 6, but whose concepts can be applied to performance analysis of task-parallel applications and run-times in general.

The conclusions on the work presented in this thesis and directions for future research are given in Chapter 11.

2 Context and problem statement

In this chapter, we introduce the scientific and technical context of this thesis. In Section 2.1, we first explain why the advent of massively parallel general-purpose architectures in mainstream computing has raised the need for alternative programming models. We present the expectations on these models and introduce task-parallel programming as an approach that addresses these issues. The aspects of task-parallel programs that are critical for performance are covered subsequently. Section 2.2 provides an overview of the architecture of high performance many-core systems and emphasizes the prerequisites for their efficient exploitation. The focus of this presentation lies on the efficient use of the memory architecture. Section 2.3 discusses how this problem can be tackled through proper orchestration at execution time. After a presentation of solutions proposed in related work in Section 2.4, we define the objectives of this thesis and how our solutions differ from existing approaches in Section 2.5.

2.1 Parallel programming models for many-core architectures

As parallel architectures have become omnipresent from embedded systems through desktop computers to systems dedicated to high performance computing, development of parallel software has become an imperative to exploit the processing power of contemporary systems. The wide variety of available parallel architectures and short periods between releases of new systems with increased core counts have lead to a shift of the expectations on programming models for parallel systems. Modern programming models are expected to enable development of applications that are able exploit the parallel processing power of a machine efficiently, that yield similar performance across machines with similar characteristics and that can take full advantage of an increasing number of processing units. To compensate the additional complexity of the development of parallel software compared to sequential implementations, more productive approaches that abstract from technical aspects of the implementation are needed, such that programmers can concentrate on the specification of parallelism. These key requirements can be summarized as *scalability* [1], *performance portability* and *productivity*.

1. There are many definitions for scalability and often the term is only defined intuitively [52]. The definition used in this thesis is that an application is scalable if its speedup over sequential execution is approximately linear wrt. the number of processing units used for execution on the same machine.

2.1.1 Task-based programming models

Task-parallel programming is an increasingly popular approach to address the issues above. Many different approaches for task-parallel programming have been proposed, ranging from generic concepts for task-parallel computations (e.g., CONCURRENT COLLECTIONS [33]), through general-purpose libraries (e.g., THREADING BUILDING BLOCKS [54]), language extensions (e.g., CILK [22, 49], OpenMP [23, 25], STARSS [70], OPENSTREAM [72, 74], X10 [36] and HABANERO [35], LIBKOMP [28]) to specialized libraries for specific domains (PLASMA [60] based on QUARK [85]). The key aspect of task-parallel programming models is to expose large amounts of parallelism by creating small units of work, called tasks, and to specify interactions between tasks that constrain which tasks can run in parallel. How tasks are declared and which methods of synchronization are available varies between the approaches for task-parallel programming. The complete set of tasks and the synchronization between tasks representing the parallel computation do not necessarily have to be constituted statically. New tasks can be created and synchronization be defined dynamically and incrementally at execution time. These requests for task creation and synchronization are handled by a *run-time system*, or *run-time* for short, whose purpose is to manage the execution of the task-parallel program.

Productivity in task-based programming is addressed by omitting technical details in the specification of a program and by focusing on the definition of tasks and their interactions. Code of task-parallel programs specifies *what* can execute in parallel, but leaves the choice of *where* and *when* to execute tasks to the run-time. This abstraction lifts the obligation to provide code for a particular kind of machine or a particular operating system and allows the programmer to concentrate on issues that are inherent to the algorithm that is being implemented.

Scalability is addressed by encouraging the specification of very fine-grained tasks with fine-grained inter-task synchronization, which increases parallelism and enables exploitation of a large number of processing units simultaneously. However, fine-grained parallelism is only a necessary condition for scalability. To unleash the parallel processing power of a machine, it is also necessary that hardware resources are exploited efficiently and that the interface of the operating system is used appropriately. This is the responsibility of the run-time system, which maps parallelism to the machine and which acts as a mediator between the application and the operating system.

Similar to scalability, performance portability is addressed both in the programming model and the specific implementation of the run-time. By leaving out platform-specific code in the specification of a program, the same code can be used to obtain specialized versions for execution on different platforms. The run-time is responsible to adapt the execution of the application to the specificities of the target platform, which does not only involve preservation of correct execution, but also efficient exploitation. This can be achieved through appropriate parameterization of the run-time or by providing platform-specific implementations with a well-defined, platform-independent interface between the application and the run-time.

2.1.2 The run-time system

The run-time system is the central component of task-parallel programming and is responsible for correct and efficient execution of the task-parallel application. Figure 2.1 shows the embedding of the run-time system into the execution environment. The services of the run-time system, e.g., task creation and synchronization, are directly invoked by the application. In many cases, the run-time is provided as a run-time library that the application is linked against dynamically and requests consist in ordinary calls to library functions. The infrastructure of the run-time system satisfying the requests is in turn based on the services provided by the operating system. This interaction is rarely direct and commonly based on system libraries with more convenient interfaces for system calls. The operating system forms the bottom of the software stack and finally provides access to the hardware.

The functionality provided by the run-time system can be grouped into multiple components. The exact set of components and the separation of components depend on the specific programming

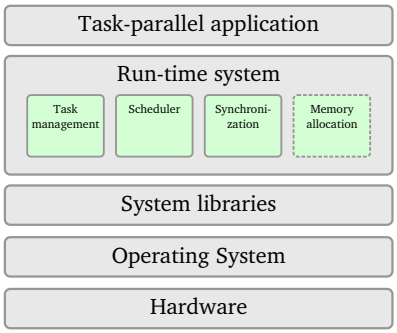


Figure 2.1: *Embedding of the run-time system into the execution environment*

model and its implementation. Generally, the run-time manages the creation and destruction of tasks, implements task synchronization, detects when a task becomes ready for execution and contains a *scheduler* that distributes ready task to the different cores of the machine. In cases where the run-time also manages memory of the application, a memory allocator is part of the run-time as well. The performance of a task-parallel application highly depends on the implementation of the run-time components:

- First, algorithms and data structures of the run-time itself should not become a bottleneck for performance. For instance, the computational complexity of algorithms for task management and dependence tracking should be sufficiently low to handle large amounts of tasks, the memory footprint of internal data structures should be small and data exchanges between concurrent activities of the run-time should be efficient. Decentralized algorithms should be preferred to prevent centralized components from becoming a bottleneck.
- Second, the run-time must interact efficiently with its environment. For example, slow system calls should be avoided or at least not be invoked frequently and the run-time should use appropriate methods for synchronization provided by the system libraries.
- Third, the execution of tasks should be orchestrated, such that hardware resources are used efficiently, resulting in the lowest possible time for execution of the application. This aspect is particularly platform-specific and requires detailed knowledge of the target architecture.

The next section presents the hardware architecture of contemporary high performance systems targeted in this thesis. Its purpose is to emphasize which aspects are relevant to performance and to point out the low-level characteristics of efficient executions of applications. Section 2.3 then discusses how this behavior can be achieved by the run-time.

2.2 High performance parallel hardware architectures

Modern high performance hardware architectures are multi-core and many-core systems, which integrate multiple processing units on the same chip and combine multiple chips to provide large amounts of parallel processing power. As energy efficiency has become the driving factor in the development of multi-core and many-core systems, the architecture of individual cores tends to be less complex than for high performance single-core architectures [43]. However, sequential performance still plays an important role [53] and, as a consequence, recent general-purpose parallel architectures inherit many optimizations from single-core architectures. In the context of this thesis, we focus on systems designed for high performance computing with less strict constraints on energy consumption and thus less drastic trade-offs between sequential performance of each individual core and the number of cores. Besides basic micro-architectural optimizations, e.g., pipelining, the use of caches and basic SIMD instructions, these systems also employ more aggressive techniques, such as out-of-order and superscalar execution, branch prediction, speculative execution and hardware prefetching.

Particular attention, both in single-core systems as well as parallel architectures, is paid to

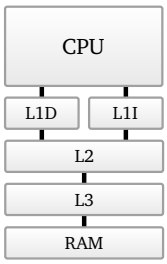


Figure 2.2: *Example of a hierarchy of caches with three levels L1 to L3 with separate and unified caches*

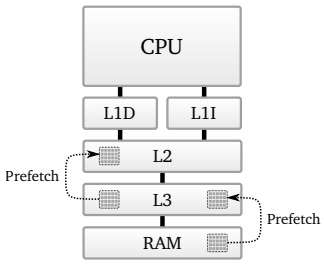


Figure 2.3: *Hardware prefetching between DRAM and the last level cache and between caches*

architectural improvements that reduce the impact of memory accesses on performance. As this thesis focuses on the analysis and mitigation of bottlenecks related to memory accesses, the following presentation of the hardware architecture of many-core systems spotlights the memory subsystem.

2.2.1 The cache hierarchy

During the past decades, technology for processors and DRAM have evolved at different speeds, leading to a dramatic gap between the computational performance and the main memory access time referred to as the *memory wall* [64]. While computations involving only the register file of the processor can be carried out fast, accesses to main memory limit performance as the processor stalls for many cycles waiting for data from DRAM before execution can be resumed. Hence, the reduction of the impact of memory accesses on performance is a major concern in computer architecture as well as in the software industry.

Cache hierarchies of single-core systems

To mitigate the impact of high-latency memory accesses, processors have been provided with small and fast on-chip *cache memory*, which enables exploitation of temporal and spatial locality of memory accesses. Temporal locality refers to the reuse of previously requested data and can be exploited by keeping data in the cache that has already been fetched from main memory. Spatial locality designates the use of data at addresses close to previously requested data and can be exploited by bringing data from neighboring addresses to the cache.

Requests to data that is already present in a cache result in *cache hits* and those to data that must be fetched from main memory are referred to as *cache misses*. On the one hand it is desirable that cache capacity is as high as possible in order to maximize the amount of data that can be held simultaneously in the cache. On the other hand, the latency of accesses to a cache increases with its size, such that smaller caches are faster than bigger caches. Dimensioning the cache thus involves a trade-off between the size and the average latency of accesses to the cache. Modern systems therefore rely on a *hierarchy of caches* with multiple *levels*, where caches at the upper levels, placed near a processing unit (e.g., first-level caches), are small and fast and caches at the lower levels farther away (e.g., third-level caches) are bigger, but also slower. Hence, the cost of a cache miss in a cache near the CPU is still higher than the cost associated to a hit, but if the request results in a hit in one of the caches at lower levels the cost is lower than an access to main memory. Typically, high performance systems employ three levels of caches with first-level caches of tens of KiB, accessible within only a few processor clock cycles and last level caches of a few MiB that can be accessed in tens of cycles.

As both data and instructions are stored in main memory, the latency of DRAM impacts both data accesses and the transfer of instructions to the CPU. Hence, the use of caches can not only improve latency of accesses to data, but can also speed up instruction fetching. *Unified caches* store instructions and data jointly and serve requests for the two types equally without

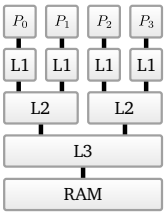


Figure 2.4: *Shared and private caches in a multi-core system*

differentiation. In contrast to this, in designs with *separate caches*, data and instructions are stored in distinct caches, namely the *data cache* and the *instruction cache*. These separate caches can operate in parallel and are smaller and thus faster than a unified cache. Hence, requests for data and instructions can be satisfied simultaneously, which increases performance for pipelined execution of instructions. However, the use of separate caches represents a static partitioning and can result in under-utilization of the cache capacity. Unused capacity of the instruction cache could be needed by the data cache and vice versa, but remains inaccessible due to the separation of instructions and data. Hence, most commonly, only the level closest to the CPU uses separate caches while the remaining levels are composed of unified caches. Figure 2.2 shows an example of a hierarchy of caches with three levels L1 to L3. The first level cache is separated into an instruction cache (L1I) and a data cache (L1D). The second and the third level cache are both unified caches.

Requests for data elements that have neither been referenced before nor stored at neighboring addresses of previously requested data result in cache misses, independently from the size of caches and the depth of the hierarchy. A hardware technique that aims at reducing the number of these *compulsory cache misses* is *hardware prefetching*. In this optimization, the history of previously accessed addresses is analyzed in order to predict which addresses will be accessed in the future. The data at these addresses is brought from main memory to a cache speculatively in the hope that the prediction is correct and that the data will actually be referenced. As prefetching can be done in parallel with the execution of instructions, the delay of the instruction that first accesses this data can be reduced or hidden entirely if prefetching finishes in time. Prefetching is also employed between caches to reduce the number of compulsory misses at upper levels of the cache hierarchy. Figure 2.3 illustrates hardware prefetching between DRAM and the last level cache as well as between the last level cache and the second level cache of the memory hierarchy from Figure 2.2.

Cache hierarchies for architectures with multiple cores

Cache hierarchies of systems with multiple cores are slightly more complex than those of single-core architectures. An important decision that must be taken for the design of such a hierarchy is whether a cache is *shared* among multiple cores or whether it is *private*. The advantage of a private cache is that its capacity is dedicated to the associated core and cannot be *polluted* with data from another core. Moreover, the absence of concurrent accesses reduces the complexity of the interface and reduces contention, increasing cache performance. Shared caches, however, enable low-overhead communication between cores, as data can directly be exchanged within the cache. Furthermore, data that is accessed by multiple cores must only be stored once, which reduces the total amount of required cache memory. As the impact on performance of cache sharing is application-specific [87] and as general-purpose architectures must yield acceptable performance for a wide variety of applications, they cannot opt for one extreme and thus employ both private and shared caches. As a rule of thumb, private caches are employed at the upper levels of the cache hierarchy near the core (e.g., first and second level caches) and shared caches are used for the lower levels of the hierarchy (e.g., the third level cache).

An important issue regarding the cache hierarchy of parallel systems is related to *cache coherency*. With private caches, or, more generally speaking, with caches which are not shared by all cores, data can be present in multiple caches at once. In order to provide a consistent view on memory for all cores, modification of shared data must result in invalidation or update of its copies. Systems that provide this coherence transparently are referred to as *cache coherent architectures*.

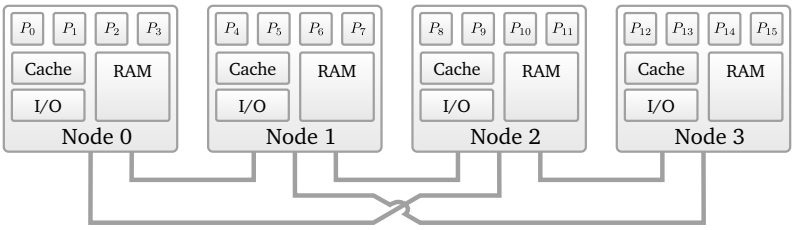


Figure 2.5: *Example of a NUMA system with 16 cores and 4 nodes*

Figure 2.4 shows an example of a hierarchy of caches for a multi-core system composed of four cores. The first level caches are private, second level caches are shared by pairs of cores and the third level cache is shared by all cores.

2.2.2 Non-uniform memory access

Although caches and prefetching greatly improve performance if exploited efficiently, not all accesses to main memory can be eliminated and improvement of DRAM access latency remains a major concern for performance. Parallel architectures exacerbate this difficulty, as each additional core potentially increases the total number of requests to main memory per time unit and thus increases pressure on the memory controller. Once the bandwidth of the controller is saturated, latency of accesses to DRAM increases and memory accesses rapidly become a bottleneck for performance. Therefore, high performance general-purpose parallel architectures contain multiple memory controllers which are physically distributed over the machine. This allows the hardware to satisfy memory accesses in parallel and overall bandwidth is increased.

Cores, caches and memory controllers in these systems are grouped into *nodes* connected through large-scale links. The interconnection formed by the links can contain direct connections as well as indirect connections between nodes. For indirect connections, data cannot be exchanged directly and must traverse one or more intermediate nodes on the way from the source to the destination. The distance between a core and the targeted memory for such a transfer is expressed in *hops*, representing the number of links on the shortest path between the core and the controller.

Accesses from cores to memory of the same node are referred to as *local memory accesses* and accesses to the memory of different nodes are called *remote memory accesses*. Local memory accesses can be handled without engaging the interconnect and can thus be carried out rapidly. Accesses to remote nodes require the use of the interconnect and are thus slower, with an increasing latency for each additional hop. As the latency of a memory access depends on the location of the requesting core and the distance to the targeted memory controller, these systems are referred to as systems with *non-uniform memory access* (*NUMA*). Non-uniform memory access in addition with cache coherence is abbreviated as *ccNUMA* (*cache-coherent NUMA systems*). As we are targeting only architectures with cache coherency, we use the terms NUMA and ccNUMA interchangeably in the rest of this thesis.

Figure 2.5 shows a sample architecture with 16 cores and four memory controllers grouped into four nodes. For cores P_0 to P_3 the memory of Node 0 can be reached at a distance of 0 hops and thus represents the local memory. The direct neighbors, Node 1 and 2, are at a distance of one hop. Node 3 can only be reached by passing through Node 1 or Node 2 first, its distance relative to Node 0 is thus two hops.

Despite the physical distribution of memory controllers across multiple nodes with *non-uniform access*, NUMA systems provide a *uniform addressing scheme* that provides access to the entire memory of the system using a single address space. The translation from addresses to nodes and routing within the interconnect is managed by the hardware and the physical distribution remains essentially hidden to programs executing on the machine. However, modern operating systems explicitly support NUMA and provide interfaces that allow application to allocate memory on specific nodes or to obtain information on data placement.

The efficient use of high performance parallel hardware architectures requires that cores, caches and non-uniform memory access are taken into account. The next section defines the goals for efficient exploitation from a software perspective.

2.2.3 Efficient exploitation of many-core architectures and NUMA

Efficient exploitation of the hardware consists in minimizing execution time through appropriate low-level behavior at the micro-architectural level. Due to the complexity of the architecture of many-core NUMA systems, this is a difficult task involving many aspects of the execution of a program. The main directions of efficient exploitation are the following:

- **Maximizing parallelism**
  Leveraging the parallel processing power of the machine requires that computations are distributed to as many cores as possible. Ideally, all of the cores can be used simultaneously throughout the entire execution time with minimal overhead for communication resulting from the distribution.

- **Maximizing sequential performance on each core**
  Sequential execution on each individual core should be as fast as possible. This involves maximizing instruction level parallelism for superscalar architectures, the use of SIMD instructions to perform computations simultaneously on multiple ALUs and optimizations for the instruction pipeline. Memory accesses should be avoided through the efficient use of registers.

- **Efficient use of caches**
  To keep the impact of memory accesses on performance as low as possible, the cache hit rate should be as high as possible. Spatial and temporal locality should be maximized through appropriate layout of data elements in memory and appropriate order of memory accesses. Data that is accessed frequently should fit into the hierarchy of caches in order to avoid conflicts and thus eviction of critical data due to limited cache capacity. The pattern of memory accesses should be sufficiently regular to be captured by the prefetchers, such that the number compulsory misses can be kept as low as possible.

- **Minimizing the latency of accesses to main memory**
  The latency of memory accesses depends on the contention of memory controllers as well as the distance between the requesting cores and the controllers satisfying the requests. To minimize contention, it is important to distribute memory accesses over all of the nodes of the machine. To minimize the average distance of memory accesses, the ratio of local memory accesses to the total number of accesses should be as high as possible.

Depending on the characteristics of the application, the importance of each of the directions for efficient exploitation above varies. For example, the performance of an application with frequent accesses to memory might be more sensitive to an improvement of the cache hit rate than to an increase of the number of exploited cores, while the performance of a compute-bound application is mostly determined by the number of cores and maximization of sequential performance on each core and may remain insensitive to improvement of the cache hit rate. Each of the directions represents a major challenge on its own with large configuration spaces and complex relationships.

In addition, overall performance generally relies on multiple optimizations belonging to different directions. As improvements in one direction can constrain the configuration space in other directions, it is usually impossible to consider each direction separately and to simply combine the optimizations. Furthermore, optimizations can be applied at different software layers and different stages. For example, partitioning of data for caches can be done manually by the programmer, by the compiler or dynamically at execution time. Each of these solutions has its limitations, advantages and drawbacks. For optimal performance, different aspects must therefore

be considered jointly throughout the whole process from the implementation to execution of an application.

However, each direction for the improvement of application performance represents an entire field of research on its own and exhaustive exploration of all possible combinations is infeasible. In this thesis, we concentrate on the aspects of non-uniform memory access in the context of the execution of task-parallel programs. Other directions for improvement are not directly addressed by our approaches, but were taken into account during development and parametrization of the applications for experimental evaluation.

One possibility to avoid node contention and to improve the locality of memory accesses of task-parallel programs is to address NUMA-related issues through efficient mapping of parallelism to the machine, i.e., the efficient mapping of computations to cores and of data to memory controllers. In the next section, we discuss the principles of this approach and motivate which software components are commonly involved in this process.

2.3 Efficient mapping of parallelism to the hardware

The mapping of parallelism to the hardware consists of two parts: the mapping of computations to cores and the mapping of data to memory controllers. The definition for an *efficient* mapping in this thesis is a mapping that keeps the wall clock execution time of an application as low as possible. To this end, both the mapping to cores and to memory controllers must exploit the hardware efficiently by inducing the low-level behavior described in the previous section. For the mapping of computations to cores this means that ideally, all cores of the system are used simultaneously and that each core is used efficiently with maximal sequential performance. The mapping of data to nodes should minimize the latency of memory accesses by avoiding contention on memory controllers and by keeping the distance between cores and the targeted nodes low. Ideally, all memory accesses are local and requests are distributed equally over all memory controllers.

In some cases, these goals can be achieved simultaneously. For example, computations that do not share any data can execute concurrently on cores of different nodes and the data that is accessed by each computation can be mapped to the same node as the computation. However, in many cases, the goals are difficult or even impossible to achieve at the same time. As an example, consider a data buffer that is accessed by multiple concurrent computations. One possible mapping of the data is to place the buffer on a single node. If all the computations accessing the buffer are mapped to cores of the same node, all memory accesses target local memory, but the computing resources of the machine are under-utilized as the cores of the remaining nodes are not used at all. If the computations are spread on cores of multiple nodes, the parallel processing power of the machine is well exploited, but a significant part of the accesses to memory target a remote node. In addition, contention of the node that contains the buffer is high as it has to deal with concurrent accesses from multiple cores. The last possibility for a mapping is to distribute the buffer over multiple nodes and to spread computations over the entire machine. This mitigates the contention problem, but most of the memory accesses still target remote nodes. An efficient mapping is thus often a trade-off between optimizations for data locality, contention and parallel execution.

Whether a trade-off is necessary and which trade-off is required heavily depends on the behavior of the application. To determine *where* to carry out computations and *where* to place data, it is crucial to determine *which computations* have to be mapped, *which data* has to be placed and, most importantly, *how* data is accessed. For this purpose, program behavior must be detected and predicted adequately. More detailed information on application behavior and more precise prediction allow for more efficient mapping strategies and better performance.

Two common approaches to implement dynamic mappings of computations to cores and dynamic mappings of data to nodes consist in providing appropriate algorithms for scheduling memory allocation. For efficient exploitation of the hardware, the scheduler must be modified to take into account the topology of the hardware, the behavior of the application at execution time, which data is accessed and the data placement at the moment when a scheduling decision is taken or a combination of these aspects. Efficient memory allocation must take into account

the topology of the hardware, the execution locations of threads or tasks, or both characteristics to place data accordingly. The placement of data can be carried out either synchronously upon allocation of new memory or dynamically through data migration during execution. By combining scheduling and memory allocation, the mapping of computations and data can be addressed at the same time, avoiding that each approach only reacts passively to the other. For example, the scheduler can advise the memory allocator to place data on certain nodes on which computations will be scheduled in the future. Similarly, the memory allocator can provide hints about future data placement and advise the scheduler for future mappings of computation. The range of possible mappings is thereby extended and bottlenecks that arise from insufficient communication between the scheduler and the memory allocator can be avoided. As the determination of an optimal solution for a mapping is generally considered to be computationally intensive and thus ill-suited for on-line techniques, the algorithms used for scheduling and allocation are usually based on *heuristics*.

2.4 Related work

Past research has led to a multitude of approaches for efficient exploitation of parallel architectures based on scheduling and data allocation. The main characteristics of these approaches are:

- The set of heuristics (scheduling only, data placement only or scheduling and data placement).
- The layer of the software stack at which the approach is implemented (e.g., the operating system, the run-time system, the compiler, the application or a combination of these layers).
- Which information is used by the approach and how this information is obtained (e.g., hints by the application that on data placement, data affinities obtained through profiling, affinities derived from the structure of computations and data accesses in specific types of algorithms).
- To which programming model or framework the approach applies (e.g., independent processes, OpenMP or Cilk).
- The characteristics of the applications targeted by the approach (e.g., loop-level parallelism or algorithms that operate on arrays).

To our knowledge, only few approaches are specific to task-parallel applications executing on many-core NUMA systems. In this section, we present a set of related approaches relevant in the context of this thesis, covering the characteristics above. The presentation groups the approaches by the type of mapping, i.e., data placement and scheduling.

2.4.1 Data placement

The presentation of approaches for data placement below starts with a simple on-line page migration technique called AFFINITY-ON-NEXT-TOUCH that can be implemented in user space or by the operating system. We then introduce CARREFOUR, a more sophisticated kernel-space approach, which focuses on the avoidance of node contention through decisions for page migration, page replication and interleaving based on statistics gathered with hardware performance counters. An application-level approach for manual improvement is provided by MAI. The approach is used by the MINAS framework, which instruments the source code of an application to benefit from placement strategies from MAI automatically. The presentation finishes with two trace-based approaches. The first is based on full-system simulation and uses MINAS for data placement. The second approach performs off-line profiling using hardware performance counters and places data accordingly at execution time.

Affinity-on-next-touch

Many operating systems use *first-touch placement* as the default placement strategy, in which a page of physical memory is allocated on the node associated to the core that first writes to the page (a detailed discussion of data placement by the operating system will be given Section 4.1). If the cores that initialize data structures and those that access them are located on the same node,

first-touch placement yields high locality of memory accesses. However, if the initializing nodes and the accessing nodes do not match, this strategy may lead to high contention and a high fraction of remote memory accesses. A common strategy to circumvent this problem is to migrate pages dynamically after initialization to the nodes that perform the next write accesses. This strategy, referred to as AFFINITY-ON-NEXT-TOUCH or MIGRATE-ON-NEXT-TOUCH, can be implemented entirely in user space using system calls for memory protection and synchronous page migration or in kernel space for transparent, asynchronous migration. Löf and Holmgren [61] have evaluated a user space implementation of AFFINITY-ON-NEXT-TOUCH on an isolated domain of 8 nodes of a Sun Fire 15000 system running an application calculating the scattering of electromagnetic waves in a three-dimensional space mainly by solving a set of equations using the conjugate gradient method. Using AFFINITY-ON-NEXT-TOUCH the performance is improved by up to 166%, showing that data placement can have a huge impact on application performance.

Goglin and Furmento [50] have implemented AFFINITY-ON-NEXT-TOUCH for the Linux kernel and compared the performance to a user space implementation. The kernel-based implementation is about 30% faster on a four-node AMD Opteron 8347HE system and displays significantly less overhead than the user space implementation for small memory regions. However, the authors conclude that a user space implementation performs better in cases where larger memory areas known by the application have to be migrated. The kernel-space implementation migrates such areas page-by-page, whereas a user space implementation can migrate each of these areas in a single operation with lower overhead.

AFFINITY-ON-NEXT-TOUCH represents a simple and elegant way to migrate data to nodes on which it is accessed. However, it is up to the programmer or a software component on a higher level to trigger page migration. AFFINITY-ON-NEXT-TOUCH can be seen as a basic method for data placement that can be employed in more complex and more specific approaches for optimization.

Carrefour

Dashti et. al [44] have proposed CARREFOUR, a NUMA-aware data placement mechanism for the Linux kernel. Unlike other approaches for NUMA-aware data placement, CARREFOUR focuses on the avoidance of congestion on memory controllers and interconnect links and considers the reduction of the latency of memory accesses by improving data locality only as a secondary goal. The approach is based on four techniques:

- *page co-location* places a page on the same node than the accessing core,
- *page interleaving* places pages on nodes in a round-robin fashion,
- *page replication* replicates pages on multiple nodes and
- *thread clustering* co-schedules threads according to their intensity of data sharing.

Which combination of these techniques is used and how each technique is applied depends on the behavior of the applications that execute on the machine. This involves *global decisions* that enable or disable individual techniques globally and *page-local decisions* that enable or disable techniques per page. The statistics that serve as a basis to characterize program behavior are derived from values provided by a measurement component that uses INSTRUCTION-BASED SAMPLING [48] (IBS), a sampling mechanism available on recent AMD processors that provides detailed information on the execution of instructions (e.g., whether an instruction performs a memory access, whether the access targets local or remote memory, the duration of the access, etc.).

Global decisions are taken in four steps. In the first step, the system decides whether data placement is necessary or not. To this end, CARREFOUR compares the number of memory accesses per time unit of the entire system to an experimentally determined threshold of 50 accesses per microsecond. If the actual value for the application is below the threshold, CARREFOUR is disabled and no further actions are taken. The second step consists in deciding whether page replication should be enabled or disabled. To avoid high synchronization overhead due to frequent updates of the contents of pages, page replication is only used for applications whose fraction of read accesses to DRAM compared to the total number of accesses to DRAM is above 95%. In addition, there must be enough free memory before page replication to avoid that the replication causes pages to be swapped out to disk. In the third step, CARREFOUR checks whether interleaving should

be used to distribute requests to main memory to all memory controllers. This decision is based on the *memory controller imbalance*, which is defined as the standard deviation of the frequency of memory accesses among nodes. Interleaving is only applied if the value is higher than a threshold of 35%. The decision whether page co-location should be used is taken in the fourth and final step. Co-location is enabled for applications whose *local access ratio* is below 80%, i.e., the fraction of memory accesses that targets a local node is below 80%.

Page-local decisions are taken individually for each page by analyzing statistics that are derived from IBS samples that belong to instructions accessing the page. A page is *migrated* to a node if page migration is enabled globally and if the page is accessed only by cores of a single node. *Page replication* triggers if the mechanism is allowed globally and if the page has only been accessed by reading instructions. A page that is accessed by cores from multiple nodes in both read and write mode is placed using the *interleaving* mechanism that moves the page on the node with the smallest number of memory accesses per time unit in order to reduce contention.

The experimental evaluation of CARREFOUR has been conducted on two AMD Opteron systems machines with 16 and 24 cores, respectively, grouped into four nodes. The applications that have been used for this evaluation are the PARSEC BENCHMARK SUITE [18, 9] (version 2.1), the FACEREC facial recognition engine [10] (version 5.0), the METIS [5] benchmark suite and the NAS PARALLEL BENCHMARKS [6]. The performance of CARREFOUR have been compared to the default *first-touch* page placement strategy of the Linux kernel, interleaving across all nodes as well as the AUTONUMA patchset [41] for the Linux kernel, which migrates pages to the nodes of the accessing cores. For single-application runs, CARREFOUR performs significantly better than default page placement (up to $3.63\times$ faster). Compared to interleaving across all nodes, CARREFOUR performs significantly better in most cases and limits performance degradation in cases, where interleaving across all nodes degrades performance significantly compared to the default placement of the operating system (the maximum performance degradation of CARREFOUR is 4%). In comparison with AUTONUMA, CARREFOUR provides performance comparable results or performs significantly better. CARREFOUR fails to improve performance of applications with fast changes in behavior due to the limited sampling accuracy necessary for low-overhead sampling.

CARREFOUR shows that contention is an important issue on NUMA systems as optimizations decreasing contention result in significant improvement of the execution time. The approach is also an example of an optimization that reacts to actual behavior of an application at execution time and that is thus able to react to dynamic changes. Implementation at the operating system layer allows a wide variety of applications to benefit from the optimizations, but limits the granularity for data placement to entire pages of memory.

MAi

The MEMORY AFFINITY INTERFACE [77] (MAI) is an interface for data placement designed for high performance computing applications that operate on large arrays. The implementation of MAI provides seven policies for the distribution of the pages of an array: *bind_all*, *bind_block*, *cyclic*, *cyclic_block*, *skew_mapp*, *prime_mapp* and *random*:

- The *bind_all* policy places all pages on a single node and switches to other nodes only if all of the memory of the current node is in use.
- The *bind_block* policy first divides the array into blocks and then places each block on a different node.
- The *cyclic* policy distributes the pages of an array in a round-robin fashion over all nodes of the machine, such that the ith page is placed on the node whose identifier is $i \bmod M$, with M being the number of nodes.
- Similarly, the *cyclic_block* policy distributes blocks of subsequent pages on nodes in a round-robin manner, for example, the first two pages could be placed on Node 0, the third and fourth page on Node 1 and so on.
- The *skew_mapp* policy places the ith page of an array on node $n = (i + \lfloor \frac{i}{M} \rfloor + 1) \bmod M$.
- *Prime_mapp* combines two policies: first, it associates pages to P virtual blocks of data using the *cyclic* policy with P being a prime number and $P \geq M$. The second step consists of

distributing the virtual blocks to nodes using the *cyclic* policy again.
- The last policy used in the paper is *random*, which places pages randomly across nodes.

The purpose of the more complex policies *skew_mapp* and *prime_mapp*, originally proposed in [56], is to avoid node contention that results from very regular memory accesses and distributions by the *cyclic* or *cyclic_block* policy. For example, this is the case if an array is divided into equal-sized blocks whose size in pages is a multiple of the number of nodes. The *cyclic* policy would distribute the pages of the entire array, such that the distribution within each block is identical, which can lead to contention when the blocks are processed in parallel. The *skew_mapp* and *prime_mapp* policies yield different distributions for each block and thus avoid contention due to regular access patterns.

The evaluation of MAI has been conducted on systems with four and eight NUMA nodes for the *FFT* and *CG* applications from the OpenMP version of the NAS PARALLEL BENCHMARKS [57] as well as for an OpenMP implementation of a geophysics application [34]. The policies proposed by MAI can improve performance by up to 31% compared to the default *first-touch* policy of the operating system, but must be chosen manually. The authors have concluded that the best strategy for data placement depends on the target architecture as well as on the structure of memory accesses. Machines with a high difference between the latency of local and remote accesses benefit from data placement that optimizes for locality, such as *bind_block*, while execution on machines with a low difference between these latencies can be improved with placement that improves load balancing, such as *cyclic*, *random* or *skew_mapp*. Applications with a clear affinity of computations and data yield higher performance with *bind_block* and applications with irregular accesses benefit from distribution of data over nodes.

The results presented in the experimental evaluation of MAI show that the behavior of an application requires different kinds of distributions of data to nodes and highlights that the architecture also plays an important role in the selection of a placement strategy.

Minas

The conclusions drawn from the evaluation of MAI form the basis of the MINAS [76] framework, which combines the data placement capabilities of MAI with a preprocessor called MAPP and NUMARCH, a module that provides information about the target architecture. MAPP processes the source code of an application, finds shared, static arrays and replaces their declarations with appropriate calls to memory allocation and distribution functions of MAI. The actual policy for the data distribution chosen by MAPP depends on the characteristics of the NUMA platform reported by NUMARCH. For systems with a high remote access latency compared to the latency of local accesses, the *bind_block* policy is chosen in order to optimize for latency. On systems with lower remote access latency, the framework optimizes for bandwidth and uses the *cyclic* policy. In the experimental evaluation, the authors compare the performance of the automatic optimization with MINAS to the default page placement policy of the operating system as well as to hand-tuned versions of the applications using combinations of distribution policies that best match the data access patterns. The applications used for evaluation are the same as for the evaluation of MAI with an additional benchmark that simulates wave propagation in three dimensions. The automatic solution improves performance compared to the default page placement strategy of the operating system, but remains behind the performance of the hand-tuned codes. The difference between the automatic and the hand-tuned versions ranges from 0% to 25%.

MINAS is an effort to reduce the burden of the programmer to identify relevant data structures and to choose architecture-specific distributions. The results show that although the automatic approach cannot match the performance of hand-tuned code, automatic optimization can improve performance significantly.

Data placement with MINAS based on data sharing

In [66], the MINAS framework has been employed to improve data placement of applications from the C version of the NAS PARALLEL BENCHMARKS [15]. In a first step of this approach, the application is executed in a full-system simulator and its memory accesses are recorded to a trace

file. The trace file is then analyzed in order to generate sharing matrices that indicate for each pair of threads how intensively these threads communicate.

The approach uses two metrics to characterize communication, the first is based on the amount of memory that is accessed by two threads, while the second metric measures the number of accesses to shared memory blocks. The two metrics are evaluated separately and threads are grouped into pairs with maximum communication according to the metric. These pairs are then used to generate a second sharing matrix that captures communication between pairs of threads. The reason for this grouping is that caches are often shared between pairs of cores. Pairs with maximum communication can thus take advantage of the shared cache, reducing the mapping to a mapping of pairs of pairs of threads optimizing off-cache communication. However, the number of cores per cache can be higher than two and more complex groupings than pairs might be necessary for an optimal mapping. According to the authors, using pairs is still a reasonable approximation in these cases.

The experimental evaluation has been conducted on an AMD Opteron 875 system with 8 NUMA nodes and 16 cores in total, as well as on an Intel Xeon X7560 system with 4 NUMA nodes and 32 cores in total. It showed that for thread and data mappings based on the sharing matrix, significant improvements on the execution time of up to 75% can be achieved over default thread and data mapping of the operating system. These performance gains have been achieved for applications that initialize data sequentially, and for which the default page placement policy of the operating system allocates all pages on the same node. For applications that initialize data in parallel, no significant speedup has been achieved. For the different types of applications and platforms the choice of the metric for the sharing matrix (amount of memory or number of accesses) did not have a significant impact.

The approach shows that profiling can be used to obtain detailed information on data exchanges between threads, which can be exploited for improved data placement of applications with distinct patterns for memory accesses.

Feedback-directed page placement for OpenMP

Marathe et al. have proposed trace guided placement of pages for OpenMP programs [62]. The approach is divided into three phases: *trace generation*, *affinity decision* and *trace-guided page placement*. During trace generation, the framework executes a truncated version of the program whose data placement is to be improved. The framework samples detailed information about memory accesses and memory allocations by using the processors' performance monitoring units and by intercepting calls to the memory allocator of the C / FORTRAN run-time library. The subsequent affinity decisions after the sampling consist in determining on which node each page should be placed, based on the accesses from the trace. The framework provides a simple model, in which the latency of a remote access is assumed to be independent from the distance between the requesting core and the node that satisfies the request as well as a more sophisticated model that takes into account varying latencies. In the former model, a page is associated to the node with the highest number of accesses to the page. The latter model determines the *aggregate access cost* for each node and associates a page with the node with the minimal cost. The aggregate access cost corresponds to the sum of the products of the number of accesses from a node and the cost of an access.

The actual page placement is carried out at execution of the entire program by intercepting calls to the memory allocator and by initializing pages on the appropriate node before handing the memory regions to the application. The authors have shown that for applications of the C version [7] of the NAS PARALLEL BENCHMARKS [15] and applications from the SPEC OMPM2001 [14] benchmarks, the number of remote memory accesses and thus the execution time can be decreased significantly on a NUMA system with four nodes. The authors have also pointed out that simple round-robin page placement on the available nodes using LIBNUMA [58] yields similar results.

Feedback-directed page placement is another approach that exploits information obtained through profiling for improved data placement. However, in contrast to MINAS with profiling, the feedback-directed page placement does not rely on a full system simulator, but uses mechanisms in the run-time library as well as hardware support. The results also illustrate that interleaving across

NUMA nodes can also be effective and that sophisticated mechanisms are not always needed.

2.4.2 Scheduling

The following approaches rely on scheduling as the main mechanism to improve the performance of memory accesses. We first present SCHEDULE REUSE, which schedules loop iterations to cores to match the node associated to the core executing an iteration with the node whose memory contains the data that is accessed during the iteration. The second approach covers scheduling in applications that do not impose any specific order for task execution.

Schedule reuse

Nikolopoulos et al. [68] have proposed an approach that addresses NUMA-aware scheduling of loop iterations in OpenMP applications with irregular accesses to main memory. The first example of irregularity handled by the approach results from loop nests, where the iteration space of an inner, parallel loop depends on the index of an outer loop, such that the assignment of iterations of the inner loop to processors changes between iterations of the outer loop. For data structures, such as arrays, that have been distributed to nodes using a regular structure, e.g., in blocks or by interleaving across nodes, this leads to a mismatch between the nodes to which the loop iterations are assigned and the nodes that contain data. The other example given in the paper contains parallel loops that perform array accesses whose array indexes are calculated from the loop indexes by indexation of an indirection table. In addition, the array itself can be distributed irregularly, e.g., with blocks of different size resulting from a generalized block distribution.

The main idea of the approach is to first distribute the pages of a data structure accessed by a loop nest over the nodes of the system using an application-specific description of the distribution provided by the programmer through code annotations and to schedule loop iterations accessing the data on the same nodes. The actual placement of pages is implemented by scheduling the iterations first accessing the data structure to nodes according to the description, such that first-touch allocation places the pages appropriately. Data locality is addressed by assigning subsequent iterations accessing the data to the processors associated to the containing nodes.

The key aspect for local accesses after the distribution is the construction of a two-dimensional array that contains one column for each processor, each of which contains the loop indexes that result in accesses to the local memory of the processor according to the data distribution. Hence, to determine the set of iterations that should be carried out by a specific processor, it is sufficient to iterate over the corresponding column in the array. The loop in the original program is then replaced by an outer loop with indexes from the lowest to the highest node identifier and an inner loop that iterates over the elements of the column that is associated to the node. The outer loop is executed in parallel, with one loop index assigned to every processor.

The authors did not describe any formal method to construct the array and to transform loops, but suggested that an optimizing compiler could carry out this task.

The approach has been evaluated on an application that performs LU decomposition of dense matrices and several kernels from a weather forecast system performing transpositions between grid spaces with irregular densities. All experiments have been conducted on an SGI Origin 2000 system with 64 processors grouped into 32 nodes. For LU decomposition, three versions have been compared: an unmodified OpenMP version, a modified OpenMP version with data distribution directives supported by the SGI compiler and the SCHEDULE REUSE approach of the authors. The kernels from the weather forecast system have been implemented using OpenMP, with the SCHEDULE REUSE approach and with explicit data partitioning and message passing using MPI. The conclusions from the experiments are that the *schedule reuse* approach outperforms both OpenMP versions for LU decomposition (up to two times faster than the unmodified OpenMP version), that it outperforms the unmodified OpenMP versions of the irregular kernels and that it provides performance comparable to the MPI versions.

SCHEDULE REUSE illustrates that detailed, static information about data accesses, derived from the source code of the application can be combined with a description of an architecture-specific data distribution. This aspect is particularly important to address irregular data distributions and

irregular accesses to memory.

Scheduling of unstructured parallelism

The optimizations proposed by Yoo et al. in [86] address applications with unstructured parallelism, i.e. parallel sections with independent tasks that can be scheduled in any order. The framework for task-parallelism used in the paper is a custom run-time library for task-parallelism provided by the authors. Although there are no explicit data dependences between tasks in these applications, two or more tasks can share data if they access a common set of addresses. The authors did not address NUMA issues directly and focused on cache performance by executing tasks that share data on cores that are near in the memory hierarchy. However, the approach is relevant for this discussion of related work as it shows how information on fine-grained data sharing can be exploited in the run-time system of a task-parallel language.

The approach consists of three major parts. In the first part, the workload is profiled in order to derive information on data sharing between tasks. The second part consists in grouping tasks, ordering groups and assigning the groups to work-queues associated to the components of the memory hierarchy. The third part addresses dynamic load balancing through locality-aware task-stealing.

The result of the profiling run is a *task sharing graph* whose vertexes represent tasks and whose undirected edges capture data sharing relations between tasks. The weight associated to an edge indicates how many cache lines are accessed by the two tasks that are connected by the edge. The graph is then partitioned heuristically and recursively into groups for each level of the memory hierarchy, starting with the last level cache. Each task group is chosen such that the working set of the tasks fits into a cache of the targeted level in the memory hierarchy and such that intra-group data sharing is maximized. The result is a hierarchy of task groups that can be scheduled over the work-queues associated to each component of the memory hierarchy, e.g., the result can be a hierarchy with task groups for the work-queues of first level caches that belong to groups for the work-queues of second level caches that in turn belong to a groups for the work-queues of third level caches and so on.

At execution time, the tasks are executed by worker threads with one worker thread per core. When a worker has finished executing a task, it first tries to obtain a task from its local queue associated to its first-level cache. If this queue is empty, the worker attempts to steal tasks from one of the queues associated to the first-level cache of its sibling cores with respect to the next level in the memory hierarchy (i.e., the cores sharing a second-level cache with the core of the stealing worker). If these queues are also empty, the worker tries to obtain a task group from the queue associated to its second level cache.

The authors have evaluated their approach on a set of general-purpose workloads (database, 3d image reconstruction, collision detection, image processing, matrix multiplication and a solver for partial differential equations) ported to the author's framework for task parallelism executing on three simulated systems with core counts ranging from 32 to 1024. Besides performance evaluation of the entire approach with fixed parameters, the authors have also explored different values (e.g., how many tasks are stolen by a single attempt for work-stealing) and different heuristics (e.g., group and task ordering, whether to use locality-aware work-stealing or not or the heuristic for the selection of the victim for a steal). However, we only provide a summary of the results without reproducing all the details. Task grouping, ordering and assignment to the queues, but without locality-aware work-stealing can speed up execution by up to $2.39\times$ on 32 cores for memory-intensive benchmarks and up to $3.57\times$ on 1024 cores. Locality-aware work-stealing on 32 cores can speed up execution by as much as $1.9\times$ compared to random work-stealing with artificial load imbalance created through a number of workers that is smaller than the number of cores.

The approach shows that exploiting data sharing in the scheduler can lead to significant performance improvements with higher benefits for larger systems. It also illustrates that initial assignment of tasks can be combined with a locality-aware technique for load-balancing that reacts to the circumstances at execution time. Our approach for topology-aware scheduling presented in Section 7.3 uses a similar technique.

2.4.3 Combined scheduling and data placement

For the presentation of related work combining NUMA-aware scheduling and data placement, we have selected three approaches. The first approach, FORESTGOMP, relies on precise hints on affinities between OpenMP threads and data provided by the programmer. The second approach targets algorithms operating on arrays and relies on the specification of a more general pattern for the distribution of array elements to nodes provided in the source code. The last approach, LAWS, is designed for *divide-and-conquer algorithms* and does not require any modification of the source code of the application.

ForestGOMP

Broquedis et al. have proposed FORESTGOMP [29, 31], an OpenMP run-time with a resource-aware scheduler based on the BUBBLESCHED [83] scheduler and a NUMA-aware allocator based on the MAMI [32, 4] memory interface. FORESTGOMP uses three key concepts: grouping of OpenMP threads into *bubbles*, scheduling of threads and bubbles using a hierarchy of *run-queues* and dynamic migration of data upon load balancing.

At the beginning of the execution, FORESTGOMP extracts information about the memory hierarchy of the target platform automatically using HWLOC [30] and creates a hierarchy of run-queues reflecting this topology. For example, the run-time system might create one run-queue for the entire machine, one run-queue for each NUMA node, one run-queue for each shared cache and one run-queue for each core. Each of the run-queues forms a scheduling domain that restricts the execution of *scheduling entities* in the queue to the part of the memory hierarchy that the queue is associated to. The scheduling entities used by the BUBBLESCHED scheduler are OpenMP threads and *bubbles*. Bubbles are groups of threads or nested groups of bubbles and express data sharing among threads or access of a group of threads to data on the same node. The threads that form a bubble are kept together as long as possible to avoid that threads accessing the same data are scattered across the entire machine.

The creation of bubbles is carried out by the run-time system and takes place every time a parallel section is encountered. The set of threads that forms a bubble is identical with the team of threads of a parallel section. Nested parallel sections, i.e., parallel sections encountered within a parallel section, lead to the creation of nested bubbles aggregating other bubbles.

Resource-aware scheduling in FORESTGOMP is implemented using two scheduling algorithms: the *memory bubble scheduler* and the *cache bubble scheduler*. The NUMA-aware scheduler relies on so-called *memory hints* that summarize which data regions will be accessed by a single thread or a team of threads. Memory hints are provided by the programmer by calling appropriate functions of the run-time system before creating a parallel section or from a thread within a parallel section. The run-time attaches information derived from these hints to threads and bubbles, which allows the scheduler to distribute threads accordingly. In a first step, each thread is associated to the node that holds the highest fraction of the thread's data among all nodes. Bubbles that contain threads which are associated to different nodes *explode* and their threads are distributed accordingly. If the distribution resulting from the first step leads to an imbalance between cores, the run-time relaxes the scheduling constraints for certain threads and associates them to unused cores. To limit the overhead associated to data migration from one node to another, the run-time chooses the threads with the least amount of attached data for redistribution. Once the distribution of threads has finished, the system migrates the data of relocated threads to the right nodes. The distribution of threads and data to nodes is thereby completed and the *cache bubble scheduler* can start distributing threads within each node.

The goal of the *cache bubble scheduler* is to maximize data reuse within caches. Three methods are combined for this purpose. First, the *cache bubble scheduler* attempts to place the threads of a bubble on cores that are close in the cache hierarchy, e.g., on cores sharing the same cache, as threads of the same bubble are likely to access a common set of memory regions. Second, the scheduler tries to exploit temporal locality by restoring the mapping of threads to locations from the previous invocation of the scheduler. Finally, a core that becomes idle first tries to steal a thread

from a nearby core before trying to steal a thread from a remote core.

In order to be able to react to changes in program behavior, the run-time allows the application to update memory hints during execution of a parallel region. Every time a hint is updated, the scheduler is invoked to check the current distribution of threads and to redistribute threads appropriately if necessary. To detect changes of affinities not indicated by the programmer, FORESTGOMP also monitors hardware performance counters and invokes the scheduler automatically if the fraction of remote memory accesses exceeds a certain threshold.

The evaluation of FORESTGOMP was carried out on a modified version of the STREAM BENCHMARK [63] named TWISTED-STREAM and an application performing LU decomposition executing on an AMD Opteron platform with four NUMA nodes.

The TWISTED-STREAM benchmark operates on three vectors A, B and C, divided into M blocks $A_i, \ldots, A_M$, $B_i, \ldots, B_M$ and $C_i, \ldots, C_M$, with M being the number of nodes of the machine. A team of threads is created for each node and each team operates on three blocks, one from each vector (the first team operates on blocks A_0, B_0 and C_0, the second on blocks A_1, B_1 and C_1, etc.). In the middle of the execution the affinities between threads and data change. For the configuration referred to as TWISTED-66, each team changes the affinities for two of the three blocks (e.g., the first team operates on A_0, B_1 and C_1, the second on A_1, B_2 and C_2, etc.). In another configuration, called TWISTED-100 each team changes the affinities for all of its blocks (e.g., the first team operates on A_1, B_1 and C_1, the second on A_2, B_2 and C_2, etc.).

The performance of FORESTGOMP on the TWISTED-STREAM benchmark is compared to the OpenMP run-time of the GNU C compiler named LIBGOMP [40] and to page migration using AFFINITY-ON-NEXT-TOUCH. For TWISTED-100, FORESTGOMP only needs to adjust the distribution of threads to nodes without migration of data, resulting in 25% less execution time compared to LIBGOMP. The gain over page migration depends on the number of iterations that the benchmark performs after the change of affinities, as the relative overhead for page migration decreases with each iteration. The speedup over page migration ranges from $7.9\times$ for a single iteration to $1.3\times$ for 128 iterations. For TWISTED-66 FORESTGOMP has to migrate one third of the data. For less than three iterations, FORESTGOMP is thus slower than LIBGOMP, but still faster than AFFINITY-ON-NEXT-TOUCH. For more than three iterations, FORESTGOMP outperforms both LIBGOMP and AFFINITY-ON-NEXT-TOUCH.

Data affinities in the application performing LU decomposition are more complex and change more frequently than for the STREAM benchmark. Hence, instead of giving precise hints for affinities, the authors have configured FORESTGOMP to mark the entire matrix for AFFINITY-ON-NEXT-TOUCH each time the fraction of remote memory accesses measured with hardware performance counters exceeds a certain threshold. This results in a 30% decrease of execution time compared to interleaving of the matrix across all nodes of the machine.

In conclusion, FORESTGOMP performs best with clear affinities between threads and data and if locality for changing affinities can be restored through scheduling without migration. The approach shows that information that is missing in a more abstract software layer, i.e., the run-time, can be compensated by propagating more detailed information from the application.

Data distribution based on node arrangements

Bircsak et al. have proposed a set of directives for NUMA-aware programming in OpenMP [19]. The approach mainly addresses data placement for arrays, but also provides directives that define constraints for the placement of computations. The distribution of the elements of an array requires two parts: a (possibly) multi-dimensional arrangement of the nodes of the system in the form of a matrix containing the node identifiers and a set of distribution policies specifying one policy for each dimension of the array to be distributed. The arrangement does not necessarily represent any physical relationship of nodes and serves only as an auxiliary data structure to determine the mapping of array elements to nodes. The role of a distribution policy is to (1) partition the range of a dimension of the array whose mapping is to be determined and (2) to define the set of nodes that is available for each partition. The combination of the policies from the outermost to the innermost dimension and the arrangement of nodes defines for each element of the array to which node the element will be associated. There are three policies to choose from: *BLOCK*, *CYCLIC* and *. The

implications of these policies are the following:

- The *BLOCK* policy divides the range of an array dimension into equal-sized blocks and creates equal-sized partitions from the respective dimension of the node arrangement.
- The *CYCLIC* policy associates to each element of the array dimension an element of the respective dimension of the node arrangement in a round-robin fashion.
- Finally, the * policy defines that the respective dimension should not be partitioned at all.

The following examples are adapted from the examples given in [51] and illustrate the use of distribution policies and node arrangements. As a first example, consider a one-dimensional array with 1024 elements that is to be mapped to a total of eight nodes $N0$ to $N8$. Using the *BLOCK* policy, the array is divided into 8 blocks of 128 elements. The first 128 elements are mapped to the first node, the second 128 elements to the second node and so on, as illustrated in Figure 2.6a. If the *CYCLIC* policy is used, the elements are distributed in a round-robin fashion to the nodes, such that the ith element is placed on the $(i \mod 8)$th node, as shown in Figure 2.6b.

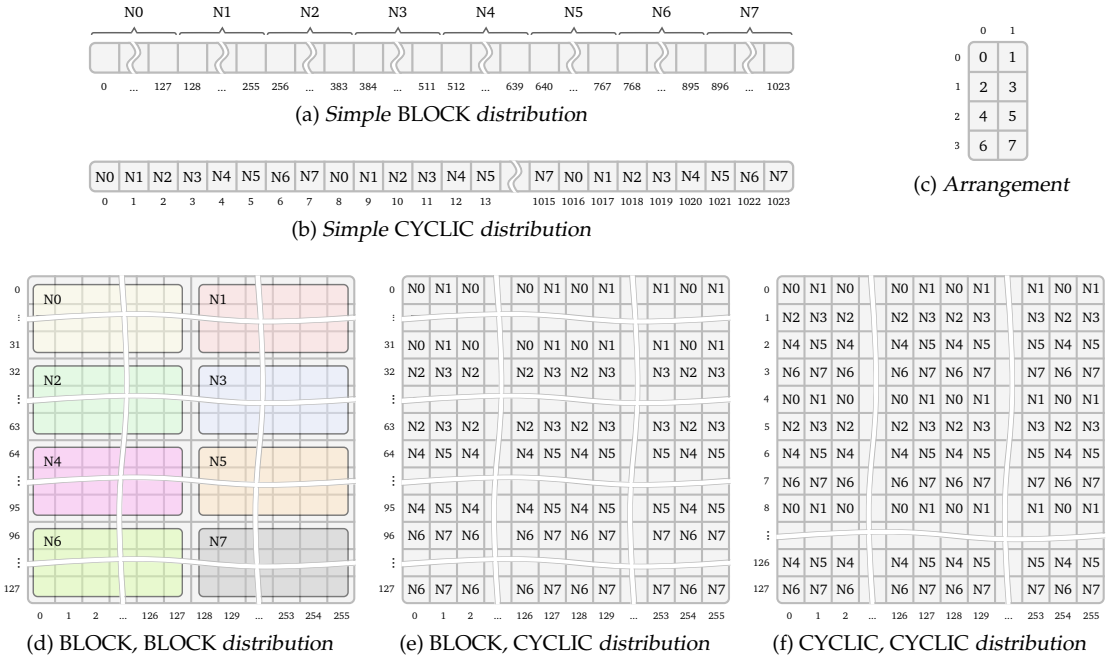
More complex distributions can be defined by combining several policies and by changing the arrangement of nodes. As an example, consider a two-dimensional array with 128 rows and 256 columns and an arrangement of eight nodes in a 4×2 matrix as in Figure 2.6c. By using a *(BLOCK, BLOCK)* policy, the array is divided into four rows of 32 elements and each row is divided into two columns of a width of 128 elements. Hence, the matrix is divided into blocks of size 32×128, each associated to a different node, as shown in Figure 2.6d. The block in the first row and the first column is placed on the node at the first row and first column of the arrangement, the block in the first row and second column to the node at the first row and second column of the arrangement and so on. For a *(BLOCK, CYCLIC)* distribution, the matrix is divided into eight rows and the columns of each row are distributed in a round-robin fashion among the nodes of the second dimension of the arrangement. Hence, the ith element in the jth row of the matrix is placed on the node at the ith row and the $(j \mod 2)$th column of the arrangement. Figure 2.6e illustrates this distribution. As a last example, consider a *(CYCLIC, CYCLIC)* distribution that associates both columns and rows of the matrix with columns and rows of the arrangement. In the resulting distribution the ith element in the jth row of the matrix is placed on the node at the $(i \mod 4)$th row and the $(j \mod 2)$th column of the arrangement as illustrates by Figure 2.6f.

The approach also allows the programmer to define the granularity of a distribution by specifying if the distribution applies to elements or pages. In the first case, the partitioning of the array is done with element granularity as in the examples above. In the latter case, elements are grouped into pages and the pages are mapped to nodes according to the distribution.

To turn accesses to an array within a loop nest into accesses to local memory, the iterations must be distributed to cores of the NUMA nodes according to the distribution. This is achieved with help from the compiler as well as support by the run-time. In a first step, the compiler parametrizes the loop bounds and loop increments with the node identifier, such that the cores of a node only perform the loop iterations that result in accesses to the node's local memory. In the second step, the compiler partitions the iteration space of the outermost loop among the cores of a NUMA node. If this loop has only few iterations, the programmer can specify that another loop in the loop nest should be chosen for the partitioning among cores. The partitioning can be achieved using a cyclic assignment of loop indexes to cores or by dividing the iteration space into blocks that are each assigned to a different core. The role of the run-time system is to pin threads on the correct cores and to make sure that the iterations are assigned correctly to the appropriate threads.

If the assignment of loop iterations to nodes is not optimal, e.g., if multiple arrays are involved in a computation, the programmer can specify explicitly on which node a computation should be carried out. This is done by using the *ON HOME* directive and by passing an reference to an array element as a parameter. The node to which the element is associated to defines where the computation should take place.

However, which combination of policies yields optimal performance depends on the pattern of accesses to array elements. In some cases, it is easier to rely on dynamic page migration. This is supported through two directives, namely *MIGRATE_NEXT_TOUCH* and *MIGRATE_TO_OMP_THREAD*. The former causes the pages that form an array to be migrated to the nodes of the first



(a) *Simple* BLOCK *distribution*

(b) *Simple* CYCLIC *distribution*

(c) *Arrangement*

(d) BLOCK, BLOCK *distribution*

(e) BLOCK, CYCLIC *distribution*

(f) CYCLIC, CYCLIC *distribution*

Figure 2.6: *Examples of distributions using* BLOCK *and* CYCLIC

cores modifying the pages after the encounter of the directive. The latter directive migrates pages to the node of the core executing the thread with the thread identifier that is passed as a parameter to the directive.

By comparing different versions of an application performing LU decomposition without pivoting on a 32-core ALPHASERVER GS320 system, the authors have concluded that data layouts using the *DISTRIBUTE* directive employing the concepts of distribution policies and node arrangements above perform substantially better than dynamic page migration due to the overhead of the migration.

The approach shows that explicit data distributions for regular data structures provided by the programmer can be combined with very fine-grained scheduling of instructions with appropriate support by the compiler. However, deciding which distribution policy fits best and specifying an appropriate node arrangement can be a difficult task that requires detailed knowledge of the application. In addition, arrangements must be developed for each of the systems on which an application is intended to execute in order to match the number of nodes of the system.

LAWS

Chen et al. have proposed LAWS [38] that combines NUMA-aware allocation and scheduling with cache-aware scheduling of CILK tasks from earlier work [39] of the authors. The approach targets *divide-and-conquer* algorithms with the following characteristics: the recursive steps of the algorithm are represented by a tree of tasks in which each node represents one step, data accesses only happen in leaf tasks and data sharing is likely to occur between siblings in the tree. Furthermore, the program is assumed to perform multiple iterations with identically structured task trees and equal relationships between tasks and the memory regions accessed during task execution. LAWS has three main components: a NUMA-aware memory allocator, an *adaptive DAG packer* and a NUMA-aware and cache-aware work-stealing mechanism.

NUMA-aware data placement in LAWS is carried out during the first iteration of the algorithm by assigning the data sets of recursively created tasks of the task tree to NUMA nodes using the following scheme. The root task of the tree for the iteration is assumed to represent the entire computation performed by all tasks of the iteration and is thus associated to the entire data set

of size D. At each recursive task creation below the root, the data set is assumed to be divided equally among tasks. Hence, if the root task has n child tasks, the children are each assumed to treat a data set of size $\frac{D}{n}$, with the first task treating the interval $[0; \frac{D}{n})$, the second task treating the interval $[\frac{D}{n}; \frac{2D}{n})$ and so on. The same scheme is applied recursively to the resulting intervals and the tasks located in the sub-trees of the children of the root task. NUMA-aware data placement for M nodes consists in partitioning of the entire data set into M equal-sized intervals, allocating each part on a different node, and associating tasks to nodes by analyzing the data intervals that the task represent. For example, the data set of a task that represents an interval $[a; b)$ with $a \geq \frac{i \cdot D}{M}$ and $b < \frac{(i+1) \cdot D}{M}$ should be placed on node i. The actual data placement makes use of *first-touch placement*, which is the default placement policy of many operating systems. To achieve the placement of the M intervals as described above, this implies that the tasks associated to the intervals must be executed by cores of the appropriate nodes during the first iteration. That is, a task whose data interval should be allocated on node i must be executed by a core of node i.

The role of the adaptive DAG packer is to provide the run-time system with information for cache-aware task scheduling. In a first step, the working set of each task is estimated by measuring the number of misses in the last level private cache during the execution of the task and by multiplying this value with the size of a cache line. The measurement of cache misses is performed at the first iteration of the algorithm using hardware performance counters. Based on this information, the task tree is divided into so-called *CF subtrees*. A *CF subtree* is a tree for which the sum of the working sets of its tasks fits into the shared cache of a node. The tasks of a same *CF subtree* are called *socket local tasks* and will be scheduled to cores of the same node in order to favor inter-task communication in the shared cache and to limit the number of capacity misses. However, the estimation of the data set size of a task only approximates the actual size of the data set and the packing into *CF subtrees* might yield suboptimal performance. Therefore, the DAG packer adapts the packing from one iteration to another. The packing can be shrinked by breaking a *CF subtree* into smaller trees by declaring the child tasks of the subtree's root as roots of *CF subtrees*. Similarly, it can be coarsened by aggregating *CF subtrees* by declaring their common parent task as the root of a *CF subtree*. The adaptive packer compares the execution time of packings and shrinks or coarsens the packing until a point is reached where a finer and a coarser packing both yield longer execution times than the current packing.

The last component of LAWS is a triple-level hierarchical work-stealing scheduler that schedules tasks to their associated nodes and that ensures that the *socket local tasks* of a *CF subtree* are executed by cores of the same node. Each node is provided with a *CF task pool* that can only contain *CF root tasks* and each core has a *socket local task pool* which can only contain *socket local tasks*. When a core runs out of tasks, it first tries to steal a task from the *socket local task pool* of the cores of the same node. If none of these pools contains a task that can be stolen, the core tries to steal a task from the *CF task pool* of the local node. If this attempt also fails, the core tries to steal a task from the *CF task pool* of another node. To avoid concurrency in the shared cache due to the execution of tasks from multiple *CF subtrees*, the scheduler also ensures that all tasks of a *CF subtree* have been executed before execution of a new *CF subtree* starts.

The authors have evaluated *LAWS* on a four-node AMD Opteron 8380 system executing a set of applications that perform stencil computations and algorithms for Gaussian Elimination as well as successive over-relaxation. Each benchmark is available in two versions. The first version has a regularly structured execution DAG, while the computations of the second version form an irregularly structured DAG. The performance of LAWS has been compared to plain CILK without any modification and to an algorithm from earlier work [39] named CATS, which does not adapt the packing after the first iteration. The improvement of LAWS over plain CILK ranges from 23.5% to 54.2%. LAWS systematically outperforms CATS, which improves performance over CILK only by up to 19.6%.

LAWS shows that implicit information on the structure of computations as well as the data structures involved in the computations can be exploited both for data placement and scheduling to increase the locality of memory accesses fully automatically by the run-time system.

	Optimization for NUMA	Optimization for caches	Data placement	Scheduling	Implementation layer
AFF.-ON-NEXT-TOUCH [61]	✓	-	✓	-	Library
AFF.-ON-NEXT-TOUCH [50] (OS)	✓	-	✓	-	OS
CARREFOUR [44]	✓	-	✓	(✓)*	OS
MAI [77]	✓	-	✓	-	Library
MINAS [76]	✓	-	✓	-	Preproc. + Library
MINAS + profiling [66]	✓	✓	✓	(✓)	Preproc. + Library
Feedback-directed placement [62]	✓	-	✓	-	Library
SCHEDULE REUSE [68]	✓	-	(✓)**	✓	Comp. + run-time
Unstructured parallelism in [86]	-	✓	-	✓	Run-time
FORESTGOMP [29, 31]	✓	✓	✓	✓	Run-time
Node arrangements [19]	✓	-	✓	✓	Comp. + run-time
LAWS [38]	✓	✓	✓	✓	Run-time

* Part of CARREFOUR, but not evaluated in the paper ** Data is only placed initially and the approach focuses on scheduling

Table 2.1: *Overview of basic characteristics of approaches in related work*

2.4.4 Summary

The above presentation on related work illustrates that there are many different existing approaches to improve data locality with respect to NUMA, to reduce contention on memory controllers and interconnects and to improve the exploitation of caches through optimized scheduling and data placement, resulting in significant improvements on performance. Although the discussion only presents a small part of research in this area, the selected publications cover a certain range of methods for the placement of data and instructions. The purpose of this section is to summarize the approaches and to highlight the differences between them. The summary is divided into three parts. First, we provide a overview of general characteristics. The second part focuses on features related to optimized data placement and the third part summarizes characteristics of the approaches for scheduling.

Main characteristics

The general characteristics of the approaches can be summarized as follows:

- the part of the memory hierarchy that the approach optimizes for, i.e, whether it aims at improving accesses to main memory or accesses to caches.
- the implementation layer of the approach (e.g., a user space library to be used by a programmer, the operating system or the compiler).
- the supported framework for parallelism (e.g., OpenMP, POSIX threads or Cilk).
- whether the approach is based on optimized scheduling, optimized data placement or whether it combines both of the techniques.

Table 2.1 details these points for all approaches presented earlier. Except the approach for unstructured parallelism, all of the papers we have selected provide NUMA-related optimizations. In addition, MINAS with profiling, LAWS and FORESTGOMP also implement cache-related optimizations. Optimized data placement is a cornerstone of the optimizations for the majority of the approaches. The only exceptions are SCHEDULE REUSE, which only places data initially via the first-touch mechanism for deterministic behavior with respect to data locality for subsequent scheduling and the approach for unstructured parallelism, which does not provide any form of explicit data placement at all. Optimized scheduling is provided by the last five approaches in the table, namely SCHEDULE REUSE, the approach for unstructured parallelism, FORESTGOMP, node arrangements and LAWS.

The principal layers of implementation are user space libraries (AFFINITY-ON-NEXT-TOUCH, MAI and feedback-directed page placement), the operating system (the kernel version of AFFINITY-ON-NEXT-TOUCH and CARREFOUR) and the run-time system (the approach for unstructured parallelism, FORESTGOMP and LAWS). Some approaches cover two layers, in particular MINAS, which provides a preprocessor and also relies on the user space library of MAI and SCHEDULE REUSE as well as node arrangements, which combine optimizations in the compiler with a modified run-time.

	Determination of relevant data by	Supported data structures	Granularity	Time of decision	Autom. and dyn. adjustment	Passive (P) / Active (A)
AFF.-ON-NEXT-TOUCH [61]	Programmer	Any	Pages	Execution	-	P
AFF.-ON-NEXT-TOUCH [50] (OS)	Programmer	Any	Pages	Execution	-	P
CARREFOUR [44]	Profiling	Any	Pages	Execution	✓	P
MAI [77]	Programmer	Arrays	Blocks / pages	Compil. + Exec.	-	A
MINAS [76]	Preprocessor	Static arrays	Blocks / pages	Compil. + Exec.	-	A
MINAS + profiling [66]	Preprocessor	Static arrays	Blocks / pages	Compil. + Exec.	-	A
Feedback-directed placement [62]	Profiling	Any	Pages	Before execution	-	A
SCHEDULE REUSE [68]	Programmer	Arrays	Elements	Compile-time	-	A
Unstructured parallelism in [86]	-	-	-	-	-	-
FORESTGOMP [29, 31]	Programmer	Not specified	Not specified	Execution	✓	P+A
Node arrangements [19]	Programmer	Arrays	Elements / Pages	Compile-time	-	A
LAWS [38]	Task graph	Arrays	Blocks	Execution	-	A

Table 2.2: *Overview of the features of data placement in related work*

Characteristics of data placement

The approaches that support data placement differ in:

- how they determine which data is relevant for improved data placement, in particular whether this is done automatically or if the programmer needs to specify relevant data structures.
- which kinds of data structures are supported (e.g., arrays or any contiguous memory region).
- the granularity for data placement (e.g., single data elements, pages or blocks representing large memory regions).
- the time when the decision for placement is taken (e.g., dynamically at execution time, statically at compile time or during off-line profiling).
- the kind of data placement, i.e., passive placement that reacts to accesses and places data accordingly or active placement that places data before it is accessed.
- whether data that has once been placed can be migrated dynamically to react to dynamic changes at execution time.

Table 2.2 indicates that many of the approaches either rely on the programmer to determine which data regions are relevant for data placement or determine these regions using profiling. MINAS uses a preprocessor for this task, but is limited to statically allocated memory and LAWS derives this information from the task graph, assuming that data is partitioned equally among leaf tasks. Low-level approaches, i.e., AFFINITY-ON-NEXT-TOUCH, CARREFOUR and feedback-directed page placement can handle any data structure, but the granularity for data placement is limited to entire pages of memory. The other approaches handle arrays with varying granularity for the placement: while MAI-based solutions distribute blocks of memory to different nodes, SCHEDULE REUSE and node arrangements can handle individual elements of an array due to support by the compiler. LAWS always places blocks of memory whose size depends on the number of tasks operating on the array and the size of the array.

Regarding the moment at which the decision for data placement is taken, the approaches can be divided into three sets. AFFINITY-ON-NEXT-TOUCH, CARREFOUR, FORESTGOMP and LAWS take this decision at execution time. In contrast to this, data placement decisions in SCHEDULE REUSE and node arrangements are the taken at compile time as the placement is specified in the application's source code. Feedback-directed page placement determines this relationship after the profiling phase and before the start of the execution. The MAI-based approaches represent an intermediate form in which some decisions are taken at compile time, i.e., the type of data distribution, and others are taken at execution time, e.g., the exact set of nodes for a distribution.

Another difference between the approaches consists in the duration of the placement. In most of the approaches data that has once been placed in is never migrated, unless explicitly requested by the application. However, CARREFOUR and FORESTGOMP can react to dynamic changes in application behavior and relocate data from one node to another.

	Type of placement	Source for placement decision	Scheduling entity	Time of decision	Autom. and dyn. adjust-ment
AFF.-ON-NEXT-TOUCH [61]	-	-	-	-	-
AFF.-ON-NEXT-TOUCH [50] (OS)	-	-	-	-	-
CARREFOUR [44]	(Thread clustering)*	(OS / PMU)*	(OS threads)*	(Execution)*	(✓)*
MAI [77]	Thread pinning	-	Pthreads	Start of exec.	-
MINAS [76]	Thread pinning	-	Pthreads	Start of exec.	-
MINAS + profiling [66]	Co-scheduling	Data sharing	Pthreads	Start of exec.	-
Feedback-directed placement [62]	Thread pinning	-	Pthreads	Start of exec.	-
SCHEDULE REUSE [68]	Loop scheduling	Data distrib.	Loop iterations	Execution	-
Unstructured parallelism in [86]	Task placement	Data sharing	Tasks	Start of exec.	✓
FORESTGOMP [29, 31]	Thread placement	Data distrib.	OpenMP threads	Execution	✓
Node arrangements [19]	Loop scheduling	Data distrib.	Loop iterations	Before loop start	-
LAWS [38]	Task placement	Data distrib.	Cilk tasks	Execution	✓

* Part of CARREFOUR, but not evaluated in the paper

Table 2.3: *Overview of the features of scheduling in related work*

Finally, the approaches can be classified according to the type of placement. Passive methods, such as AFFINITY-ON-NEXT-TOUCH and CARREFOUR react to the events of a given placement and try to improve the mapping of data to nodes accordingly. The other approaches actively place data before it is referenced. FORESTGOMP supports both active data placement, e.g., when migrating threads and attached data, as well as passive placement when using AFFINITY-ON-NEXT-TOUCH.

Characteristics of scheduling mechanisms

To distinguish the approaches for scheduling we have identified the following characteristics:

- the type of placement (e.g., simple thread pinning, loop scheduling).
- the source of information on which placement decisions based (e.g., performance monitoring units, information on the data distribution established by the approach itself or data sharing between threads).
- the type of scheduling entities handled by the approach (e.g., operating system threads, loop iterations, OpenMP threads or tasks).
- the time of placement decisions (e.g., dynamically during execution, at the beginning of program execution or before a parallel loop).
- support for dynamic adjustments of an initial placement.

Although some of the approaches have been presented as solutions for data placement only, they still rely on thread pinning, representing a minimal form of scheduling. Thread pinning is a common technique to virtualize processing units by preventing the operating system from migrating threads between cores, which guarantees a static mapping of threads to cores for the entire execution time. Hence, these approaches do not use scheduling as a mean to place computation close to data, but as a mean for deterministic mappings of threads to cores.

The approaches that use scheduling for improved data locality either rely on co-scheduling, placing threads or tasks that share data on shared caches or closely in the memory hierarchy (MINAS with profiling and the approach for unstructured parallelism), the distribution of loop iterations to cores (SCHEDULE REUSE and node arrangements) or placement of threads and tasks depending on the data that is being accessed (FORESTGOMP and LAWS).

The choice of where a scheduling entity is executed is either based on data sharing to minimize communication between the entities or based on the distribution of data, limiting the delay of accesses from cores to main memory. The type of scheduling entities depends on the supported framework for parallelism and the granularity and can be POSIX threads, individual loop iterations, OpenMP threads or fine-grained tasks, such as Cilk tasks or those of the custom framework for unstructured parallelism.

Most approaches establish an initial placement at the beginning of the execution and only few of them react to dynamic changes at execution time, e.g., load imbalance, by adjusting the initial placement during execution.

Conclusion

A closer look on the characteristics above reveals that not all of them are independent. A key role is taken by the implementation layer, which has an influence on both data placement and scheduling. The layer defines not only which information is available for placement decisions, but also limits when decisions are taken, which granularity for data placement and which entities for scheduling are available. This heavily influences the accuracy for capturing program behavior as well as the accuracy of the prediction of future behavior. The farther away the layer is from the application, the more it abstracts from the data structures and instructions and the less accurate the method becomes. For example, compiler-assisted approaches can rely on detailed information on data structures and instructions, e.g., individual elements of arrays and individual iterations of a loop, while implementations at the operating system layer only have access to coarse information, such as accesses to entire pages and threads. At the same time, low-level approaches are more generic and support a wider variety of programming languages and frameworks for parallelism. The choice of the implementation layer thus limits the range of solutions that is available for data placement and scheduling but also determines to which applications an approach applies.

2.5 Summary and problem statement

As motivated in Section 2.1, task-parallel programming is an increasingly popular approach to address the expectations on scalability, performance portability and productivity for applications intended to run on many-core systems. The performance of the execution of a task-parallel program strongly depends on an optimized run-time system that is able to exploit the underlying hardware efficiently. Section 2.2 showed that optimization for the memory hierarchy, i.e., for caches and non-uniform memory access, is a key factor in this context. The issues related to the efficient exploitation of the memory hierarchy can be addressed through optimized placement of tasks on cores and optimized placement of data on memory controllers as pointed out in Section 2.3. Ideally, this placement is fully automatic and thus does not require any intervention by the programmer to ensure high productivity. Furthermore, the solution should be able to adapt to an increasing number of cores and memory controllers and be able to react to dynamic changes of application behavior at execution time for load balancing to allow applications to scale on large machines. Performance portability can only be ensured if the run-time is able to adapt to a wide variety of machines and applications.

From the approaches discussed in the previous section, only LAWS and the approach for irregular parallelism address task-parallel languages or task-parallel frameworks. The other approaches apply to POSIX threads, OpenMP threads and loop iterations. The difference between POSIX threads and tasks is that POSIX threads are intended to run persistently and independently for a longer period of time with occasional synchronization, while tasks are short-lived entities that are synchronized by a run-time. OpenMP threads are closer to the concept of tasks, but originate from regularly structured loops [2]. Loop iterations represent an extreme case regarding the granularity, as an iteration can be composed of only a few instructions. However, to be efficient, loop scheduling to different cores must aggregate loop iterations. In addition, these solutions only apply to regularly structured applications based on loops. Finally, the approaches for task-parallel frameworks rely on specific properties of the program whose execution is to be optimized: the approach for unstructured parallelism relies on the property that tasks can be executed in any order while LAWS addresses divide-and-conquer-style computations. Hence, none of the approaches discussed in the previous section meets the needs with respect to the program structure and the granularity for scheduling and data placement for task-parallel programs.

As far as transparency is concerned, the approaches can be divided into three groups. The first group is composed of approaches that can be seen as technical solutions that help establish a specific distribution. Both information on relevant data as well as the actual distribution for approaches of this group must be provided by the programmer (e.g., AFFINITY-ON-NEXT-TOUCH, SCHEDULER

2. OpenMP 4 [25] supports fine-grained, dependent tasks, but the approaches above apply to earlier versions of OpenMP that support only parallel loops and independent tasks.

REUSE or node arrangements). The second group consists of approaches that automate placement decisions, but which rely on information provided by the programmer (e.g., FORESTGOMP). Approaches that belong to the third group gather information on relevant data automatically and take placement decisions autonomously (e.g., CARREFOUR, MINAS or LAWS). However, the fully automatic solutions of the last group are either restricted to specific kinds of data structures (e.g., MINAS for static arrays), to specific kinds of computations (e.g., LAWS to divide-and-conquer algorithms) or cannot react to dynamic changes at execution time due to the use of off-line profiling (feedback-directed page placement).

None of the solutions is fully transparent to the program, supports irregular applications and is able to react to dynamic changes at the same time.

Hence, efficient placement of data and tasks for many-core NUMA systems motivates revisiting previous run-time system designs and inventing new optimizations for the memory hierarchy fitting the needs of task-parallel applications. One of the crucial points for fully automatic data and task placement is the transparent collection of information on affinities between tasks and data. However, recent task-parallel programming models afford new opportunities to the run-time system to obtain detailed information on data that is accessed by a task as well as inter-task dependences. Thus, mechanisms for data and task placement operating at execution time can base placement decisions on this information. For example, point-to-point synchronization between individual tasks expressed in the programming model and preserved by the compiler indicates which tasks become ready in the future and which events lead to their activation. Models such as OPENMP 4 [25], STARSS [70] or OPENSTREAM [72] allow inter-task dependences to be specified as data dependences and are thus able to provide the run-time system with accurate information on accesses to data by each task and data sharing among tasks. Our approaches for task and data placement differ from existing work in the exploitation of this information for fully automatic, portable on-line placement of tasks and data by the run-time that improves the locality of memory accesses and thus performance significantly.

The solutions presented in this thesis combine point-to-point data dependences with detailed knowledge on target architectures. This requires a profound understanding of all software layers from the application over compilation to the run-time and the operating system. Application behavior has to be taken into account as it defines to which requests and situations the run-time system must be able to respond. Compilation has to be considered to decide which static information and how it must be preserved for placement decisions by the run-time at execution time. The run-time system represents the most important part of the solutions as it is responsible for all placement decisions. Finally, the run-time must be NUMA-aware and must interact with the operating system and the hardware efficiently in order to carry out requests of the application with low overhead and to avoid being a bottleneck for performance.

The theoretical part of our work involves the design of data placement and task placement techniques and elaborates the key aspects of application behavior and of the interaction with the hardware and operating system that have an influence on performance in general and the locality of memory accesses in particular. The practical side of our work consists in the implementation and integration of these techniques into a state-of-the-art task-parallel run-time system, their validation with a set of high performance, scientific applications on different hardware architectures and the development of adequate methods for performance analysis.

Most of the concepts presented in this thesis apply to task-parallel programming in general and can be adopted in the run-time systems of different task-parallel languages. However, the language extension for task-parallel applications that we have chosen for our optimizations is OpenStream. Like other modern task-parallel approaches, such as STARSS and OPENMP 4, dependences in OpenStream are expressed as point-to-point data dependences between tasks. However, the use of streams in OpenStream as first-class objects for synchronization offers a high degree of flexibility and enables advanced patterns for parallel programming, such as dynamic pipelining. Our choice for OPENSTREAM over other approaches has also been motivated by the fact that a widely accepted standard such as OPENMP 4 had not been published at the beginning of this thesis as well as by a close collaboration with Antoniu Pop, the author of OpenStream, and Albert Cohen.

The next chapter provides an introduction to OpenStream. We present the basic concepts of OpenStream, the syntax of OpenStream programs and its execution model.

3 OpenStream

This chapter provides an overview of OpenStream, a data-flow extension to OpenMP 3.0, which we have chosen as a state-of-the-art, task-parallel language for the experimental evaluation of the concepts presented in this thesis. We first introduce the basic concepts of OpenStream, namely data-flow streams, dependent data-flow tasks and synchronization of tasks based on streams. Producer-consumer relationships of an OpenStream program are captured by a dynamic task graph, for which we give a lightweight formal definition. The syntax of OpenStream programs is discussed in Section 3.2, followed by a series of examples illustrating this syntax in Section 3.3. The execution model of OpenStream is presented in Section 3.4. The chapter closes with an outline of the compilation of an OpenStream program in Section 3.5. All aspects of OpenStream presented in this chapter refer to the original implementation before any modification for the concepts proposed in this thesis.

3.1 Basic concepts

OpenStream [72] is a language extension [1] to OpenMP 3.0, which supports the specification of fine-grained task parallelism, data parallelism, and pipelining in the C programming language. The three central concepts of the extension are the *control program*, *data-flow tasks* and *streams*, presented below. The semantics of OpenStream are underpinned by the *Control-Driven Data Flow* (*CDDF*, [73]) formal computation model. However, in the following definitions we do not use the fully-fledged CDDF model, but a simplified, partial model that focuses on the aspects relevant for the discussion of the execution model.

Streams Streams are infinite sequences of elements of the same type that act as unbounded *FIFO queues* for communication between tasks. Each element of a stream has a unique integer index and is written using *dynamic single assignment*, i.e, each stream element is written at most once. Elements that have not been written are undefined and remain inaccessible for read accesses. Conceptually, a stream has a *read position* and a *write position* that define which elements of the stream are affected by subsequent read and write accesses.

Control program The control program instantiates tasks and specifies inter-task dependences, which are expressed as read and write accesses to stream elements. As an element of a stream cannot be read before it has been written, stream accesses effectively determine the (partial)

1. OpenStream has been proposed by Antoniu Pop and was developed in the context of his thesis [71]. The work presented in this thesis related to OpenStream is the result of a close collaboration with Antoniu Pop and Albert Cohen.

execution order of tasks. In order to guarantee deterministic behavior, OpenStream requires that the control program is sequential. However, under certain conditions the control program can be parallelized, as shown in Section 5.5. In the sequential case, the control program is executed by the root task of the OpenStream application, which corresponds to the main function of the program.

Data-flow tasks Tasks in OpenStream are short-lived, dynamic instances, defined by a *work-function* and an a set of *views*. The work-function is generated from the *body* of the task and contains the instructions to be executed when the task is scheduled. A view is a *sliding window* that allows the task to access a set of consecutive elements from a single stream or from several streams at once and is characterized by three attributes: the access type (read or write), the size of the window called *horizon* and the *burst*, which corresponds to the number of elements the read or write position of the stream is advanced after having determined the exact set of elements that the window provides access to. The CDDF computation model allows the burst to be smaller than the horizon for read accesses and restricts the burst to be equal to the horizon for write accesses. Likewise, the current implementation of OpenStream allows the burst to be smaller than the horizon only for read accesses, with the additional restriction that the burst must be zero. The reason for this restriction is explained in Section 3.4.5.

Access to stream elements is only possible indirectly through views, there is no mechanism to address specific elements of the stream directly. In the following parts, we illustrate how the actual stream indexes of the elements of views are determined.

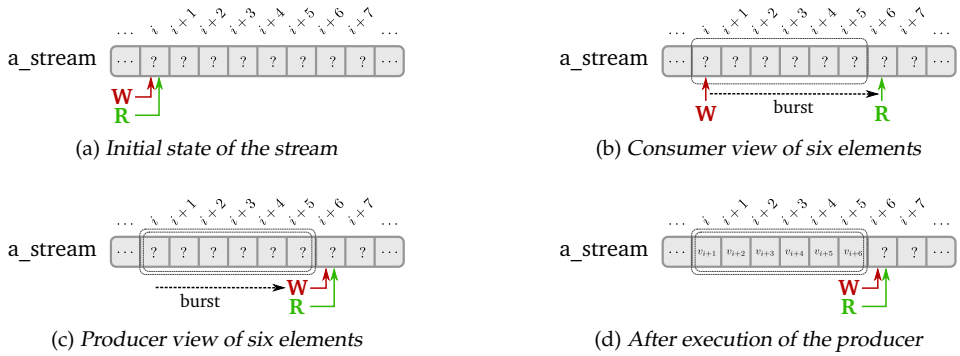
3.1.1 Stream accesses using views

Figure 3.1 illustrates accesses on a single stream with a reading and a writing view of two different tasks. The initial state of a stream named `a_stream` before any access of producers and consumers is shown in Figure 3.1a. All stream elements are undefined and no sliding window provides access to them. The read position and the write position, indicating at which indexes the next sliding windows of reading and writing views will start, are represented by R and W, initially pointing to the same position i. Figure 3.1b shows what happens when the sliding window of the reading view with a horizon and a burst of six elements is added. The base of the sliding window is determined according to the current read position, enabling access to elements $i, i+1, i+2, i+3, i+4$, and $i+5$ in read mode as indicated by the dotted rectangle. The read position is shifted by the burst and now points to element $i+6$, causing the first element of the window of a subsequent read view to start at index $i+6$. The write position is not modified and remains at element i. At this point, the set of elements the consumer view provides access to is determined, but the consumer cannot execute as the value of the elements is still undefined. Figure 3.1c shows the effect on the stream state of a writing view with the same characteristics as the reading view treated before. Again, the first element of the sliding window is i and access to elements $i, i+1, i+2, i+3, i+4$, and $i+5$ is provided. The burst of six elements advances the write position of the stream, which becomes equal to the read position. The elements of the stream remain unchanged until the task with the writing view has finished execution. The state after termination of the writing view is shown in Figure 3.1d, where the elements at indexes $i, i+1, i+2, i+3, i+4$, and $i+5$ have received their respective values $v_i, v_{i+1}, v_{i+2}, v_{i+3}, v_{i+4}$, and v_{i+5}.

As shown in the examples above, the first element in the horizon of a view is always the element at the read or write position when the sliding window is defined. Hence, it is not possible to access elements at arbitrary positions without advancing the read or write position to the desired element. In fact, stream indexes are only a concept of the formal semantics behind OpenStream, but do not appear in the actual implementation at all.

3.1.2 Dynamic task graphs

Tasks with sliding windows on the same stream elements are able to communicate by changing the elements' values. Due to the principle of dynamic single assignment, each stream element can only be written once and communication is unidirectional from a single producer of an element to one or more consumers. These producer-consumer relationships are captured by a structure



(a) *Initial state of the stream*

(b) *Consumer view of six elements*

(c) *Producer view of six elements*

(d) *After execution of the producer*

Figure 3.1: *Illustration of stream accesses with burst and horizon*

called *dynamic task graph*. As OpenStream programs can create an arbitrary amount of streams and tasks dynamically at run-time and as the instructions necessary to create them can be fully embedded into the control-flow of the application, this graph can in general not be constructed statically. Streams, tasks and stream accesses are only known at execution time and to find out which tasks communicate. It is thus necessary to analyze their dynamic relationships obtained from an execution trace. We say that an output view and an input view have been *matched*, when it has been determined that they provide access to a common set of elements of the same stream. Similarly, we define that two tasks are matched if there is at least one pair of matched views with one view belonging to one task and the other view belonging to the other task. In the following section, we define a minimal formal model for the definition of the dynamic task graph based on matched views. The matching itself is neglected in this model and will be described informally in Section 3.4.3.

Definition of a dynamic task graph

Each matched view can be formalized as a quadruple $(u, s, i_s, i_e) \in \{R, W\} \times \mathcal{S} \times \mathbb{N} \times \mathbb{N}$ where u indicates whether the view provides read or write access, s indicates the stream being accessed from the set of streams $\mathcal{S}$, i_s is the index of the first element of the sliding window defined by the view and i_e is the index of the last element included in the window. The horizon thus corresponds to $i_e - i_s + 1$. The burst is not modeled, as it is only necessary to determine the stream indexes, which are already known in this definition. Let $\mathcal{V}$ denote the set of all possible views. Each view can be broken down into a set of accesses to individual stream elements $\text{vacc} : \mathcal{V} \to \mathcal{P}(\{R, W\} \times \mathcal{S} \times \mathbb{N})$ with:

$$\text{vacc}(u, s, i_s, i_e) = \{(u, s, i) | i_s \leq i \leq i_e\}$$

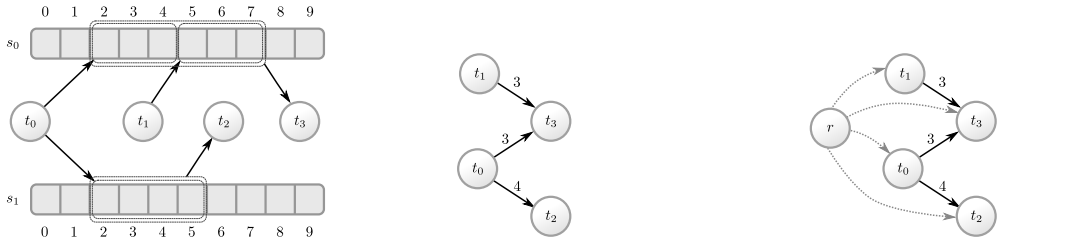
Let T_τ denote the set of dynamic task instances created from the beginning of the execution of the OpenStream program until a timestamp $\tau \in \mathbb{N}$ and let T_∞ be the (possibly infinite) set of tasks created during the whole execution of the program. We further define $\text{views}(t)$ of a task $t \in T_\infty$ as the set of views of t with $\text{views}(t) : T_\infty \to \mathcal{P}(\mathcal{V})$. The set of stream accesses $\text{sacc}(t)$ of a task t is the set of accesses to individual stream elements of all of its views:

$$\text{sacc}(t) = \bigcup_{v \in \text{views}(t)} \text{vacc}(v)$$

The set of read and write accesses, sacc_R and sacc_W, can be defined as:

$$\text{sacc}_R(t) = \{(R, s, i) \in \text{sacc}(t)\} \qquad \text{and} \qquad \text{sacc}_W(t) = \{(W, s, i) \in \text{sacc}(t)\}$$

We define the *dynamic task graph* $G = (T_\infty, E)$ as a graph whose vertexes represent dynamic task instances and its weighted, directed edges $E \subset T_\infty \times T_\infty \times \mathbb{N}$ indicate producer-consumer relationships between tasks. An edge (t_p, t_c, w) indicates that t_p writes w bytes of data to stream



(a) *Tasks and views* (b) *Resulting dynamic task graph* (c) *Extended dynamic task graph*

Figure 3.2: *Example of a dynamic task graph*

elements read by t_c. Let size : $\mathcal{S} \to \mathbb{N}$ be a function that specifies the size in bytes of elements for each stream. Based on the definitions above, E can be formalized as:

$$(t_p, t_c, w) \in E \Leftrightarrow C = \{(W, s_p, i_p, R, s_c, i_c) \in \text{sacc}_W(t_p) \times \text{sacc}_R(t_c) | s_p = s_c \wedge i_p = i_c\} \wedge$$
$$w = \sum_{\substack{(W, s_p, i_p,\\ R, s_c, i_c) \in C}} \text{size}(s_p) \wedge w \neq 0$$

Figure 3.2 shows an example of the formal definitions above. The tasks, streams and views are shown in Figure 3.2a and the resulting dynamic task graph is given in Figure 3.2b. Assuming an element size of one byte, the set of tasks, stream accesses of the tasks and the dynamic task graph are:

$$\begin{aligned}
T_\infty &= \{t_0, t_1, t_2, t_3\} \\
\text{views}(t_0) &= \{(W, s_0, 2, 4), (W, s_1, 2, 5)\} \\
\text{views}(t_1) &= \{(W, s_0, 5, 7)\} \\
\text{views}(t_2) &= \{(R, s_1, 2, 5)\} \\
\text{views}(t_3) &= \{(R, s_0, 2, 7)\} \\
\text{sacc}(t_0) &= \{(W, s_0, 2), (W, s_0, 3), (W, s_0, 4), (W, s_1, 2), (W, s_1, 3), (W, s_1, 4), (W, s_1, 5)\} \\
\text{sacc}(t_1) &= \{(W, s_0, 5), (W, s_0, 6), (W, s_0, 7)\} \\
\text{sacc}(t_2) &= \{(R, s_1, 2), (R, s_1, 3), (R, s_1, 4), (R, s_1, 5)\} \\
\text{sacc}(t_3) &= \{(R, s_0, 2), (R, s_0, 3), (R, s_0, 4), (R, s_0, 5), (R, s_0, 6), (R, s_0, 7)\} \\
E &= \{(t_0, t_2, 4), (t_0, t_3, 3), (t_1, t_3, 3)\}
\end{aligned}$$

Extended dynamic task graphs

The dynamic task graph as defined above contains information about the flow of data between tasks, but does not capture information about task creation. We define the *extended dynamic task graph* G^* of a dynamic task graph $G = (T_\infty, E)$ as $G^* = (T^*_\infty, E, E^*)$. The set of tasks T^*_∞ is the set of tasks created during execution of the program T_∞ extended with the root task r: $T^*_\infty = T_\infty \cup \{r\}$. The set of edges $E^* \subset T^*_\infty \times T^*_\infty$ defines the task creations. As the control program is executed by the root task, only r can create tasks, such that $E^* = \{r\} \times T_\infty$. The extended dynamic task graph of Figure 3.2b is shown in Figure 3.2c. The sets that were not present in the dynamic task graph are:

$$\begin{aligned}
T^*_\infty &= \{t_0, t_1, t_2, t_3, r\} \\
E^* &= \{(r, t_0), (r, t_1), (r, t_2), (r, t_3)\}
\end{aligned}$$

More examples of dynamic task graphs are provided in the following section, describing the syntax of OpenStream programs.

Terminology related to task graphs

Task graphs are a cornerstone of the analysis of OpenStream applications and will be used frequently throughout the remainder of this document. In order to be able to express certain properties of the task graph succinctly, we define the following terms: *light dependences*, *heavy dependences*, *balanced dependences* and *unbalanced dependences* and *dependence paths*.

The former two terms are defined relatively to the highest weight associated to an edge of a dynamic task graph. In many applications analyzed in the experimental evaluation the weights can be divided into a set of low weights and a set of high weights, where the high weights are orders of magnitude higher than the low weights. For example, the task graph of an application might be composed of edges with a weight of a few hundred bytes and other edges with weights of a few hundred KiB. Edges whose weights are associated with the set of lower weights are referred to as *light dependences*, while the other edges are referred to as *heavy dependences*.

A task that has both heavy and light input dependences is referred to as a task with *unbalanced input dependences*. If a task has only heavy or only light input dependences, we say that the task has *balanced input dependences*.

A *dependence path* in a dynamic task graph $G = (T_\infty, E)$ or an extended task graph $G^* = (T^*_\infty, E, E^*)$ is a path that is composed of edges from E. A path of heavy dependences only contains edges with weights from the set of high weights and a path of light dependences only contains edges with weights from the set of low weights. An application is said to have short dependence paths if the number of edges on the longest path between two tasks in the task graph is below a certain threshold, e.g., two edges. In contrast to this, an application with long dependence paths has a task graph whose shortest paths are above a certain threshold, e.g., ten edges.

3.2 The syntax of OpenStream programs

After the introduction of the basic concepts of OpenStream, we now explain the textual representation of an OpenStream program. Consistently with the syntax of OpenMP programs, OpenStream uses *pragmas* to embed OpenStream-specific statements into a program written in the C programming language. The OpenStream compiler translates these pragmas into appropriate data structures and code for interaction with the OpenStream run-time. All pragmas start with `#pragma omp`, followed by a more specific construct and, depending on the construct, a set of *clauses*. Currently, OpenStream uses three constructs:

- the `task` construct creates a new task; views for stream accesses can be specified using additional clauses as part of the construct.
- the `taskwait` construct creates a barrier that blocks execution until all the tasks of the current context have terminated.
- the `tick` construct advances the read position of a stream after the creation of tasks with views on the stream having a burst of zero elements.

All constructs can be embedded anywhere in the control-flow of the application, enabling the construction of dynamic task graphs as shown before. Dynamic creation of streams is not done using pragmas, but relies on a special *attribute* `stream` that, added to the definition of a variable, defines the variable as a stream. In the following part of this section, we present the elements necessary for the specification of an entire OpenStream application, starting with the declaration of streams. The description of the general syntax is followed by a set of examples illustrating its use.

3.2.1 Declaring streams and stream references

Streams in OpenStream can be created anywhere the C99 standard [55] allows the definition of a local variable. The syntax of a stream declaration is straightforward: as the state of a stream is entirely managed by the run-time system, all the programmer needs to specify is the type of the stream elements and an identifier, followed by the attribute `stream`, which lets the compiler distinguish a stream declaration from the declaration of an ordinary variable:

```
element_type stream_identifier __attribute__((stream));
```

OpenStream treats streams as first class objects and therefore also supports stream references. A stream reference can be declared as follows:

```
element_type stream_ref_identifier __attribute__((stream_ref));
```

As with any other data type, it is also possible to create arrays of streams and arrays of stream references, in which case the identifier is followed by an arbitrary expression for the size of the array in brackets:

```
element_type stream_arr_identifier[size_expr] __attribute__((stream));
element_type stream_ref_arr_name[size_expr] __attribute__((stream_ref));
```

The following example creates a stream of floating point elements and an array of 100 streams of characters, as well as a reference to the first stream from the array of streams:

```
/* Single stream of floating point elements */
float float_stream __attribute__((stream));

/* Array of 100 streams of characters */
char char_stream_arr[100] __attribute__((stream));

/* Reference to a stream of characters */
char char_stream_ref __attribute__((stream_ref));

/* Assignment of a stream reference */
char_stream_ref = char_stream_arr[0];
```

3.2.2 Declaring views

The definition of a view is syntactically split into two parts: a declaration specifying its type and horizon and a reference in the clauses of the task construct. The clause specifies which stream is to be accessed as well as the access type (read or write access). The syntax of a view declaration is identical to the declaration of a statically or dynamically sized local array in C99, where the horizon of the view corresponds to the expression that specifies the size of the array:

```
element_type view_identifier[size_expr];
```

For example, a view on a stream of floating point elements with a statically defined horizon of 5 elements and another view on a stream of integers with a dynamic horizon would be declared as follows:

```
float a_view[5];
int another_view[2*n+3];
```

If only a single element needs to be accessed, the horizon can be omitted and the declaration can be abbreviated to:

```
float a_view;
```

A special form of views are *multi-dimensional views*, which provide access to multiple streams at once, using the same horizon. The declaration of such a view is identical to the declaration of a multi-dimensional array:

```
element_type view_identifier[num_streams][size_expr];
```

If the expression specifying the number of streams is not constant, the view is called a *variadic view*. The following example declares a variadic view with a variable horizon on a stream of double precision floating point elements:

```
double view[num_streams][horizon];
```

3.2.3 Creating tasks

The connection between views and stream elements is realized by the task construct, enabling the dynamic creation of tasks. The views to be used by a task are specified by adding input, output, or peek clauses to the construct. Input and output clauses provide read and write access to stream elements, respectively. The peek clause is semantically equivalent to the input clause, but implies a burst of zero elements. Thus, a task using the peek clause has access to the elements of the stream according to the view's horizon, but does not advance the read position of the stream, allowing subsequent views to access the same elements. Task constructs without any clause create tasks that do not access any stream and which are therefore neither producer nor consumer. The full syntax of the task construct is:

```
#pragma omp task ⟨input(stream_expr >> view_expr, ...) |
                 peek(stream_expr >> view_expr, ...) |
                 output(stream_expr << view_expr, ...) |
                 sharing_clause |
                 ... ⟩
task_body
```

Sharing clauses allow the programmer to define how scalar variables declared outside the task are accessed within the task body (e.g., a private copy per task or shared use). The stream and view expressions define whether a single stream or multiple streams are referenced at once and whether the burst of the view should be identical to or different from the horizon of the view's declaration. A stream expression can be:

- the name of a stream or a stream reference (e.g., `a_stream`), in which case only a single stream is referenced and the view specified after the stream expression provides access to a set of consecutive elements of that stream
- an array expression composed of the name of an array of streams or stream references and an index expression in brackets (e.g., `a_stream_array[num_streams-x]`), also giving access to a set of consecutive elements from a single stream as in the first case
- the name of an array of streams or stream references, providing a two-dimensional window to the elements of a variable number of streams

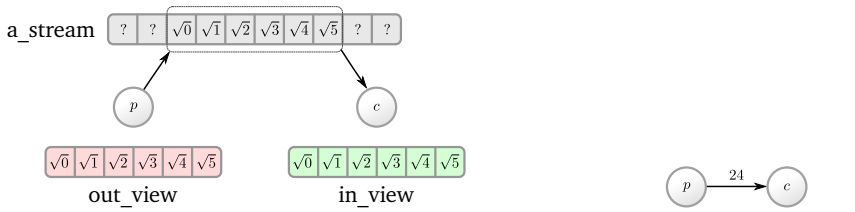
Depending on the stream expression, a view expression is either:

- the name of a view, which implies a burst of only one element
- a view with an explicit, constant or variable burst (e.g., `a_view[10]` or `a_view[2*n+3]`)
- a multi-dimensional or variadic view referencing several streams at once with an explicit burst for all of the streams (e.g., `a_view[num_streams][burst]`)

The task body can be either a single statement or a compound statement. The instructions forming the task body are automatically outlined by the compiler into a so-called *work-function* that is called when the task is executed. Besides access to stream elements through its views, a task body is also allowed to access scalar variables from the context surrounding the task construct using sharing clauses as defined by the OpenMP standard [24]. For input and output clauses, the burst specified in the access clause must be identical to the horizon of the view declaration due to restrictions from the execution model of OpenStream (cf. Section 3.4.5).

3.2.4 The tick construct

The tick construct modifies the read position of a stream without creating a task and is used for broadcasts of stream elements to multiple views. Broadcasts are implemented in two steps. First, the producer writing the elements to be broadcast as well as all the consumers are created in any order using the task construct with appropriate stream access clauses. The producer uses an ordinary output clause to obtain write access to the elements of the stream like any other producer task not involved in a broadcast. The consumers cannot use ordinary input clauses as these would automatically advance the stream's read position, such that consumers would not be able to access the same elements. Therefore, the consumers of a broadcast must use the peek clause, which



(a) *Stream accesses / sliding windows* (b) *Resulting task graph*

Figure 3.3: *Simple example with a single producer and a single consumer*

does not advance the read position of the stream and hence allows multiple consumers to obtain access to the same stream elements. For practical restrictions of the implementation, explained in Section 3.4.3, the burst of the producer's output view must match the horizon of the consumers peeking views. The second step of a broadcast consists of a tick operation that advances the stream's read position. At this point, all the set of consumers of the broadcast is determined and no further consumers can be added. The syntax of the tick construct is:

```
1  #pragma omp tick(stream_expr >> size_expr)
```

The stream expression must be either the identifier or an array expression addressing a single stream or a single stream reference. The expression for the size specifies by how many elements the read position is advanced and can be any expression of type `size_t` that matches the producer's burst.

3.2.5 Barriers

OpenStream offers built-in support for local barrier synchronization with the taskwait construct, which causes the task that encounters it to be suspended until all of the tasks of the current context have terminated. The syntax is conceivably simple:

```
1  #pragma omp taskwait
```

Barriers created with the taskwait construct are often employed at the end of the control program to make sure that all tasks have terminated before shared resources are freed.

3.3 Examples

To illustrate the principles and the syntax above, this section provides some basic examples with increasing complexity. The presentation starts with programs based only on input and output views and then shows how to implement broadcasts.

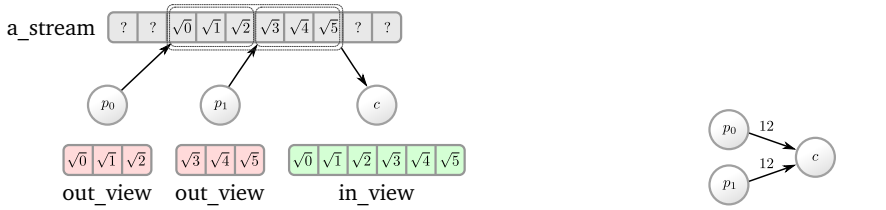
Tasks with ordinary output and input views

Figure 3.3a shows the stream accesses of a very simple program with a single output view and a single input view. A producer p writes the square roots of 0 to 5 to a stream named `a_stream`, read by a consumer task c. As c is the only consumer on `a_stream`, the elements can be discarded when c terminates, which makes a broadcast unnecessary.

To put into effect this behavior, both tasks need a sliding window of six elements to the same elements of `a_stream`, with p accessing the elements in write mode and c accessing them in read mode. Hence, the horizon of the input and output views must be six. As the elements are to be discarded afterwards, the burst of the input view is also six. The following listing shows the complete code of the example.

Listing 3.1: Single producer and single producer operating on a single stream

```
1  int main(int argc, char** argv)
2  {
```



(a) *Stream accesses / sliding windows* (b) *Resulting task graph*

Figure 3.4: *Two producers and a single consumer*

```
3    float a_stream __attribute__ ((stream));
4
5    int horizon = 6;
6    float out_view[horizon];
7    float in_view[horizon];
8
9    /* Producer p */
10   #pragma omp task output(a_stream << out_view[horizon])
11   {
12     for(int i = 0; i < horizon; i++)
13       out_view[i] = sqrtf((float)i);
14   }
15
16   /* Consumer c */
17   #pragma omp task input(a_stream >> in_view[horizon])
18   {
19     for(int i = 0; i < horizon; i++)
20       printf("Read %f\n", in_view[i]);
21   }
22
23   #pragma omp taskwait
24
25   return 0;
26 }
```

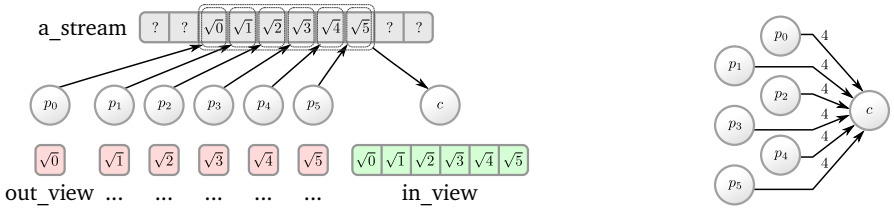
The code starts with the declaration of the stream of floating point elements in line 3 using the attribute `stream`. Lines 6 and 7 declare the views used by p and c, `out_view` and `in_view`, both with a horizon of six elements. The producer and consumer tasks are created in lines 10 and 17, respectively, using the `task` construct and appropriate clauses. Note that the task bodies reference the variable `horizon`, although it was declared in the surrounding scope and no sharing clause defines how it should be accessed. This is possible because scalar variables declared outside, but referenced inside a task are declared *firstprivate* by default, meaning that the compiler creates an individual copy of the variable for each task initialized with the value from the surrounding context at the time of the task creation. The taskwait construct in line 23 blocks the control program until p and c have finished. This prevents the application from ending prematurely before the producer and consumer have executed.

Nothing in the example specifies the direct producer-consumer relationship between p and c. The producer and the consumer just *happen* to operate on the same elements of the same stream and the producer-consumer relationship is the result of the matching of the input view and the output view on the stream. Figure 3.3b shows the dynamic task graph of this application. The exact mechanism matching producers and consumers will be explained in section 3.4 describing the execution model of OpenStream.

The next example adds some complexity to the previous one. Instead of a single producer that writes all the elements at once, two producing tasks p_0 and p_1 each produce three of the six elements, as illustrated in Figure 3.4a. To implement this behavior, the code from the previous listing needs only a few adaptations:

Listing 3.2: Two producers and a single consumer operating on a single stream

```
1  int main(int argc, char** argv)
2  {
```



(a) *Stream accesses / sliding windows*

(b) *Resulting task graph*

Figure 3.5: *Six producers and a single consumer operating on the same stream*

```
3   float a_stream __attribute__ ((stream));
4
5   int horizon_out = 3;
6   float out_view[horizon_out];
7
8   int horizon_in = 6;
9   float in_view[horizon_in];
10
11  /* Producer p0 */
12  #pragma omp task output(a_stream << out_view[horizon_out])
13  {
14    for(int i = 0; i < horizon_out; i++)
15      out_view[i] = sqrtf((float)i);
16  }
17
18  /* Producer p1 */
19  #pragma omp task output(a_stream << out_view[horizon_out])
20  {
21    for(int i = horizon_out; i < 2*horizon_out; i++)
22      out_view[i] = sqrtf((float)i);
23  }
24
25  /* Consumer c */
26  #pragma omp task input(a_stream >> in_view[horizon_in])
27  {
28    for(int i = 0; i < horizon_in; i++)
29      printf("Read %f\n", in_view[i]);
30  }
31
32  #pragma omp taskwait
33
34  return 0;
35 }
```

Due to the different horizons of the producers and the consumer, the previous view declarations have been replaced by declarations referencing different variables, `horizon_out` and `horizon_in` (lines 6 and 9). The two producer tasks are created in lines 12 and 19. Although they both use `out_view` in their output clauses, they do not access the same elements of the stream. In fact, the compiler uses the declaration of a view only to determine the element type and the horizon of a view. Within different task bodies, the same view can refer to completely different data locations. The code of the consumer task (lines 26–30) and the rest of the code are almost identical to the previous example. The dynamic task graph resulting from the execution is shown in Figure 3.4b.

As stated earlier, task creation can be fully embedded into the control flow of the control program. This concept is put into practice by the next two examples creating tasks dynamically within a for-loop. Assume that the production of stream elements needs to be parallelized further, such as illustrated in Figure 3.5a and Figure 3.5b, where each producer task writes only a single element of to the stream. To this end, the task construct creating a producer can simply be embedded into the body of a for loop:

Listing 3.3: Creation of producers in a for-loop

```
1  float out_view;
2
3  for(int i = 0; i < 6; i++) {
4    #pragma omp task output(a_stream << out_view)
```

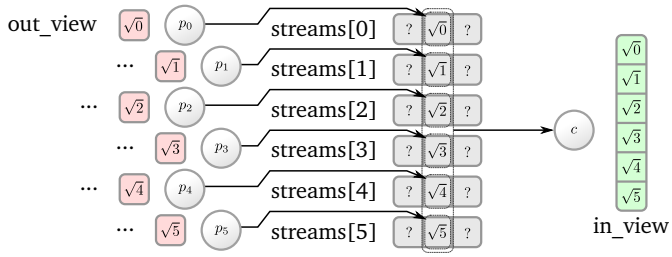


Figure 3.6: *Six producers and a single consumer operating on six streams of an array of streams*

```
5     {
6       out_view = sqrtf((float)i);
7     }
8   }
```

As only one element is written per task, the output view is declared as a scalar and the output clause uses the abbreviated syntax with an implicit burst of one element.

An alternative way to specify the behavior of the previous example is to use multiple streams, e.g., one stream per element and to let the consumer read from all these streams at once. This is shown in Figure 3.6: each producer p_i writes a single element to a single stream `streams[i]` from an array of streams named `streams`. The program can be written as:

Listing 3.4: Consumer using a variadic view

```
1   int main(int argc, char** argv)
2   {
3     float streams[6] __attribute__ ((stream));
4
5     float out_view;
6     float in_view[6][1];
7
8     for(int i = 0; i < 6; i++) {
9       /* Producers p0 ... p5 */
10      #pragma omp task output(streams[i] << out_view)
11      {
12        out_view = sqrtf((float)i);
13      }
14    }
15
16    /* Consumer c */
17    #pragma omp task input(streams >> in_view[6][1])
18    {
19      for(int i = 0; i < 6; i++)
20        printf("Read %f\n", in_view[i][0]);
21    }
22
23    #pragma omp taskwait
24
25    return 0;
26  }
```

Line 3 defines the array of streams with six elements. The output view of the producers in line 5 remains unchanged, but the input view, now uses a two-dimensional format. The outer dimension specifies the number of streams that will be used by the view and the inner dimension specifies horizon, which must be identical for all streams. Within the task body of the consumer, the input view can be referenced like any two-dimensional array of floating point elements (line 20).

Broadcasts

Broadcasts can be implemented easily using an output view and peeking views on a stream. In the example shown in Figure 3.7, the elements written by a single producer are read by three consumers c_0, c_1 and c_2. Listing 3.5 shows how the different consumers use their input data: c_0 (line 17) calculates the sum of all input elements, c_1 (line 28) calculates their product and c_2 (line 39) computes the sum of the squares. All consumers use the peek clause, which implies a burst of 0 elements. The tick construct advances the read position and effectively triggers the

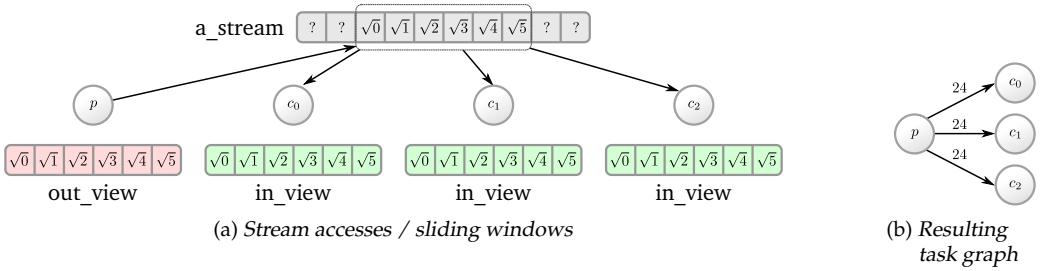
broadcast by activating the producer task, i.e., the producer task becomes ready for execution. Upon termination of the producer, the data is available and the consumers are ready for execution. A detailed description of the run-time mechanisms related to broadcasts at execution time is given in Section 3.4.3.

Listing 3.5: Multiple consumers reading the same elements

```
1  int main(int argc, char** argv)
2  {
3    float a_stream __attribute__ ((stream));
4
5    int horizon = 6;
6    float out_view[horizon];
7    float in_view[horizon];
8
9    /* Producer p */
10   #pragma omp task output(a_stream << out_view[horizon])
11   {
12     for(int i = 0; i < horizon; i++)
13       out_view[i] = sqrtf((float)i);
14   }
15
16   /* Consumer c0 */
17   #pragma omp task peek(a_stream >> in_view[horizon])
18   {
19     float accu = 0.0f;
20
21     for(int i = 0; i < horizon; i++)
22       accu += in_view[i];
23
24     printf("Sum: %f\n", accu);
25   }
26
27   /* Consumer c1 */
28   #pragma omp task peek(a_stream >> in_view[horizon])
29   {
30     float accu = 1.0f;
31
32     for(int i = 0; i < horizon; i++)
33       accu *= in_view[i];
34
35     printf("Product: %f\n", accu);
36   }
37
38   /* Consumer c2 */
39   #pragma omp task peek(a_stream >> in_view[horizon])
40   {
41     float accu = 1.0f;
42
43     for(int i = 0; i < horizon; i++)
44       accu += in_view[i] * in_view[i];
45
46     printf("Sum of squares: %f\n", accu);
47   }
48
49   #pragma omp tick (a_stream >> horizon)
50
51   #pragma omp taskwait
52
53   return 0;
54 }
```

3.4 Execution model

After the discussion of the central concepts and the syntax of OpenStream programs in the previous sections, we give an overview of the execution model. We first explain how tasks that are ready for execution are scheduled and executed on the different cores of the machine. We then discuss how the run-time manages the creation of tasks and how it detects that a task is ready. This discussion also includes the explanation of how views are associated to stream elements. Finally, we introduce the memory management layer of the run-time based on memory pooling.



(a) *Stream accesses / sliding windows*

(b) *Resulting task graph*

Figure 3.7: *Multiple consumers reading the same elements*

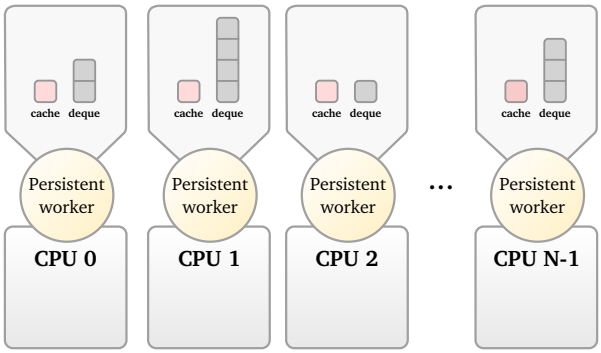


Figure 3.8: *Per-worker data structures and worker placement in OpenStream*

3.4.1 Scheduling and work-stealing

One of the central components of the run-time is the scheduler, which manages the execution of tasks that are ready. As the run-time is intended to run on massively parallel systems, the approach used for scheduling is distributed and uses lock-free implementations of the most critical data structures. This avoids high synchronization overhead and thus prevents the scheduler from becoming a bottleneck for performance. In this approach, each core involved in the execution of the application has a persistent worker thread, an ordinary POSIX thread running a scheduling loop, which executes ready tasks on the local core independently from the other processing units. All worker threads are created at the beginning of the execution of the application and remain alive until its termination. By default, one persistent worker is placed on each core as shown in Figure 3.8, but workers can be placed in any order on the cores of the machine as long as no more than one worker executes on every core. However, the mapping of workers to cores can only be set at the beginning and remains the same for the entire execution time.

Figure 3.8 also shows the data structures involved in the scheduling process. The first structure is a *work-stealing deque*, a double ending queue that can contain an arbitrary amount of tasks ready for execution. The second structure is a *single entry software cache* that can only contain up to one single ready task at a time. When a worker activates a task, it tries to add the task to the single entry software cache first. If the cache is empty, this operation immediately succeeds. However, if the cache already holds a task, this task is removed from the cache and added to the work-deque in order to leave the entry of the cache to the newly activated task. Hence, the cache always contains the latest task activated by the worker.

When the worker finishes execution of a task, it first checks if there is a task in the software cache and, if so, removes it from the cache and executes it. If the cache is empty, it tries to *pop* a task from bottom of the work-deque. If both the cache and the deque are empty, the worker chooses a random victim worker using a uniform distribution and tries to *steal* a task from the top of the victim's work-deque. Work-stealing is only allowed on the work-deque and the software cache

Algorithm 1: scheduler_loop(w)

```
1  w.cached_task ← null
2
3  while true do
4      t ← w.cached_task
5      w.cached_task ← null
6
7      if t = null then
8          t ← pop_bottom(w.work_deque)
9      end
10
11     while t = null do
12         victim ← random_worker()
           t ← pop_top(victim.work_deque)
13     end
14
15     execute_task(t)
16 end
```

Algorithm 2: add_task_locally(t, w)

```
1  if w.cached_task ≠ null then
2      push_bottom(w.work_deque,
3                  w.cached_task)
4  end
5
6  w.cached_task ← t
7
8
9
10
11
12
13
14
15
16
17
```

remains inaccessible to other workers than the owner.

The purpose of the software cache is thus twofold. First, as only the worker itself has access to the cache, adding or removing a task can be accomplished without any synchronization overhead. Second, a task in the software cache cannot be stolen by another worker, which avoids the following situation. Let w be a worker that is currently executing a task t and let t_r be the task that was last activated by w. Assume further, that the work-queue of w was empty before t_r was activated. If t_r is stolen by another worker, w runs out of work and has to steal another task after termination of t, with atomic operations on a remote work-deque.

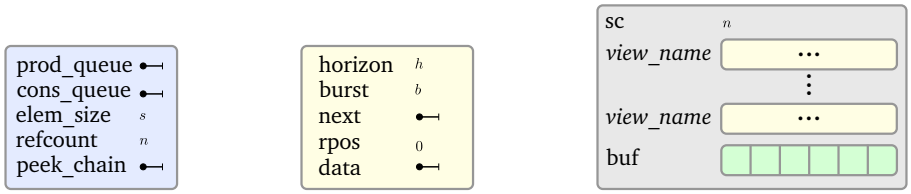
The work-deque is implemented using the lock-free deque proposed by Chase and Lev [37]. Tasks are put into the deque at the bottom end and can only be stolen by other workers at the top end. The only worker that is allowed to remove a task from the bottom is the owner of the deque. From a worker's perspective, tasks are executed in LIFO order, favoring local execution of tasks whose input elements have been written recently, resulting in better cache usage. In contrast to this, task stealing occurs in FIFO order, meaning that tasks whose data is less likely to be present in the cache hierarchy are executed remotely.

Algorithms 1 and 2 summarize the principles of scheduling presented above. Each worker enters *scheduler_loop* upon its creation, which contains an infinite scheduler loop, ensuring constant execution of tasks obtained from the local software cache, the local work-stealing deque or through work-stealing from another worker's work-deque. The function *add_task_locally* is called whenever a worker w causes a task to become ready for execution and adds that task to the software cache.

3.4.2 Data structures

Each entity of an OpenStream program (i.e., streams, tasks and views) is associated with a data structure in the run-time as shown in Figure 3.9. A stream is characterized by the attributes shown in Figure 3.9a, which are:

- A list of unmatched output views on the stream (`prod_queue`)
- A list of unmatched or partially matched input views on the stream (`cons_queue`)
- The size in bytes of its elements (`elem_size`)
- A reference counter for garbage collection (`refcount`)
- A list of unmatched peeking views on the stream (`peek_chain`)



(a) *Stream* (b) *View* (c) *Frame*

Figure 3.9: *Major data structures of the OpenStream run-time*

The list of producers, consumers and peeking consumers are initially empty when the stream is created and the element size is initialized according to the size of elements specified in the declaration of the stream. The initial value of the reference counter is one and is increased by one at the creation of every stream reference referring to the stream. Views are represented by the data structure illustrated in Figure 3.9b with the following fields:

- the horizon expressed in bytes (`horizon`)
- the burst expressed in bytes (`burst`)
- a field used for chaining of the view in a linked list (`next`)
- the reached position used for indexing of the data buffer and to check if the view is unmatched, partially matched or fully matched (`rpos`)
- a pointer to the elements of the sliding window (`data`)

Note that there is no field indicating whether the view provides read or write access. This information is kept by the compiler and passed to the run-time as a parameter upon a call to the function that matches a view with stream elements. Horizon and burst are initialized according to the horizon and burst of the view. The data structure of a view from a peek clause receives a burst of zero, allowing the run-time code to recognize it as such. For all other views, burst and horizon are identical. Upon creation, the data location of the view is unknown and hence initialized to `NULL`. The reached position is set to 0, indicating that the view is unmatched, i.e., not associated to any producer or consumer.

The last data structure presented in Figure 3.9 is the *data-flow frame* or *frame* for short and represents a task. A frame is composed of:

- A synchronization counter `sc`, indicating if the task is ready for execution (`sc` = 0) or if it has unmet dependences (`sc` $>$ 0)
- A set of views, each identified by its respective name from the declaration of the view
- A data region `buf` which has enough space to store all elements of its input views

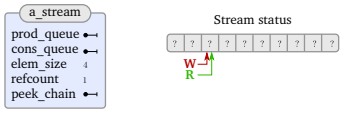
The synchronization counter of a task is initialized with a value representing the sum of the horizons of its views, each multiplied with the size of the views' element types.

3.4.3 Dependence management

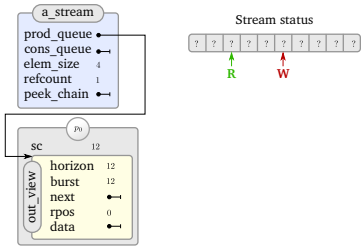
As shown in the introduction and the examples of this chapter, producers and consumer are matched dynamically, only based on their stream accesses. In this section, we show how this matching is implemented by the run-time library based on the data structures presented previously. We start with the matching of ordinary input and output views and present the same procedure for broadcasts using peeking views afterwards.

Ordinary input and output views

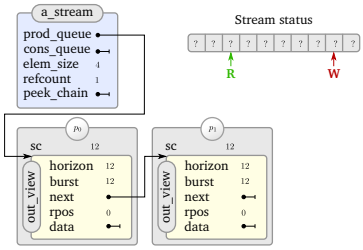
When a task is created, its frame is allocated and initialized, including all the data structures for the task's views. Once the initialization is finished, output views are matched with consumers and input views are matched with producers by invoking a procedure called `resolve_dependences` for each view.



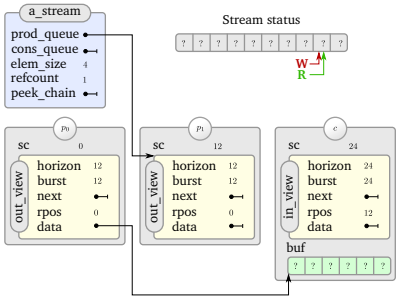
(a) *Initial state after creation of the stream*



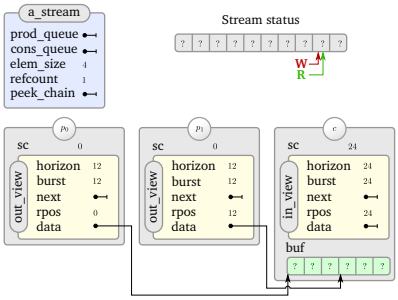
(b) *Creation of the first producer*



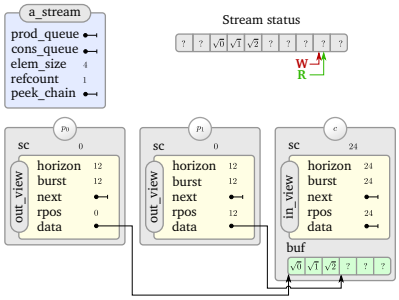
(c) *Creation of the second producer*



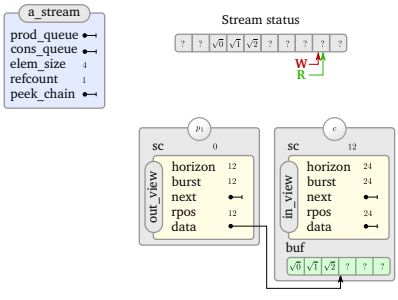
(d) *Creation of the consumer, matching with the first producer*



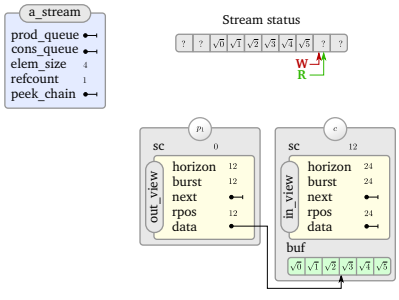
(e) *Matching between the second producer and the consumer*



(f) *Execution of the first producer*

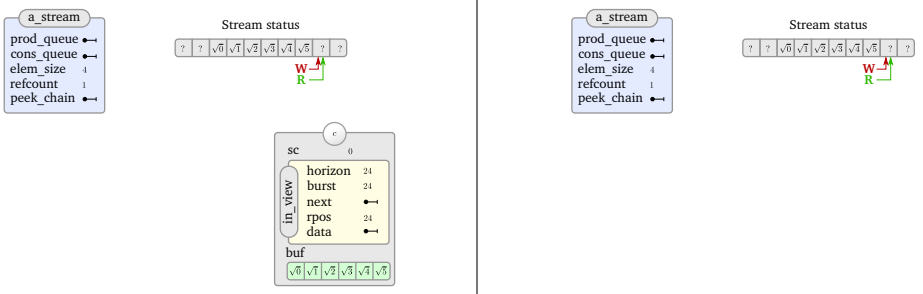


(g) *Update of the consumer's synchronization counter*



(h) *Execution of the second producer*

Figure 3.10: *Resolution of the dependences of the tasks from listing 3.2 on page 39*



(i) *Update of the consumer's synchronization counter*

(j) *After execution of the consumer*

Figure 3.10: *Resolution of the dependences of the tasks from listing 3.2 on page 39 (continued)*

To illustrate how the different data structures are used during dependence management by `resolve_dependences`, reconsider the example of listing 3.2 on page 39, in which a single consumer reads a total of six floating point elements of four bytes produced by two tasks, each writing three of the six elements. The entire process from the creation of the stream and the tasks to the execution of the tasks is illustrated in Figure 3.10.

Figure 3.10a shows the state of the run-time after the creation of the stream `a_stream`. As there are neither producers nor consumers yet, the corresponding chains of unmatched views, `prod_queue` and `cons_queue`, are empty. The stream status at the upper right of the figure does not represent an actual data structure of the run-time and only serves to illustrate the current read and write positions as well as the contents of the stream.

Figure 3.10b shows what happens when p_0 is created. As there is no consumer reading from the stream, the output view cannot be matched with an input view yet and the data structure representing the output view of the task is added to the stream's queue of unmatched producers, `prod_queue`. However, even though the matching is not complete, conceptually, the write position of the stream is advanced by the burst of the view, such that subsequent producers write the elements at positions following the elements written by p_0. Note that the values for horizon and burst in the figure are specified using bytes and not the number of elements. The pointer `next`, used for chaining of unmatched views, is initialized to `NULL` as the output view of p_0 is currently the only unmatched view writing to the stream and thus does not have a successor in the list. The field `rpos` is unused for output views in this example and can be ignored. As the data location is currently unknown due to the incomplete matching, the field `data` it is initialized to `NULL`. The synchronization counter of the task, `sc`, keeps its initial value of 12 (one unmatched output dependence with three floating point elements of four bytes).

The next step in the example program is the creation of the second producer, p_1, as shown in Figure 3.10c. Exactly as was the case at creation of the first producer, no consumer reading from the stream has been created so far. Hence, the output view of p_1 is added to the queue of unmatched producers using the field `next` of the output view of p_0. Again, the write position of the stream is advanced by the burst of the view of three elements. The remaining fields are initialized with the same values as for the previous task.

The consumer task c is finally created in the step illustrated by Figure 3.10d. Burst and horizon are both set to 24 bytes corresponding to the six elements specified in the declaration of the input view of c. The buffer for the input elements is embedded into the frame, indicated by the array of 6 elements below `buf`. The current write position within this array is the first element, thus, `rpos` is initialized to 0 (this is not shown in the figure, since it represents the state after matching with p_0). At execution of `resolve_dependences` for the input view of c, the list of unmatched producers is consulted. If this list would be empty, the input view would have been added to `cons_queue` as seen for the producers and `prod_queue` before. However, as the list is not empty, the first output view is removed and matched with the input view of c. This is done in several

steps. First, the data pointer of the output view is set to the current write position of the input view, which is calculated by indexing `buf` using the current value of `rpos` of the input view. The result of the indexing operation is the base address of `buf`, i.e., `&buf[0]`. The value of `rpos` is updated according to the horizon of the output view of p_0, i.e., a value of 12, as shown in the figure. Next, the synchronization counter of the producer task is updated by subtracting the burst of the output view. The new value of 0 indicates that the task is ready for execution. The matching with p_0 is now complete. However, the reached position of the input view of c still has not reached the horizon, such that `resolve_dependences` continues matching with the second output view.

The result of this process is shown in Figure 3.10e. Similar to the previous matching, the output view of p_1 is removed from the list of unmatched producers, the data pointer of the output view of p_1 set to the current write position of the input buffer and the synchronization counter of the output view is updated accordingly. The field `rpos` of the output view receives the value of the input view before the second matching, which was 12. However, in this example, this field can be ignored for output views and is only shown in order provide a coherent description of the matching algorithm. The reached position of the input view now matches its horizon, which indicates that matching of this view is complete. Note that the synchronization counter of the consumer is not updated yet, due to the fact that its input data only becomes available when the producing tasks terminate.

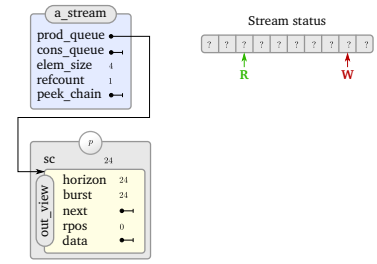
Assume that p_0 is executed first and finishes its execution. Figure 3.10f shows the state of the different data structures right at termination: the input buffer of c now contains the data written by p_0. After termination of the work function of p_0, the synchronization counter of c is reduced by the burst of the output view, resulting in an updated value of 12 (cf. Figure 3.10g). When p_1 executes, the last three elements become available and the synchronization counter of c finally reaches zero, which activates c (Figures 3.10h and 3.10i). The task eventually executes and its resources are freed, as shown in Figure 3.10j.

Broadcasts

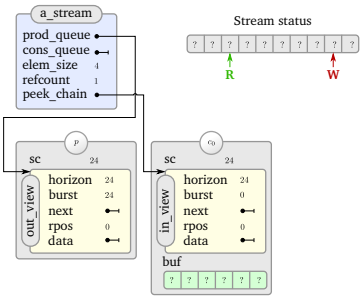
For the illustration of broadcasts, we show what happens during execution of the code using peek clauses and the tick construct presented in listing 3.5 on page 42. At its creation the producer task is entirely unaware of the broadcast and is treated like any ordinary producer whose consumers have not been created, yet. This is shown in Figure 3.11a: the output view is simply added to the list of unmatched producers just as in Figure 3.10b of the previous example. The consumers, however, are treated differently than before. When `resolve_dependences` is called with an input view with a burst of 0, i.e., a peeking view, it does not match producers and consumer directly, but defers this action until execution reaches the tick clause. Peeking views on the same elements, i.e., peeking views that are matched before the read position of the stream is advanced with a `tick` clause, are queued using the field `peek_chain` of the stream, as shown in Figures 3.11b to 3.11d. Neither the producer, nor the consumers are ready for execution during this period, as indicated by their synchronization counters keeping their initial values of 24. When the run-time encounters the tick clause, it resets the chain of peeking views and resolves the dependences of the first consumer view (Figure 3.11e). The producer is removed from the chain of unmatched output views and its data pointer is set to the first element of the buffer of the input view. The synchronization counter of the producer reaches zero and the producer is ready to execute. At termination of the producer in Figure 3.11f, all input elements of the first consumer have been written, but the task remains blocked until the data has been copied to the remaining consumers (Figure 3.11g). Upon completion of this operation, the consumers' synchronization counters are updated and the producer's data structures can be freed (Figure 3.11h).

3.4.4 Allocation of data structures

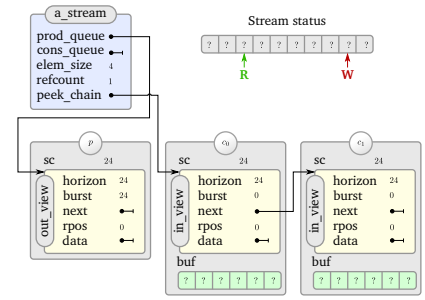
There are multiple data structures involved in dependence management and scheduling of tasks, e.g., streams and data-flow frames, presented above. Many of them need to be allocated and freed dynamically throughout the execution of a stream application. Often they are used only from creation of a task until its termination and can therefore have a very short lifetime, resulting



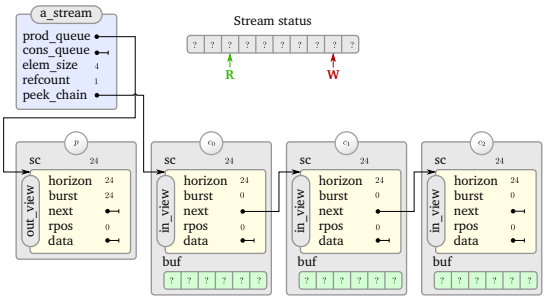
(a) *Creation of the producer task*



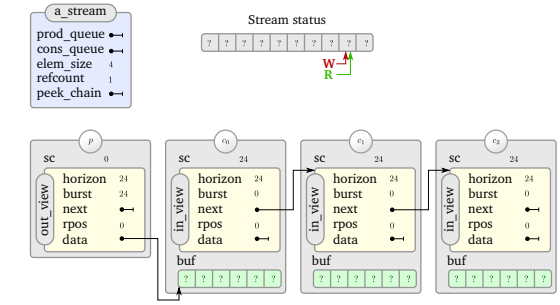
(b) *Creation of the first consumer*



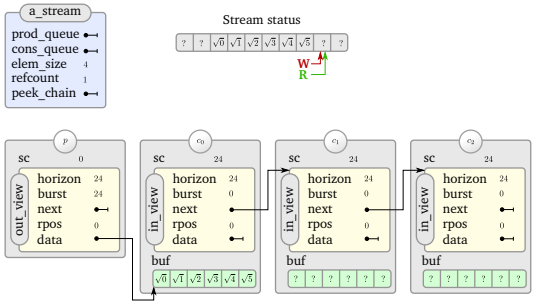
(c) *Creation of the second consumer*



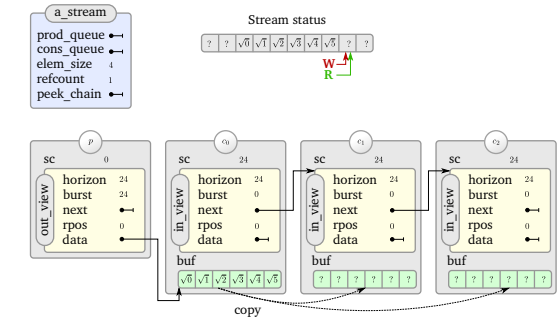
(d) *Creation of the third consumer*



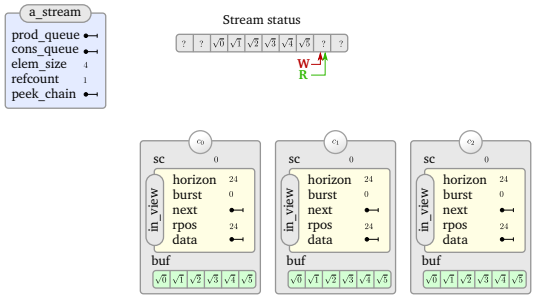
(e) *Matching of the producer and the first consumer at the tick operation*



(f) *Termination of the producer*



(g) *Broadcast of the data to the remaining consumers*



(h) *Activation of the consumer tasks*

Figure 3.11: *Dependence resolution of broadcasts*

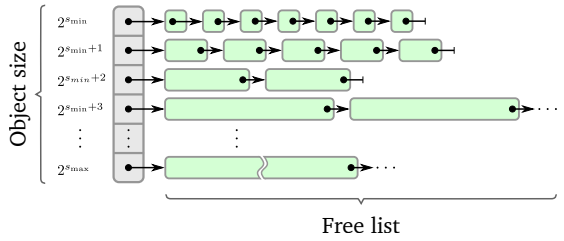


Figure 3.12: *Illustration of the principles of a per-worker memory pool*

in frequent invocations of functions allocating and freeing memory resources. In addition, on many-core systems with a high number of workers executing in parallel, these functions might be called with a high degree of concurrency. To prevent memory management from becoming a bottleneck, the run-time system must thus rely on an optimized memory allocator.

Memory pooling

Due to the parallelism within the run-time itself, resulting from the concurrent activity of workers, a centralized memory allocator satisfying all requests would require a substantial effort on synchronization of concurrent calls. Instead, the OpenStream run-time uses a decentralized approach based on per-worker *memory pools*.

The principles of memory pooling are straightforward. The size of each data structure used by the run-time system is assumed to be between $2^{s_{\min}}$ and $2^{s_{\max}}$ bytes. For each power of 2^i with $i \in \{s_{\min}, s_{\min}+1, s_{\min}+2, \ldots, s_{\max}\}$, a linked list of free blocks of size 2^i bytes is maintained, as illustrated in Figure 3.12. When an allocation of size m takes place, the allocator first checks if $m > 2^{s_{\max}}$ holds. If this is the case, the size of the request is too big to be handled by the memory pool and the request is redirected to the standard C memory allocator (e.g., `malloc`). For $m \leq 2^{s_{\max}}$ the allocator checks whether there is a free block in the list of blocks whose size corresponds to the next greatest power of two at least of size $2^{s_{\min}}$, i.e., 2^j with $2^j \geq m \wedge \nexists j' : s_{\min} \leq j' < j$. If such a block exists, the allocator removes it from the list of free blocks and returns it as the result for the request. If no such block is available, the allocator performs a *refill* operation that allocates a contiguous chunk of memory of size $M = k \cdot s^j$, splits it into k equal-sized blocks, adds the first $k-1$ blocks the free list and returns the last block as the result of the request. Freeing a block works similarly: the allocator determines the corresponding free list and adds the block at its head. If the size of the block to free exceeds the maximum size handled by the memory pool, it forwards the request to the standard memory allocator (e.g., `free`).

The main advantage of using memory pools for memory management is that almost all requests can be carried out in constant time. The only exception are refill operations, which become less frequent once the maximum number of blocks used simultaneously is reached. Additionally, per-worker memory pools guarantee that the free lists are completely private and do not need to be protected for concurrent accesses, e.g., using locks or atomic operations. Therefore, they do not induce any synchronization overhead.

Life cycle of objects from a memory pool

The life cycle of blocks that are handled by a memory pool, i.e., objects whose size does not exceed $2^{s_{\max}}$, resulting from the allocation scheme above has five distinct stages:

1. Allocation from the operating system due to a refill operation
2. Allocation from the free list of a memory pool
3. Exclusive use of the block by the run-time or use by the run-time and the application
4. De-allocation of the block by putting it back to a free list
5. Return of the memory of the block to the operating system

Due to the reuse of blocks when using memory pooling, stages 3 and 4 can occur an arbitrary number of times. An important aspect of these allocations and de-allocations is that they do not necessarily have to involve the same memory pools. For example, a data-flow frame is allocated

(a) *Creation of the first consumer* (b) *Creation of the second consumer*

Figure 3.13: *Invalid program with bursts smaller than the horizons*

(a) *Creation of the first consumer* (b) *Creation of the second consumer*

Figure 3.14: *Invalid program with multiple consumers reading from the same producer*

in the memory pool of the worker executing the control program, but the associated task can be executed by any other worker. The frame could thus be freed to another memory pool than the pool from which is was allocated.

3.4.5 Restrictions from the execution model

The presentation of the data structures and the algorithm for resolving dependences between tasks showed that stream data is not stored in data structures directly associated to a stream, but in input buffers located in the data-flow frames of tasks. Each structure representing a view has only a single field, `data`, pointing to the first of the elements accessible through the view. An advantage of this representation is that consecutive elements of a stream are stored at consecutive addresses and can thus be accessed by simple indexation. However, in order to guarantee this data layout at execution time, valid OpenStream programs are subject to a few restrictions.

Restriction 3.1 (Bursts and horizons of a view) *Burst and horizon of a reading view must either be identical or the burst must be equal to zero.*

This avoids that a *subset* of the elements of an output view is copied to multiple input views. Figure 3.13 shows an example of an invalid OpenStream program with an output view of six elements and two input views having a horizon of three elements and a burst of two elements. The result in Figure 3.13b shows that there is an element at index $i + 2$, which is accessed both by c_0 and c_1 and which would have to be copied to the first position of the input view of c_0 and the third position of the input view of c_1.

Restriction 3.2 (Horizons of output and input views of producers and consumers) *The elements accessible through a view cannot be scattered across multiple input buffers, i.e., there cannot be any output view whose elements are not entirely read by consumers:*

$$\begin{aligned}&\forall t \in T_\infty : \forall (W, s, i_s, i_e) \in \mathit{views}(t) : \\ &\forall t' \in T_\infty : \forall (R, s, i'_s, i'_e) \in \mathit{views}(t') : \\ &([i_s, i_e] \cap [i'_s, i'_e] = \emptyset \vee [i_s, i_e] \cap [i'_s, i'_e] = [i_s, i_e])\end{aligned}$$

Note that this restriction cannot be verified by the compiler due to the dynamic matching and is therefore checked at execution time. Figure 3.14 shows an example of an invalid program with

two input views accessing elements from a single output view. As in Figure 3.13, there is one producer writing 6 elements, but the consumers now have a burst that matches the horizon of 3 elements. However, as the data would have to be distributed onto the input buffers of c_0 and c_1, this program is invalid.

Restriction 3.3 (All elements of a stream that are written must also be read) *This restriction forbids to write any stream element that is never read. As the writer relies on the input buffer of at least one consumer to stores its produced elements, each element written to the stream needs to be read at least once:*

$$\forall t \in T_\infty : \forall (W, s, i) \in \mathit{sacc}_W(t) : \exists t' \in T_\infty : (R, s, i) \in \mathit{sacc}_R(t')$$

Restriction 3.4 (Absence of unused elements between two elements that are used) *As views are matched one after another on consecutive elements of a stream, there cannot be any element between two accessed elements that is never written:*

$$\begin{aligned}&\forall s \in \mathcal{S} : \\ &(\exists t, t' \in T_\infty : (W, s, i) \in \mathit{sacc}_W(t) \wedge (W, s, i') \in \mathit{sacc}_W(t') \wedge i' > i + 1) \Rightarrow \\ &(\forall i'' \in \{i + 1, \ldots, i' - 1\} : \exists t'' \in T_\infty : (W, s, i'') \in \mathit{sacc}_W(t''))\end{aligned}$$

Note that this restriction does not need any additional verification as it results directly from the matching algorithm presented earlier.

Restriction 3.5 (Finite number of consumers for broadcasts) *As the tick construct advances the stream at some point and triggers the broadcast operation, additional peeking views on the same stream cannot be matched to the same producer after the tick. This mechanism effectively limits broadcasts to a finite number of receivers.*

This restriction also results directly from the matching algorithm and does not require any specific verification.

3.5 Compilation of an OpenStream program

During compilation of an OpenStream program, it is necessary to translate the OpenStream-specific pragmas and attributes to code that links with the OpenStream run-time library. The rest of the code must be treated as an ordinary program written in the C programming language and must be translated in accordance with its specification. Due to the complexity of the standard, writing such a compiler from scratch is a large undertaking. In addition, this work has already been accomplished in a large variety of existing C compilers, which can be used as a basis for the development of specialized compilers. Therefore, the OpenStream compiler is implemented on top of the GNU C Compiler version 4.7.0 [79], reusing existing compilation infrastructure [2].

The different steps involved in the compilation of an OpenStream application, including translation of the non-specific parts are shown in Figure 3.15. The basic structure of this process is already included in the unmodified version of GCC, but has been adapted to compilation of OpenStream programs. The order of the steps is not strict, in particular steps 2 to 5 are tightly coupled and executed repeatedly for each task. However, the steps can be roughly ordered as follows:

1. During *syntax analysis* the parser analyzes the pre-processed input file and converts the C statements into a tree representation called GENERIC [65]. OpenStream-specific clauses are represented by nodes with custom types and are processed in later stages.
2. During *outlining*, the compiler creates a work-function for each task body.
3. In the third step, the compiler determines how much space is needed for the data-flow frame containing the tasks metadata and its views.

2. The current version has been updated to version 4.9.0 of the compiler, but the results presented in this thesis were obtained from an earlier branch based on version 4.7.0.

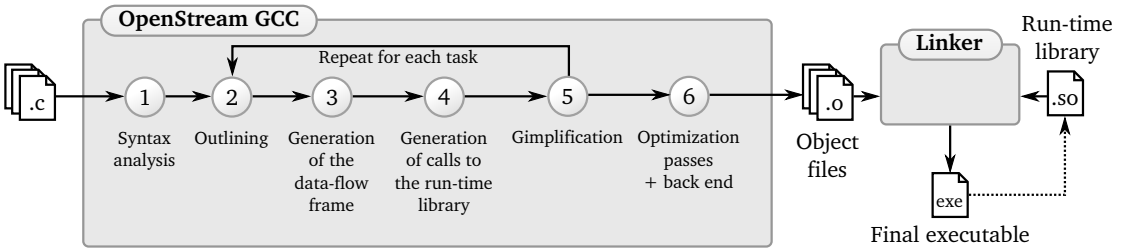


Figure 3.15: *Compilation of an OpenStream program*

4. Once the structure of the data-flow frame is known, the code initializing the fields and calling the appropriate run-time functions can be generated. In particular, the space needed for the data-flow frame is allocated by calling the allocation function of the memory pool seen in section 3.4.4, the synchronization counter of the task is set correctly and the views' bursts and horizons are initialized. For each view, a call to `resolve_dependences` is added.
5. During *gimplification*, the generated code is converted into a three-address representation of the GIMPLE intermediate representation, widely used in GCC.
6. The result of the gimplification is passed to subsequent optimization passes of GCC and the back end, which finally generates instructions for the target architecture.

The code of the run-time is kept in a separate, shared library. Hence, to resolve the symbols used by calls of run-time functions generated in step 4, the executable needs to be linked with the run-time library. The actual addresses of the symbols are determined when the executable and the run-time are loaded right before execution.

In the following example, we illustrate steps 2 to 4 on a task with two input views and an output view. The intermediate code resulting from the translation omits details from the actual implementation and therefore does not reflect the generated code by the real compiler. In addition, the OpenStream compiler does not use a source-to-source approach. Thus, the generated code only exists as an internal representation and is not exposed to the environment. However, the simplified code illustrates the concepts of how the compiler translates OpenStream code into generic C code making use of the run-time library.

Listing 3.6: Example code to be translated by the compiler

```
1  void stream_function(void)
2  {
3    ...
4    int horizon = 10;
5    int out_view[horizon];
6    float in_view_f[horizon];
7    double in_view_d[horizon];
8
9    #pragma omp task input(fstream >> in_view_f[horizon], \
10                          dstream >> in_view_d[horizon]) \
11                   output(istream << out_view[horizon])
12   {
13     for(int i = 0; i < horizon; i++)
14       out_view[i] = (int)round(in_view_f[i]*in_view_d[i]);
15   }
16   ...
17 }
```

The general lines of the code generated from listing 3.6 are represented by the listing below.

Listing 3.7: General lines of the code generated by the compiler

```
1  struct frame_1 {
2    size_t sc;
3    int horizon;
4    struct view in_view_f;
5    struct view in_view_d;
6    struct view out_view;
7    void (*work_fn)(void*);
8    char buf[];
```

```
9  };
10
11 void work_function_1(struct frame_1* fp)
12 {
13   for(int i = 0; i < fp->horizon; i++)
14     ((int*)fp->out_view.data)[i] = (int)round(((float*)fp->in_view_f.data)[i] *
15                                               ((double*)fp->in_view_d.data)[i]));
16
17   tdecrease(fp->out_view.owner, fp->out_view.horizon);
18   tend(fp);
19 }
20
21 void stream_function(void)
22 {
23   ...
24
25   int horizon = 10;
26
27   int out_view[horizon];
28   float in_view_f[horizon];
29   double in_view_d[horizon];
30
31   size_t frame_size = sizeof(struct frame_1) +
32                       horizon*sizeof(float) +
33                       horizon*sizeof(double);
34
35   struct frame_1* fp = tcreate(frame_size);
36
37   fp->work_fn = work_function_1;
38   fp->sc = horizon*sizeof(float) + horizon*sizeof(double) + horizon*sizeof(int);
39   fp->horizon = horizon;
40
41   fp->in_view_f.horizon = horizon*sizeof(float);
42   fp->in_view_f.burst = horizon*sizeof(float);
43   fp->in_view_f.next = NULL;
44   fp->in_view_f.rpos = 0;
45   fp->in_view_f.owner = fp;
46   fp->in_view_f.data = &fp->buf[0];
47
48   fp->in_view_d.horizon = horizon*sizeof(double);
49   fp->in_view_d.burst = horizon*sizeof(double);
50   fp->in_view_d.next = NULL;
51   fp->in_view_d.rpos = 0;
52   fp->in_view_d.owner = fp;
53   fp->in_view_d.data = &fp->buf[horizon*sizeof(float)];
54
55   fp->out_view.horizon = horizon*sizeof(int);
56   fp->out_view.burst = horizon*sizeof(int);
57   fp->out_view.next = NULL;
58   fp->out_view.rpos = 0;
59   fp->out_view.owner = NULL;
60   fp->out_view.data = NULL;
61
62   resolve_dependences(&fp->in_view_f, true);
63   resolve_dependences(&fp->in_view_d, true);
64   resolve_dependences(&fp->out_view, false);
65
66   ...
67 }
```

Lines 1 to 9 show the definition of the structure of the data-flow frame of the task. Each task created dynamically at execution time from the task construct upon a call to `stream_function` will be represented by an instance of this data structure. The definition of the structure specifies fields that are common to all tasks as well as fields that are specific to the task construct for which the structure was generated. The common fields are the synchronization counter `sc`, a pointer to the work function containing the instructions of the task body `work_fn` and a field `buf` that provides access to the input data of the task. Note that the size of `buf` is not specified as the size of input data is known earliest at task creation (cf. line 35). The task-specific fields are the views `in_view_f`, `in_view_d` and `out_view` and a field for the local variable `horizon` of `stream_function`.

The outlined task body is represented by the work function defined in lines 11 to 19. The data structure representing the task, i.e., the data-flow frame is passed as an argument to the function and the statements inside the function only reference fields from the data-flow frame. This also applies to the local variable `horizon` of `stream_function`, whose access has been replaced

with an access to `fp->horizon` in line 13.

Task creation takes place in the original function at line 35. The function responsible for task creation is `tcreate` and takes the size of the frame as an argument. This size is calculated from the size of the data structure representing the task and the amount of memory that is needed to store the task's input data in line 33. The allocation of the frame is carried out by the run-time during execution of `tcreate` and uses a memory pool. Hence, the code of `stream_function` does not directly call the allocator, but only assigns the return value of `tcreate` to `fp`.

The different fields of the data-flow frame are affected in lines 37 to 60. These are the fields containing metadata of the task itself, as well as the fields of the task's views. The fields of the views named `owner` point to the data-flow frame which embeds the elements of their sliding windows. For input views, the owner is always the frame containing the view and for output views this is the frame that contains the input view that was matched with the output view. The pointer is primarily used to find the correct data-flow frame when a synchronization counter needs to be decremented, such as in line 17 where the synchronization counter of the task's consumer is updated.

The code finishes with calls to `resolve_dependences` for every view in lines 62 to 64. The second parameter of this function indicates whether the call is issued for an input view (`true`) or for an output view (`false`).

3.6 Summary

In this chapter, we introduced OpenStream, a data-flow extension to OpenMP with support from streams and lightweight tasks. We showed the concepts of stream accesses using views and explained how a dynamic task graph can be derived from these accesses. The syntax of OpenStream programs was presented and illustrated with multiple examples. Moreover, we introduced the central data structures and procedures of the run-time in the discussion of the OpenStream execution model. We gave an overview of the steps of the compilation of an OpenStream program and outlined the generated code of an example.

OpenStream is a state-of-the-art language extension for task-parallel applications whose implementation enables the development of high performance applications [72]. Although the specific concepts of stream accesses using views and matching of producers and consumers are unique to OpenStream, the concept of specifying point-to-point data dependences between tasks is a trend for task-parallel languages in general [25, 70]. The *implementation* of our approaches for optimized scheduling and data placement presented in Chapter 7 and 8 and the implementation of our optimizations for broadcasts in Chapter 9 are tightly coupled with the OpenStream run-time and the OpenStream compiler. However, the *concepts* only rely on information on point-to-point data dependences between tasks and thus apply to other task-parallel languages as well.

Many of our concepts have been merged into the official distribution of OpenStream. As these apply to the run-time system, the majority of our contributions to the codebase are modifications of the OpenStream run-time. These are not only modifications that implement specific algorithms for scheduling and data placement, but also important changes to the technical infrastructure of the run-time, such as the interaction with the operating system, the addition of profiling support or NUMA-specific modifications of the run-time code. Some of the modifications also required changes in the OpenStream compiler or changes at the language level. Furthermore, the experimental evaluation of this thesis has lead to the development of multiple benchmarking applications, which have become part of the official distribution.

The next chapter presents changes to the run-time and execution model that enable efficient support for NUMA. In particular, we analyze the major issues of the NUMA-unaware strategy for memory allocation and propose concepts that solve these problems and support NUMA-aware scheduling and memory allocation.

4 A NUMA-aware run-time and execution model

The common requirement of techniques for NUMA-aware scheduling for task-parallel applications is that the placement of data structures involved in the execution of a task can be determined accurately within the run-time. NUMA-aware allocation is based on the ability of the run-time system to place data structures on specific nodes. Due to the short execution time of fine-grained tasks, this functionality is needed frequently throughout the execution of a task-parallel application and must thus be provided with low overhead. However, as data placement results from interactions between the task-parallel application, the operating system and the hardware at execution time, support for NUMA by the run-time cannot be implemented independently and must be integrated carefully into the embedding context. This requires a detailed understanding of all events that determine data placement and demands efficient use of the interface to collect information on the distribution of data provided by the operating system.

The goal of this chapter is to point out how the OpenStream run-time system can provide *efficient* infrastructure to support NUMA-aware scheduling and NUMA-aware allocation. The first part of the chapter explains which software and hardware components are involved in memory allocation and data placement from the perspective of the operating system. We show which events at execution time determine the placement on the different NUMA nodes of the machine and at which moment the placement takes place. We then discuss the influence of these mechanisms on the placement of data structures managed by an allocation mechanism based on memory pools, such as the allocator presented in the previous chapter. From this detailed understanding of the interactions, we conclude which changes must be applied to the OpenStream run-time system and which restrictions must be imposed on streaming applications in order to implement efficient and accurate NUMA-aware memory pooling. The resulting mechanisms allow the run-time to determine the location of structures and enable per-structure data placement with low overhead as a result of reduced interaction between the run-time and the operating system. The techniques form the basis of the solutions for NUMA-aware scheduling and NUMA-aware memory allocation presented in Chapter 7 and 8.

4.1 Memory allocation and data placement by the operating system

Modern general-purpose computing systems implement the concepts of *virtual memory* and *paging*. In this model, each user space process has its own, private space of virtual addresses, which is mapped to physical addresses by the *memory management unit* (*MMU*) using a translation

table managed by the operating system. Both virtual and physical memory are organized in pages, representing fixed-size intervals of the address space. The mapping from virtual to physical addresses is implemented with page granularity, i.e., the set of subsequent addresses of a page in virtual memory is mapped to a set of subsequent addresses of a physical page. The table defining the mapping between virtual and physical addresses is therefore referred to as the *page table*. *Memory protection* defines which types of accesses to memory are authorized (e.g., read-only mode that disallows write accesses, read and write mode without execution protection to prevent data from being interpreted as instructions, etc.). The protection can be configured individually for each page by setting flags for the corresponding entry of the page table. Illegal access to a page is detected by the hardware and results in an exception that is handled by the operating system.

4.1.1 Logical and physical memory allocation

Additional memory can be allocated by a user space process through a system call, usually wrapped by a function of a user space system library, such as `malloc` from the standard C library. The return value of the function call is a virtual address that points to a new region of memory that can be used by the process immediately. From the perspective of a process, memory allocation is thus an atomic operation and only involves a single system call. From the perspective of the operating system, however, additional memory is attributed to a process in two steps which we refer to as *logical allocation* and *physical allocation*.

Logical allocation is initiated by the system call issued by the process and causes the operating system to modify the page table, such that additional pages of virtual memory become accessible. The corresponding entries are set to point to the so-called *zero-page* and memory protection is configured to forbid write accesses to addresses associated to these pages. Although only the page table is modified and no additional physical memory is allocated, it *appears* to the process that additional memory becomes accessible upon return from the system call. Read access to the new virtual pages result in read accesses to the zero page and thus yield zero values as expected from a newly allocated memory region. However, as write accesses alter values in memory, they cannot be redirected to the zero page and require new pages of physical memory to be assigned to the process. This is done upon physical allocation and without notification of the process as explained below.

Due to the write protection configured during logical allocation, the first write access to a newly allocated page generates an exception. During exception handling by the operating system, an unused page of physical memory is selected and initialized with zeros. Afterwards, the entry in the page table is modified to point to the new physical page and the flags are changed to authorize write access. At the end of exception handling, control is transferred back to the application and the write access is repeated. As write access has been authorized during physical allocation, the repeated write access as well as subsequent accesses to the same page succeed without generating an exception.

Memory regions can be composed of multiple pages and it might thus be required that physical allocation is carried out multiple times. Hence, memory allocation from the perspective of the operating system is not an atomic operation, but can be distributed over a longer period of time depending on the timing of write accesses.

Figure 4.1 illustrates logical and physical allocation on a simple example in which a process requests three additional pages of memory. Note that in practice, the page table is not a flat table like the page table shown in the figure, but a hierarchical structure with multiple levels. However, for simplicity we illustrate memory allocation with a page table that has only a single level. The initial mapping is shown in Figure 4.1a, where a set of valid entries in the page table points to pages that are accessible both in read and write mode, while another set of unused entries shown at the bottom indicates that a part of the virtual address space cannot be accessed. At logical allocation, the entries of the page table corresponding to the unused region of virtual memory are modified, such that they point to the zero page and write protection is activated by setting the appropriate flags (Figure 4.1b). The first write to a page of the newly assigned region causes an exception (Figure 4.1c) and initiates triggers reservation of a previously unused physical page by

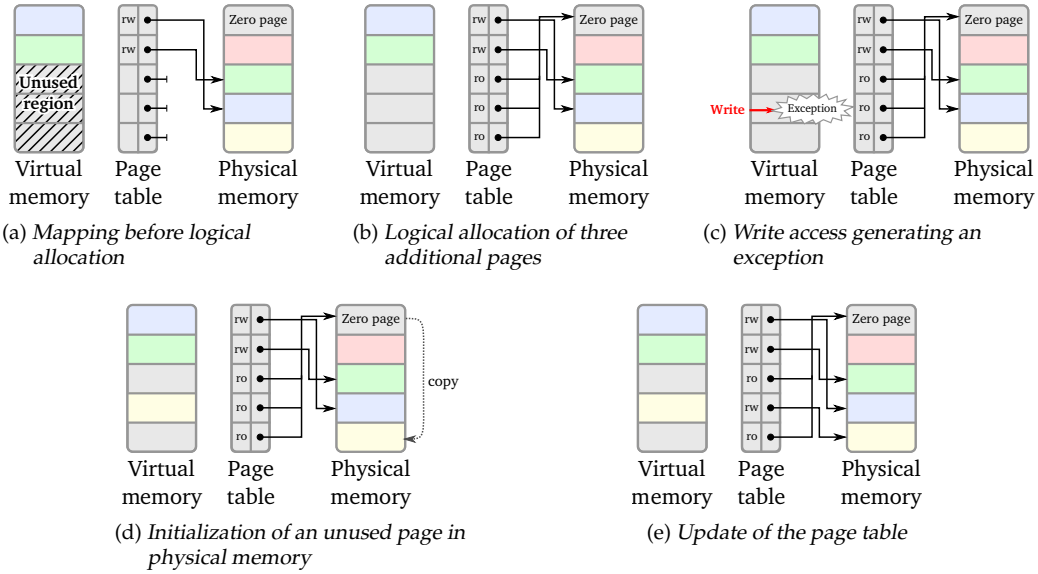


Figure 4.1: *Logical and physical allocation*

the operating system. This page is then initialized with zeros by copying the contents of the zero page to it (Figure 4.1d). Finally, the corresponding entry in the page table is updated and the flags of the entry are modified to allow read and write accesses (Figure 4.1e).

The main advantage of the separation of physical and logical allocation is a reduced memory footprint for processes that demand large amounts of memory, but which use memory only sparsely. Furthermore, the initialization of newly allocated memory can be distributed over time, which avoids that the execution of the allocating process is interrupted for a long duration at allocation time.

4.1.2 Page placement

On systems with non-uniform memory access, physical pages can be selected from multiple memory controllers at physical allocation. Which of the controllers is chosen depends on the placement strategy employed by the operating system. A common default placement strategy is *first-touch placement* that selects a page from the local memory of the core that executed the write instruction triggering physical allocation. Pages from remote nodes are only selected if all pages of the local node are already in use.

The placement resulting from first-touch allocation leads to a high fraction of local memory accesses if memory regions are initialized and accessed by cores from the same node. Sequential, independent processes, for example, display this behavior and thus benefit from this placement strategy. Also, if logical allocation is carried out on one node and write accesses are performed on a different node, first-touch placement increases data locality as it delays placement to the moment where the location of data accesses is known. However, for parallel applications with dynamic access patterns, first-touch placement does not necessarily provide adequate results for locality. An extreme case are applications that initialize data structures sequentially at the beginning of the execution and then process data in parallel. In this scenario, first-touch placement causes all data to be stored on the memory controller of the initializing node, which results in high contention and remote memory accesses in the parallel phase.

To give a process accurate control over page placement, operating systems usually provide system calls that allow processes to specify from which node the physical pages should be selected for a region of virtual memory (e.g., `mbind` provided by LIBNUMA [58] on Linux systems). These system calls are employed for two patterns of page placement: allocation on a single node and

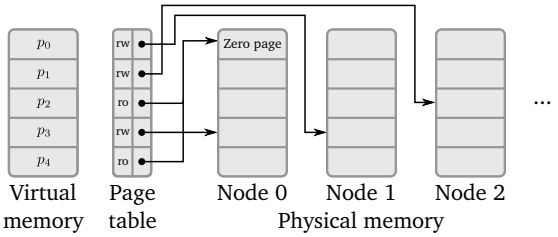


Figure 4.2: *Example of the distribution of data on three NUMA nodes*

interleaved allocation on multiple nodes. The former places all physical pages of a memory region on a single node and is particularly useful if the cores and thus the nodes accessing the memory region are known in advance. The latter uses a list of nodes on which physical pages are allocated in a round-robin fashion. Data is thus distributed evenly over multiple nodes, which enables exploitation of the overall bandwidth of multiple memory controllers and avoids contention on a single node. However, the distribution may increase the average latency of accesses as the likelihood of remote accesses increases with every additional node included in the interleaving.

4.1.3 Determining the location of data

Due to the first-touch allocation scheme, an application does not have a-priori knowledge about data placement, unless it specifies the distribution of data before physical allocation or unless it schedules the instructions initializing a memory region for execution on specific cores. To determine where the data of a memory region is located after physical allocation without specific placement and without detailed tracking of write accesses, the application must thus query the operating system explicitly through a system call, such as the `move_pages` system call of the Linux kernel [1]. This system call takes a list of virtual addresses and returns for each address on which node the page covering the address is located. For addresses whose associated pages have not been allocated physically prior to the call, the system call returns values indicating that the placement could not be determined.

Figure 4.2 shows an example for the distribution of data on three NUMA nodes. To determine the placement of the pages p_0 to p_4, the process passes pointers to two tables to `move_pages`. The first table has one entry for each page and contains the virtual addresses of the pages. The second table receives the results for the placement and is filled in by the system call. As p_0, p_1 and p_3 have been allocated physically, the table for the results contains the identifiers of the respective NUMA nodes at positions 0, 1 and 3. For the pages that have not been placed, i.e., $p2$ and p_4, the table contains negative values. The contents of the table are thus $1.$, 2, -1, 0 and -1.

4.1.4 Implications of the size of pages

Modern hardware platforms provide large amounts of main memory ranging from a few MiB on embedded systems to several hundred GiB on high performance servers. Traditionally, many hardware platforms and operating systems only provided support for a single page size of a few KiB, e.g., 4 KiB on older x86 platforms. The gap between the small page size and the large size of contiguous chunks of memory that can be allocated by an application leads to a high number of entries in the page table. The *translation lookaside buffer* (*TLB*), which caches these entries, has only a limited capacity. A translation from a virtual to a physical address that misses the TLB causes the hardware to fetch the entry of the page table that is needed for the translation from main memory. This additional access to memory takes a certain amount of time to complete and slows down execution. Large page tables may result in frequent misses of the TLB and can thus decrease performance significantly.

1. Linux uses the same system call to obtain information about page placement and to migrate pages between nodes, hence the name `move_pages`.

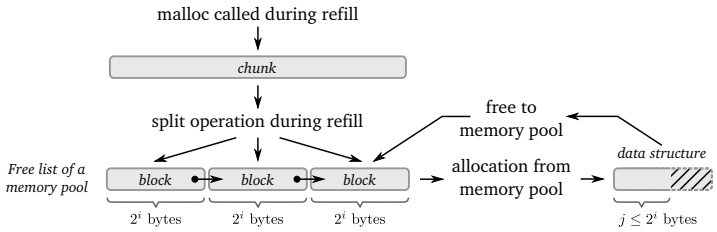


Figure 4.3: *Illustration of the terms used for memory regions managed by memory pools*

In order to reduce the number of entries in the page table and thus to decrease the likelihood of misses, modern platforms are able to handle larger pages. For example, on x86_64 platforms, the Linux kernel is able to handle pages of 4 KiB, 2 MiB and 1 GiB. Recent versions of the kernel integrate so-called *transparent huge page support* [42], where the kernel tries to allocate huge pages upon physical allocation transparently, without explicit requests for huge pages from the application.

However, while huge pages can reduce the number of TLB misses, they increase the granularity of physical allocations and data placement. With small pages, physical allocation occurs frequently, but each time only a small portion of memory is allocated physically. Huge pages reduce the frequency, but increase the amount of memory that is allocated. Even a small modification can thus cause a large amount of memory to be allocated physically and to be placed on a NUMA node.

In the following section, we discuss the consequences of first-touch placement and the page size on memory pooling.

4.2 The influence of first-touch placement and the page size on memory pooling

In this section, we show how the mechanisms for first-touch placement presented in the previous section influence how and when blocks managed by memory pools are placed on NUMA nodes. We start by examining the influence on refill operations and consider placement during the use of a block afterwards. Implications on the reuse of blocks are pointed out at the end of the section. For a clear distinction between memory regions obtained from the operating system, memory regions managed by memory pools and memory regions actually used by the run-time system and the application we use the following terminology:

- a *chunk* of memory refers to a memory region that has been allocated from the operating system (e.g., by calling `malloc`). Chunks are allocated and are divided into smaller regions during refill operations (as explained in Section 3.4.4).
- a *block* is a memory region resulting from a split operation on a chunk during a refill. As mentioned in Section 3.4.4, the size of a block is always a power of two. Blocks are chained in free lists of memory pools and are handed to the run-time upon allocation from a memory pool.
- a *data structure* refers to a memory region that is used to store an entity of the run-time or the application (e.g., a data-flow frame or a stream). The memory region of a data structure is a subset of the block that was allocated for the data structure, starting at the first address of the block.

The relationship between these terms is illustrated in Figure 4.3.

4.2.1 Page placement during refills

At allocation of a data structure from a memory pool, the allocation function first checks if the list of free blocks corresponding to the size of the structure is empty. If the list is not empty, the first block is removed from the list and returned as the result of the allocation. However, if the list is empty, the memory pool performs a refill operation in order to populate the list with new

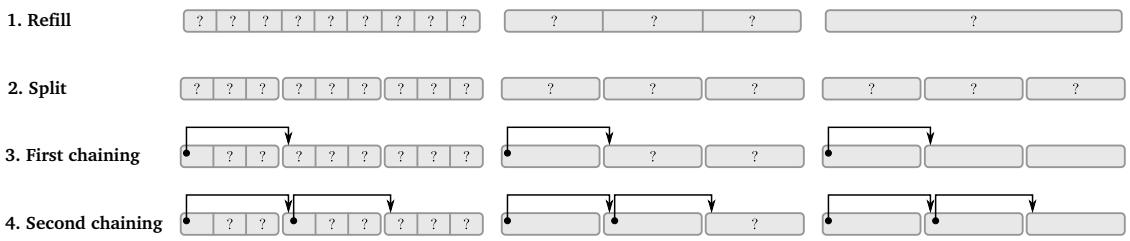


Figure 4.4: *Physical allocation upon a refill of a free list*

blocks. During a refill, a memory chunk of a configurable size (e.g., 2 MiB) is allocated logically from the operating system and is split into blocks that correspond to the block size of the free list that is to be refilled. The chain of blocks is realized by using a small portion of memory at the beginning of each block to store the address of the next block in the chain. Hence, the first write access to a block takes place immediately during the refill, but only affects a very small amount of the block's memory. However, depending on the page size used by the system and the size of the block, this determines either the placement of a portion of the block, the placement of the entire block or placement of the block as well as following blocks at subsequent addresses.

Figure 4.4 illustrates these three cases. The bar at the top of each figure (step 1, labeled *Refill*) represents the chunk of memory allocated from the operating system, which is split into blocks afterwards. The individual blocks resulting from the split are shown below the chunk (step 2, labeled *Split*). The last two lines (step 1, labeled *First chaining* and *Second chaining*) show the placement of pages after the chaining of the first block and the second block, respectively. Question marks in the figure indicate that the respective page has not been allocated physically and that its placement is thus unknown.

In Figure 4.4a the block size S_B is greater than the page size S_P. The chaining of each block only causes physical allocation and thus placement of the first page of each block. Hence, the resulting chain for the free list consists of blocks for which the placement of the majority of the pages still remains to be determined. Figure 4.4b shows the same steps for an equal size of pages and blocks. At each chaining, the entire block is allocated physically and the resulting chain only contains blocks that have been placed entirely. The last case is presented in Figure 4.4b, where a single page contains multiple blocks. The chaining of a block thus causes the current block as well as following blocks to be allocated physically. As in the previous scenario, this leads to a chain of placed blocks.

In summary, allocation from a free list after a refill operation either yields a block that has already been allocated physically or a block whose first page has been placed, but whose remaining pages have only been allocated logically. Which percentage of a block the unplaced pages represent in the latter case depends on the block size and the size of pages. Figure 4.5 illustrates three different cases. In the first case shown in Figure 4.5a, the block is composed of only two pages, such that 50% of the block are placed after a refill. In the second case (Figure 4.5b) the block composed of four pages, such that 25% are placed and in the last case (Figure 4.5c) a block of consists of eight pages with 12.5% of placed data.

The typical sizes of data structures present in the run-time when running the applications used for experimental evaluation presented in Section 6.1 can be divided into two classes. The first class consists of small structures of a few bytes up to a few KiB. These are mainly small data-flow frames and other small entities, such as the instances of structures representing streams. The second class represents large structures of several hundred KiB and is composed exclusively of data-flow frames. The size of pages on our test systems described in Section 6.3 is selected transparently by the operating system through transparent huge page support and is either 4 KiB or 2 MiB. The size of small structures is thus close to the minimal page size and the blocks used for the small structures are almost always allocated physically entirely after a refill, independently from the

(a) 50% *placed* (b) 25% *placed* (c) 12.5% *placed*

Figure 4.5: *Different amounts of placed data after a refill for blocks larger than a page*

actual page size. In contrast to this, the status of the placement of a large structure after a refill depends on the size of a page. For small pages only a very small portion of a structure is allocated physically and new structures can be considered as entirely unplaced. When using huge pages, however, the size of a large structure is below or equal to the size of a page and new structures are placed entirely.

Hence, intermediate cases with the same amount of placed and unplaced data as in Figure 4.5a do not appear in practice and all structures can be considered as either entirely placed or entirely unplaced. In addition, as only large structures can be unplaced after a refill and as all large structures are frames, unplaced structures are always data-flow-frames. In the following part, we examine the different possible scenarios for page placement of unplaced data-flow frames during their first use. This applies only to frames whose size is significantly higher than the size of a page as this is the only scenario in which a frame can be unplaced right after a refill.

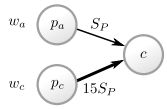
4.2.2 Placement at the first use of data structures

The first write accesses to a data-flow frame occur when the producers of the task associated to the frame write data to the task's input views. Depending on the number of producers, the amount of input data they provide and where they execute, two scenarios for the placement of the frame's pages are possible.

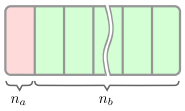
In the first scenario, all pages of the data-flow frame are placed on a single node. This happens if all producers are executed by workers of the same node. The lower the number of producers of a task, the higher the probability that all of them are executed on the same node. For tasks with only a single producer, the data-flow frame is guaranteed to be placed on a single node. Also, for highly unbalanced dependences, the outcome for data placement is similar as the majority of the pages is written by one worker and the frame can be considered as being placed on a single node. Figure 4.6a shows such a task graph, in which one producer p_a writes a single page of input data, while another producer p_b writes 15 pages of the input data of a task c. Assuming that p_a is executed by a worker w_a that operates on node n_a and p_b is executed by a worker on node n_b, the resulting distribution corresponds to the placement shown in Figure 4.6b.

In the second scenario, pages of the data-flow frame are scattered across multiple nodes. This is the case if the task associated to the frame depends on several producers which are executed on different nodes. The number of nodes is that contain the frame's pages is limited by the number nodes of the system and the number of producers. The reason for the limitation by the number of producers is that task execute from beginning to end on a single core and thus on a single node[2]. Figure 4.6c shows a task graph with balanced dependences between three producers p_a, p_b and p_c and a consumer c. The producers are executed by workers w_a, w_b and w_c and, similar to the previous example, these workers operate on different nodes n_a, n_b and n_c. A possible page placement resulting from this situation is given in Figure 4.6d. The actual order of the regions belonging to n_a, n_b and n_c within the frame can depend on the number of input views of c, the order of the matching of the views of c, p_a, p_b and p_c and the number of streams that are involved in the matching. If c has only a single input view, as in Figure 4.7a, and if the views provide access to elements of the same stream, the order of the memory regions depends on the order of calls to `resolve_dependences` for the output views of p_a, p_b and p_c. For example, if the call to `resolve_dependences` for p_b is issued before the call for p_a and if the call for p_c is issued after the call for p_a, then the order of the regions on different nodes is n_b, n_a, n_c instead of the order

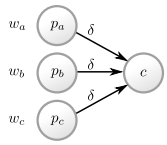
2. The only exception to this rule are tasks whose execution is interrupted by a taskwait and which are resumed on a different core. However, OpenStream applications use barriers only rarely, e.g., towards the end of the execution of the application, such that task migration can be neglected.



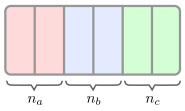
(a) *Task graph with unbalanced dependences*



(b) *Placement of the majority of the frame's pages on a single node*

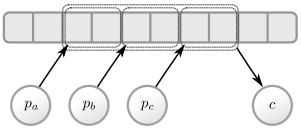


(c) *Task graph with balanced dependences*

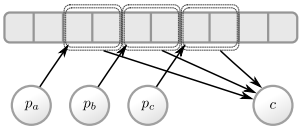


(d) *Even distribution of the frame*

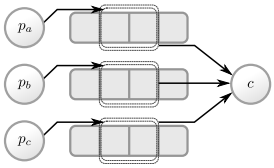
Figure 4.6: *Balanced and unbalanced dependences leading to different distributions of the pages of a frame*



(a) *One input view and three output views*



(b) *Three output views and three input views on the same stream*



(c) *Three output views and three input views on different stream*

Figure 4.7: *Different relationships between output and input views with different implications on the order of the scattering of a view*

n_a, n_b, n_c given in Figure 4.6d. In Figure 4.7a, c has three input views which provide access to elements on the same stream. In this case, the order of the memory region depends on the order of calls to `resolve_dependences` of the output views as well as the order of the calls for the input views. However, the latter is defined statically during translation by the OpenStream compiler as described in Section 3.5 and depends on the order of the input clauses in the source code. Finally, it is also possible that c has three input views that provide access to elements of three different streams as shown in Figure 4.7c. In this scenario the order of the memory regions only depends on the order of the input clauses.

4.2.3 Reuse of data structures

At the end of the execution of a task, its data-flow frame is freed by the worker that executed it. This consists of handing the frame back to the memory pool of the worker and adding the embedding block to the appropriate free list. Depending on the placement of the pages of frames that have already been freed by the worker before and the blocks resulting from earlier refills, the memory pool can thus contain a composition of (a) blocks whose pages are placed on the same node as the worker associated to the memory pool (b) blocks whose pages are placed on a remote node (c) blocks whose pages are scattered across multiple nodes and (d) blocks whose pages are not allocated physically. Subsequent allocations reuse these blocks, e.g., allocations of data-flow frames for new tasks. As the allocator always returns the block that was added last to a free list and as the blocks within the list are not sorted by their placement, the pages of the block that is returned for an allocation can be placed in any of the aforementioned ways.

Hence, by using first-touch placement in conjunction with memory pooling, the run-time does not have any direct control over the placement of data. However, NUMA-aware allocation and NUMA-aware scheduling rely on fine-grained control over the placement of data and are thus

difficult to implement with the mechanisms for memory management above. In the following part, we propose two techniques that address this problem. The first technique avoids scattering of blocks on multiple nodes by separating buffers for input data from frames and by imposing a restriction on the use of streams by an application. The second technique combines per-node memory pools with an efficient mechanism to detect the placement of blocks and avoids the presence of remotely placed blocks in memory pools.

4.3 Separation of frames and input buffers

The main circumstance that leads to scattered frames is that the input data written by multiple producers is combined in a contiguous region of virtual memory embedded into a single data structure, namely the data-flow frame. Figure 4.8a provides a detailed view of the run-time structures after the matching of views for the task graph with balanced dependences of Figure 4.6c. The `data` pointers of the producers' output views point to the data region of the input view of c, which is embedded into the data-flow frame of c. Hence, the actual scattering affects the pages of this data region.

4.3.1 Avoiding the scattering of input data across multiple nodes

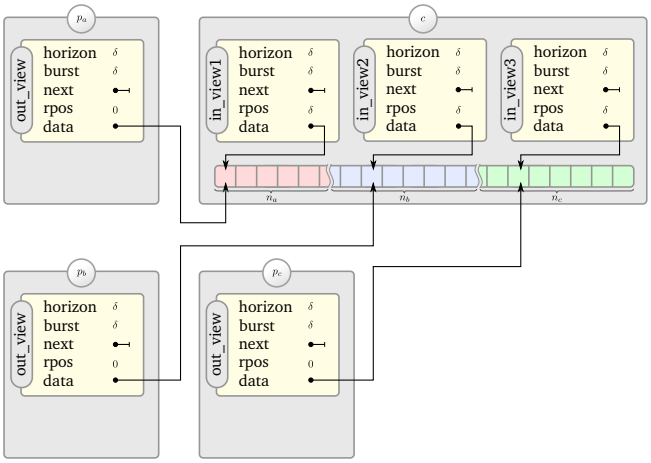
If each producer targeted a different data structure, there would be only one writer per structure and every structure could only be placed on a single node. Hence, to avoid scattered data-flow frames, the buffers for input data of a task should be separated from the data-flow frame. Figure 4.8b shows the same matching of views as before, but with separate memory regions for each of the input views of c. Each data pointer of the output views now points to the base of a distinct data structure and thus prevents that more than one producer writes to a contiguous memory region. In the remainder of the thesis, we refer to these structures as *input buffers*. The different input buffers of a task can potentially be located on a different node, but if employed correctly as described below, each input buffer is entirely placed on a single node, i.e. none of the input buffers can be scattered across multiple nodes.

However, even if each input view of a task has its own input buffer, it is still possible that the data of an input view is provided by multiple producers. This happens if multiple output views with a smaller burst are matched with a single input view with a larger horizon as in the code example below:

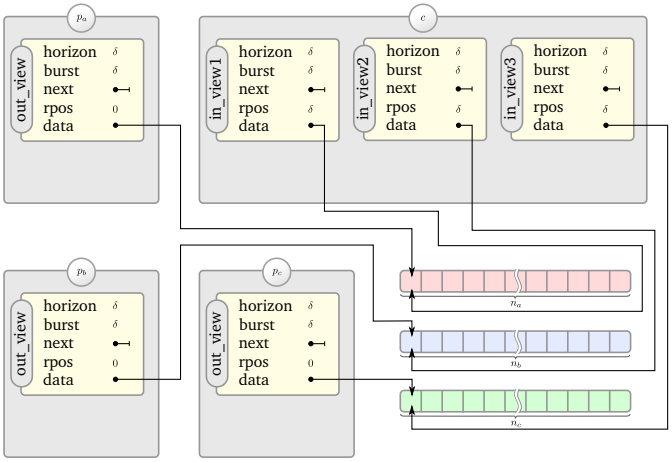
Listing 4.1: Multiple producers writing to the same input buffer

```
1  float a_stream __attribute__ ((stream));
2
3  int horizon_in = 6 * DELTA;
4  int horizon_out = 2 * DELTA;
5  float out_view[horizon_out];
6  float in_view[horizon_in];
7
8  for(int i = 0; i < 3; i++) {
9    /* Producer */
10   #pragma omp task output(a_stream << out_view[horizon_out])
11   {
12     for(int i = 0; i < horizon_out; i++)
13       out_view[i] = some_function(i);
14   }
15 }
16
17 /* Consumer */
18 #pragma omp task input(a_stream >> in_view[horizon_in])
19 {
20   for(int i = 0; i < horizon_in; i++)
21     printf("Read %f\n", in_view[i]);
22 }
23
24 #pragma omp taskwait
```

Figure 4.9 shows the run-time structures after matching of views in that situation. Due to the addressing scheme for data of input views, which is identical to indexation of an array, input



(a) *Buffers integrated into a frame*



(b) *Separate data structures for frames and input buffers*

Figure 4.8: *Separation of input buffers from data-flow frames*

data of a single view must be stored in a contiguous region of memory and cannot be split across multiple objects. The only way to avoid situations in which multiple producers write to a single input view is to impose a restriction on programs that forces the burst and the horizon of matched views to be identical.

Restriction 4.1 (One-to-one matching of input and output views) *To avoid that multiple output views provide write access to elements to which a single input view provides read access, the burst of each output view must be identical to the horizon of the matched input view:*

$$\forall t \in T_\infty : \forall (W, s, i_s, i_e) \in \mathit{views}(t) :$$
$$\forall t' \in T_\infty : \forall (R, s, i'_s, i'_e) \in \mathit{views}(t') :$$
$$([i_s, i_e] \cap [i'_s, i'_e] = \emptyset \vee [i_s, i_e] = [i'_s, i'_e])$$

This ensures that each input buffer has a unique writer and thus makes it impossible that an input buffer is scattered across multiple nodes due to write accesses to output views.

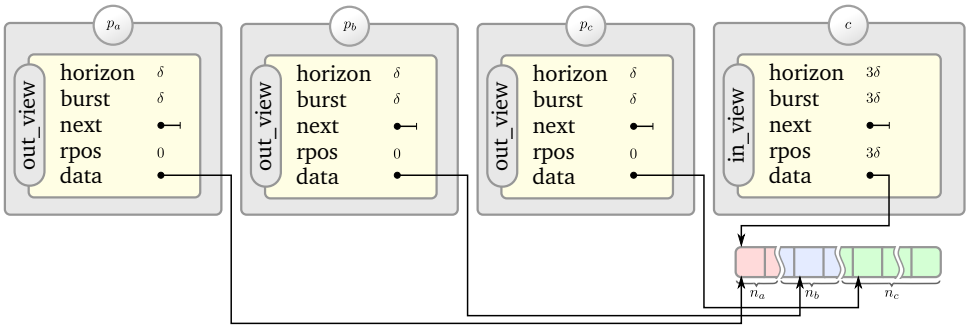


Figure 4.9: *Multiple writers of an input view with input buffers separated from data-flow frames*

4.3.2 Integration into the compiler

The separation of input buffers from data-flow frames does not only imply changes to the run-time, but also requires a modification of the compiler. In the default scheme for code generation for input data embedded into data-flow frames, the OpenStream compiler generates a single call for the allocation of the entire data-flow frame. For input buffers that are separated from the frame, the compiler must generate a call to the allocator function for each input buffers.

Consider the example below, in which a single task with three input views is defined.

Listing 4.2: Example of a task with multiple input views

```
1  void stream_function(void)
2  {
3    ...
4    int horizon = 10;
5
6    float in_view_f[horizon];
7    int in_view_i[horizon];
8    double in_view_d[horizon];
9
10   #pragma omp task input(fstream >> in_view_f[horizon], \
11                          istream >> in_view_i[horizon], \
12                          dstream >> in_view_d[horizon]) \
13   {
14     ...
15   }
16
17   ...
18 }
```

In the default scheme for code generation the program above is translated into the code of the next listing. As explained in the previous chapter in Section 3.5, the code generated by the compiler only exists in an intermediate representation and the code below only serves as an illustration.

Listing 4.3: General lines of the code generated by the compiler for input data embedded into a data-flow frame

```
1  struct frame_1 {
2    size_t sc;
3    int horizon;
4    struct view in_view_f;
5    struct view in_view_i;
6    struct view in_view_d;
7    void (*work_fn)(void*);
8    char buf[];
9  };
10
11 ...
12
13 void stream_function(void)
14 {
15   ...
16
17   size_t frame_size = sizeof(struct frame_1) +
18                       horizon*sizeof(float) +
19                       horizon*sizeof(int) +
```

```
20                          horizon*sizeof(double);
21
22    struct frame_1* fp = tcreate(frame_size);
23    ...
24    fp->in_view_f.data = &fp->buf[0];
25    fp->in_view_i.data = &fp->buf[horizon*sizeof(float)];
26    fp->in_view_d.data = &fp->buf[horizon*sizeof(float)+horizon*sizeof(int)];
27    ...
28  }
```

The data-flow frame embedding the input data is allocated by a call to `tcreate` in Line 22. The assignment of addresses within the data region of the frame to the data pointers of the input views takes place in Lines 24 to 26. The code generated for separate input buffers is presented in the next Listing.

Listing 4.4: General lines of the code generated by the compiler for input buffers that are separated from the data-flow frame

```
1   ...
2
3   void stream_function(void)
4   {
5     ...
6
7     size_t frame_size = sizeof(struct frame_1);
8     struct frame_1* fp = tcreate(frame_size);
9     ...
10    alloc_view_data(&fp->in_view_f, horizon*sizeof(float));
11    alloc_view_data(&fp->in_view_i, horizon*sizeof(int));
12    alloc_view_data(&fp->in_view_d, horizon*sizeof(double));
13    ...
14  }
```

A first difference is the calculation of the size of the data-flow frame in Line 7. As the frame does not contain input data any more the size of the allocation is equal to the size of the structure representing the frame. Furthermore, the initialization of the data pointers has been replaced with calls to `alloc_view_data` in Lines 10 to 12. This function allocates a buffer of the size specified by the second argument from a memory pool and assigns the result to the data pointer of the view that was passed as the first argument.

4.4 NUMA-aware memory pools

With input buffers separated from data-flow frames and the additional restriction on the horizon and burst of matched input and output views, each data structure that is allocated from a memory pool is placed entirely on a single node. However, it is still possible that, due to the liberation of buffers of tasks executed earlier, the free list of a worker's memory pool contains blocks that have been placed on remote nodes. In this section, we introduce NUMA-aware memory pools, where each pool is associated to a NUMA node and only contains blocks that have been placed on the node. When a data structure is freed, the run-time determines on which node the embedding block has been placed, looks up the corresponding memory pool and adds the block to the appropriate free list of the pool. This requires the run-time system to be able to determine the placement of a block accurately and efficiently, i.e., the identification of the placement of a block must be correct and its overhead on execution time should be as low as possible. We first describe how the placement of blocks can be determined efficiently and accurately and then show how these mechanisms can be integrated into the life cycle of blocks in order to enable NUMA-aware memory pooling.

4.4.1 Determining the placement of blocks

A naive solution to determine on which node a block has been placed is to query the operating system for the placement of each of the block's pages every time information on its placement is needed. This requires that the addresses that correspond to page boundaries within the address range of the block are determined and passed to the operating system using the `move_pages` system call. As the operating system must traverse the data structure representing the address

space of the requesting process for each of the block's pages, each such call takes a certain amount of time to complete. In addition, the time of each call depends on the number of concurrent calls from multiple threads, as shown below.

Figure 4.10 shows the duration of one call to `move_pages` with increasing concurrency for our two test platforms (described in Section 6.3) with 64 and 192 cores, respectively. The block size used for each call corresponds to a typical size of 512 KiB for an input buffer in the applications used in the experimental evaluation of this thesis. As the minimal page size on the test systems is 4 KiB, the addresses passed to `move_pages` correspond to the page boundaries of small pages. Each data point in the graphs represents the mean value for a total of 50 runs of a synthetic benchmark that measures the average duration of one call to `move_pages` for a set of threads, where each thread performs 10,000 calls to `move_pages`. The error bars indicate the standard deviation. For a low number of concurrent requests, the duration of a single call remains between 10 kcycles and 20 kcycles on both platforms, but becomes orders of magnitude higher when all the cores of the machine are used.

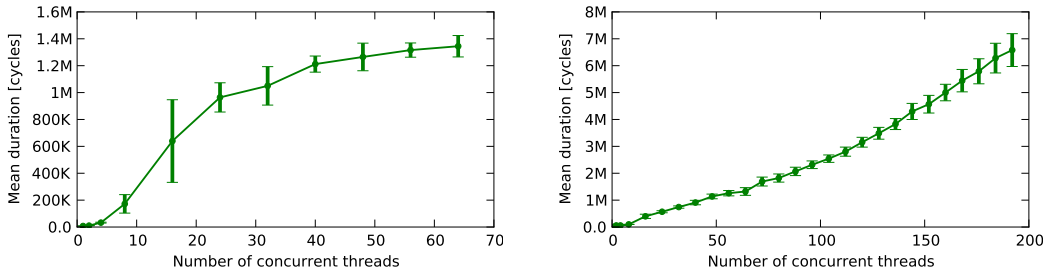
However, in a real-world scenario cores execute other instructions between two calls and thus do not constantly query the operating system. Figure 4.11 shows the mean duration of one call to `move_pages` as a function of the number of idle cycles between two calls when using all cores of the machines. For the 64-core machine the duration drops rapidly and reaches the minimal duration at about 4 Mcycles of idle time between two calls. In contrast to this, the duration on the 192-core machine drops slower and remains high even if several million cycles lie between two calls to `move_pages`, as shown in Figure 4.11b. The minimal duration is reached between 8 Mcycles and 10 Mcycles.

For NUMA-aware data placement and NUMA-aware scheduling, information on the placement of a block is needed before the execution of a task. Hence, the duration between two calls corresponds to the duration of a task. Figure 4.12 interprets the duration between two calls as the task duration and shows the relative overhead in percent of the calls to `move_pages` on execution time. The stippled lines indicate a limit of five percent, which we consider as the highest acceptable overhead. The graph shows four different curves, each for a different amount of addresses passed to `move_pages`, ranging from a single address (*1 page*) to all addresses that represent page boundaries of small pages of the block (*all pages*).

Interestingly, the determination of the placement of all pages is faster than determining the placement of a single page on the 64-core system, although the overhead increases from a single page to ten pages. Figure 4.13, showing the duration of one call as a function of the number of pages included in each query, provides more detailed information on this issue. The number of cycles between two calls was set to 1.5 Mcycles, which corresponds to the duration with the largest gap between the overhead for determination of the placement of a single page and determination of the placement of all pages. As can be seen in the graph, the duration of one call increases with the number of pages included in the query, until it reaches a maximum at about 90 pages. For a higher number of pages, the duration decreases and reaches its minimum for the total number of 128 pages of a 512 KiB block. However, as Figure 4.12a shows that the duration of a task must be higher than 2.5 Mcycles to stay below the limit of five percent independently from the number of pages included in each query, we do not have investigated the origins of the unexpected behavior above.

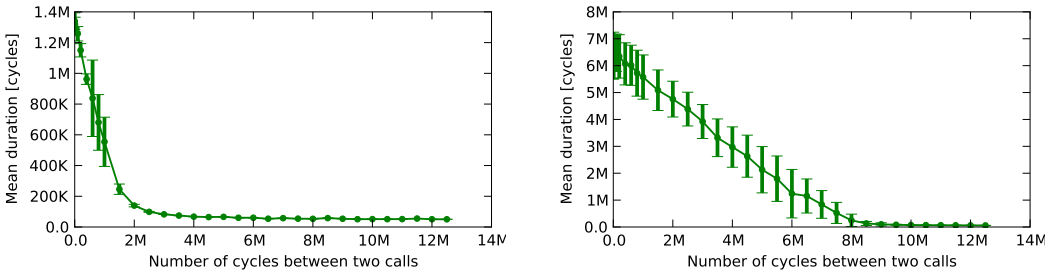
For the 192-core system the correlation between the number of pages per query and the overhead is much clearer, as the overhead increases with the number of pages with a minimum for a single page and a maximum for the entire set of pages of a buffer. The minimal duration of a task in order to stay below the limit for the overhead depends on the number of pages that are included in a query. If all pages of the block are included, tasks should take more than 8 Mcycles, while for a single page, the overhead drops below five percent at about 5 Mcycles.

In conclusion, the duration of a task should be at least higher than 5 Mcycles in order to stay below the threshold for the overhead on execution time on both systems. However, the typical duration of tasks in the applications that we have used for experimental evaluation can be below 1.5 Mcycles and is below 5 Mcycles for most of the applications. Hence, systematic redirection of



(a) *64-core system* (b) *192-core system*

Figure 4.10: *Duration of a call to* `move_pages` *with increasing concurrency*



(a) *64-core system* (b) *192-core system*

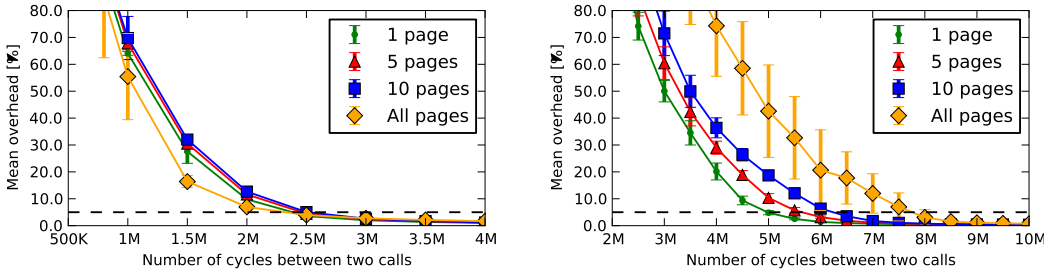
Figure 4.11: *Duration of a call to* `move_pages` *with maximum concurrency and varying duration between two calls*

requests to obtain the placement of the pages of blocks is not a viable option for the run-time.

As we have shown above, the overhead on execution time related to the determination of data placement is conditioned by three parameters: the number of concurrent system calls, the total number of calls and the number of pages whose placement is to be determined with each system call. In the following part, we introduce two techniques to reduce the number of system calls and one technique to decrease the number of pages per call.

Determination of the placement only for large blocks

The overhead for the determination of the placement of the input data of a task increases with the number of the input buffers associated to the task as for each input buffer at least one call to `move_pages` is necessary. Hence, for tasks with a large number of input dependences,



(a) *64-core system* (b) *192-core system*

Figure 4.12: *Overhead of a call to* `move_pages` *with maximum concurrency as a function of the duration between two calls for a varying number of pages*

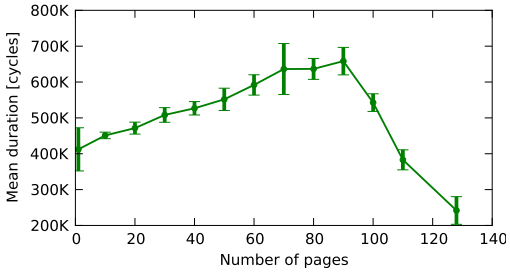


Figure 4.13: *Duration of a call to* `move_pages` *on the 64-core system as a function of the number of pages whose placement is determined with* 1.5 Mcycles *between two calls*

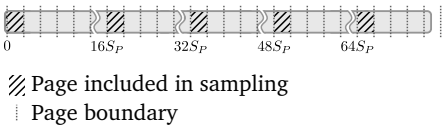


Figure 4.14: *Page sampling with a sampling distance of 16 pages*

the overhead can compensate the possible improvements of the execution time resulting from NUMA-aware scheduling and NUMA-aware allocation. Reduction of this overhead requires more elaborated techniques to determine the placement as simple calls to `move_pages` for each of the input buffers. However, in the applications studied in this thesis, such tasks only represent a small fraction of the total number of tasks.

The placement of small input buffers is only crucial for the execution time of a task if the task only reads from and writes to small buffers. The vast majority of tasks has either strongly unbalanced dependences, e.g., one input buffer of 512 KiB and a few input buffers of less than 4 KiB, or balanced dependences with relatively large input buffers, e.g., two input buffers of 512 KiB. The buffers whose placement is crucial for performance thus all exceed a threshold of a few KiB, such that the run-time does not need to determine the placement of small buffers.

As a first measure to reduce the number of system calls, the run-time can thus neglect buffers whose size does not exceed the threshold. A value of 10 kB enables discrimination between small and large buffers, we have thus configured the run-time to determine only the placement of input buffers that are larger than 10 kB.

Page sampling

As shown in the previous section, one can assume that every input buffer is placed entirely on a single node. However, in order to determine on which of the nodes contains the buffer, it is not sufficient to determine only the placement of a single page. For example, for large input buffers composed of small pages, the first page might be allocated on the node that performed the refill operation and the rest of the pages might be allocated on the node that first wrote input data to the buffer. Sampling only the second small page or a page somewhere in the middle of a block is not sufficient either, since for huge pages it is possible that a page spans two or more blocks occupying neighboring memory region. Figure 4.15 illustrates this situation. As the blocks are not aligned to boundaries of huge pages, the page in the center of the figure contains data of two blocks. From a block's perspective, this means its data might be located on two different nodes as is the case for the second block whose data is located on n_b and n_c.

However, determining the placement of all pages that form a buffer is not required either, since the number of small pages for large buffers is much higher than the maximal number of nodes that could contain the pages. A simple sampling technique that determines the placement of every nth small page is sufficient to determine where the majority of the pages of the buffer are placed. Using this technique, the containing node is defined as the node with the highest number of samples.

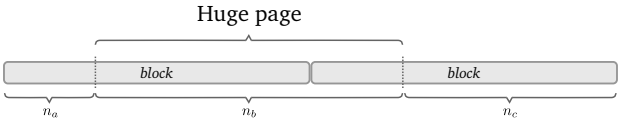


Figure 4.15: *Huge page spanning two blocks*

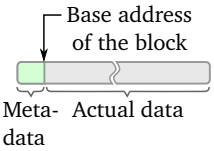


Figure 4.16: *Layout in memory of a block and its metadata section*

Figure 4.14 illustrates sampling of every 16th small page. The size of a small page is indicated by S_P and only pages with an offset which can be represented as an integer multiple of $16S_P$ are sampled. For the experiments we have used a sampling distance of 64 KiB, i.e., every 16th small page.

However, situations in which substantial parts of a buffer are placed on more than one node occur only very rarely, for less than one percent of the buffers. Thus, sampling less pages per buffer might be sufficient in most cases. As the overhead of the sampling of every 16th small page is already sufficiently small, we did not investigate if the sampling distance can be increased or whether sampling at specific positions of the buffer is sufficient.

Caching of information about data placement

Last and most important, querying the operating system multiple times for the placement of the same block is expensive in terms of execution time and is not necessary. As pages are never migrated between nodes unless explicitly requested by the application, it is sufficient to determine the initial placement of a block and to reuse this information each time information about the block's placement is needed afterwards. This information about the placement can be cached in a small metadata section in front of a block, as shown in Figure 4.16. This layout in memory enables rapid determination of information on placement simply by calculating the address of the metadata section from the block's base address and by accessing the appropriate field of the metadata structure. Storing metadata sections in front of the actual data is a common technique in memory management [84].

4.4.2 Integration into the life cycle and per-node memory pools

The methods presented above aim at reducing the overhead associated to the determination of the placement of blocks, but we did not discuss at which moment the procedure to determine the initial placement of a block should be triggered. In order to obtain correct results for the placement, it must be ensured that the sampling takes place after physical allocation of all of the block's pages. As explained in Section 4.1.1, this is the case after the pages have been written for the first time. For input buffers, which are the only data structures whose size exceeds the threshold, this means that the sampling can only take place when the producer task writing its output data to the buffer has terminated. As all the producers of a task that becomes ready are guaranteed to have terminated, the run-time can thus safely determine the placement of the input buffers of a task when the synchronization counter of the task reaches zero.

Upon termination of a task, its input buffers are not used anymore and must be freed. To avoid that a memory pool contains blocks from different nodes it is necessary to free an input buffer to a memory pool of a worker that executes on a core of the node that contains the block's pages. As there are as many workers as cores per node, the run-time would have to choose a target pool among the pools of the same node. In addition, a worker that needs to allocate a buffer on its local node, but whose free list of the appropriate size is empty would either have to search through all pools associated to the same node or it would have to initiate a refill operation. By grouping blocks of the same node in a single memory pool and by sharing this pool among the workers that execute on the procedure for allocation and liberation of buffers can be simplified. Allocation of a buffer can be carried by checking a single memory pool and liberation of a buffer can be implemented simply by handing the buffer to the unique memory pool of the node containing the block in which

the buffer was embedded. We refer to this approach as *per-node memory pools*.

Allocation of a data structure in a memory pool of a node yields either (a) a data structure of small or huge pages that has already been placed entirely on the node that the memory pool is associated to or (b) a structure composed of small pages that have not been allocated physically except the first page, which is allocated on the node of the worker that initiated the refill operation from which the structure originates or (c) a data structure that consists of a huge page that is placed entirely on the node that triggered the refill operation from which the structure originates. It is worth noting that the last situation only occurs when a worker of a remote node triggered the refill operation and when the structure is used for the first time. As each structure is freed to the correct memory pool after use, cases (b) and (c) become less likely over time, such that in most cases an allocation from a memory pool yields a structure that is entirely placed on the node to which the pool is associated. This provides the run-time system with fine-grained control over data placement through allocation from an appropriate memory pool.

4.5 Reducing the impact of per-node memory pools on performance

A drawback of per-node memory pools compared to worker-private pools is that multiple workers compete for the resources provided by a pool. Hence, the free lists of a pool need to be protected against concurrent accesses, e.g., using locks, which can introduce huge overheads for high concurrency. However, the number of workers per pool depends on the number of cores per node, which is typically relatively low in order to avoid congestion on the resources shared by the cores of a node. For example, on both of our test systems only eight cores share a node. Thus, the most important sources of overhead are the critical sections protected by the locks rather than the operations to acquire and release a lock. Freeing a structure to or unchaining a block from a free list are fast, since they only entail an update of pointers. The most time-consuming operations on free lists are refills, as these operations issue system calls for logical allocation and trigger physical allocation upon the chaining of new blocks. In this section, we propose a set of techniques that aim at reducing the duration of critical sections and which thus reduce the impact on performance resulting from the use of per-node memory pools.

Allowing concurrent operations during refills

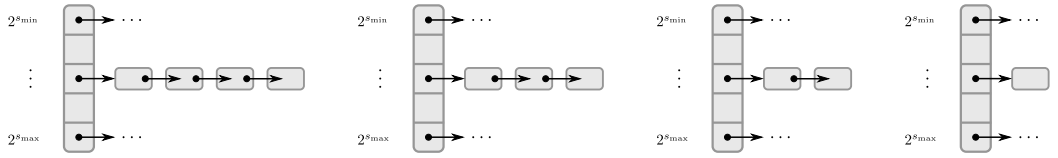
During a refill operation on a list other workers might try to free data structures to or allocate data structures from the same list. Keeping the list locked during the refill blocks these operations and prevents the run-time system from reusing existing blocks rapidly. Hence, when a worker detects that a refill is necessary it should release the lock on the free list immediately after detection. During the system call other workers can free and allocate blocks without waiting for the refill operation to finish. When the system call for memory allocation returns, the lock can be re-acquired and the resulting new data blocks can be added safely to the list. When the refill is complete, the lock is released and the new blocks become available.

Avoiding eager physical allocation at a refill through lazy splitting

Another time-consuming operation is the physical allocation of pages during a refill caused by the chaining of blocks. The amount of data that is allocated physically and thus the duration of the chaining depends on the page size, the size of blocks and the size of the chunk that is allocated from the operating system and split into individual blocks. For example, for a fixed chunk size of 2 MiB and a page size of 4 KiB the chaining of blocks of 512 KiB causes four pages or 16 KiB to be allocated physically, while for blocks of 32 KiB using the same page and chunk size a total of 64 pages or 256 KiB must be allocated physically.

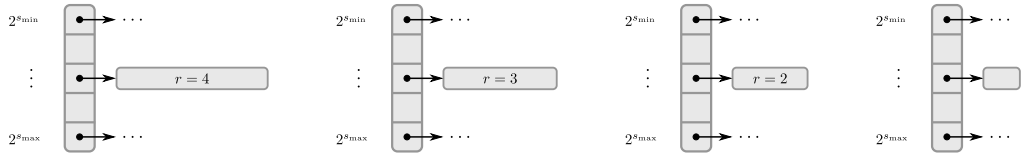
By default, the blocks resulting from a split of a chunk are all chained in the free list at the end of the refill operation as shown Figure 4.17a. We refer to this technique as *immediate splitting* of the chunk. At each allocation afterwards, one block is removed from the list and the chain of the remaining blocks forms the new free list (Figure 4.17b to 4.17d).

The number of physical allocations at the refill can be reduced by employing a mechanism which we refer to as *lazy splitting*. Figure 4.18 illustrates how this mechanism works. Instead of



(a) *Refill* (b) *Allocation of the first block* (c) *Allocation of the second block* (d) *Allocation of the third block*

Figure 4.17: *Refill and allocation with immediate splitting*



(a) *Refill* (b) *Allocation of the first object / first split* (c) *Allocation of the second object / second split* (d) *Allocation of the third object / last split*

Figure 4.18: *Refill and allocation with lazy splitting*

splitting the entire chunk obtained from the operating system into smaller blocks, the whole chunk is added to the free list as if the chunk was a an ordinary block, but with an indication for the number of blocks that can be obtained from the chunk (Figure 4.18a). At the first allocation after the refill, a first block is separated from the chunk and the remaining part with a reduced number of block forms the new free list (Figure 4.18b). This process continues, until the remaining part is reduced to a single block (Figure 4.18c and 4.18d). The refill operation only touches a single page to store the number of blocks in the chunk, just like each allocation afterwards. Lazy splitting thereby avoids eager physical allocation, such that the overhead for physical allocation and initialization of memory is distributed over time and thus reduces the duration of a refill.

4.5.1 Reducing the number of system calls for logical allocation

Finally, the last time-consuming operation during a refill is related to interaction with the operating system. At each refill, the run-time must issue a call to the operating system in order to trigger logical allocation of a new memory region that will be used as the chunk for the refill. In order to reduce the number of system calls, a technique similar to lazy splitting can be employed. Instead of allocating a chunk from the operating system at each refill, a larger chunk of memory is allocated for each memory pool at initialization. The allocation of smaller chunks needed for refills can be carried out entirely in user space, simply by using a memory region from the large chunk. When the large chunk has been consumed entirely, a new large chunk must be allocated to satisfy further refills.

4.6 Placement of persistent run-time structures

An efficient NUMA-aware run-time does not only need to care about placement of dynamic objects for the task-parallel application, but must also place its own data structures efficiently in order to prevent itself from becoming bottleneck for performance. A data structure that is used exclusively within the run-time and whose placement is critical for performance is the structure representing a worker. The size of this structure is low, but each instance is accessed frequently as it contains a work-stealing deque and a single entry software cache for the scheduling of tasks. As

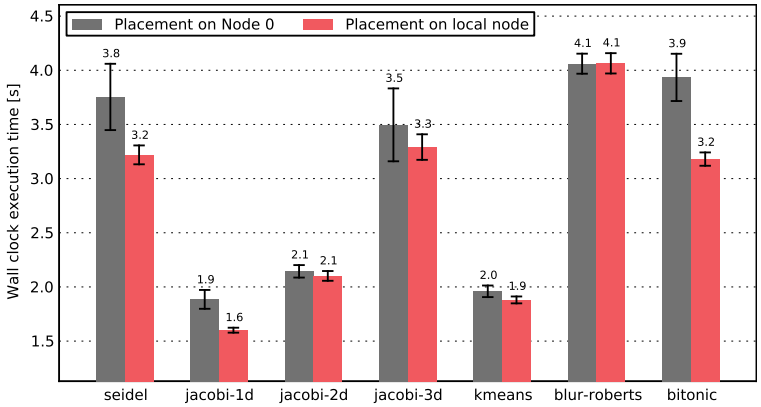


Figure 4.19: *Influence of the placement of structures representing workers on performance*

each of the instances is primarily accessed by the worker associated to it, each instance should be placed on the node of the worker in order to increase the locality of memory accesses.

However, workers are allocated and initialized during set-up of the run-time, which is done sequentially. Hence, first-touch placement would cause all of the structures to be allocated on a single node. Therefore, it is necessary to use explicit placement on the respective nodes, e.g., by using the `mbind` function mentioned in Section 4.1.2. Another issue is the layout of these instances in memory. As they are very small, using an ordinary array to store them would cause several instances to be located in the same page, which leads to multiple structures being placed on the same node. To avoid this, the structure must be padded to the size of a page and each instance must be placed individually.

Figure 4.19 shows the wall clock execution time in seconds for the dynamic single assignment versions of the benchmarks presented in Section 6.1 on the 192-core machine with 24 NUMA domains. The first bar for each benchmark represents the median execution time of 50 executions for the version of the run-time which places all worker structures on the first node of the system. Error bars indicate the standard deviation. The second bar shows the same value for the version of the run-time which places each structure representing a worker on the local node of the worker.

For most of the benchmarks (*seidel*, *jacobi-1d*, *jacobi-3d*, *bitonic*) the median execution time can be reduced significantly when placing the structures on appropriate nodes. In addition, the variation is often lower (*seidel*, *jacobi-1d* and *bitonic*). For the other benchmarks the performance of both versions is approximately the same. Hence, placing the worker structures on the local nodes of the workers can be considered as beneficial.

4.7 Summary

The solutions presented in this chapter allow the run-time to determine the placement of data efficiently and accurately and to place data structures on specific nodes. We presented the mechanisms behind the placement of data on the different NUMA nodes of the machine from the perspective of the operating system and discussed the influence of the default page placement policy, first-touch placement, and the size of pages on memory allocation using memory pools. We identified the scattering of data as a first problem for accurate data placement and proposed the separation of input data and data-flow frames as well as a restriction on the programming model as a solution. In the second part, we introduced an efficient run-time mechanism to determine the placement of blocks and showed how this mechanism can be integrated into the life cycle of data managed by memory pools. Finally, we introduced per-node memory pools that allow the run-time system to allocate data structures on specific NUMA nodes. In the last section, we focused on the placement of data structures of the run-time that are allocated at the beginning of the execution and remain in use until termination and concluded that explicit placement of these

structures using the operating system interface is sufficient.

The methods presented in this chapter are based on first-touch placement, which is the default mechanism for placement employed by the Linux operating system kernel. This strategy makes it necessary to determine explicitly on which node data has been placed after the first write accesses to the respective pages. For future work it would be interesting to investigate the behavior of other placement strategies as well. For example, a per-node memory pool could force physical allocation of a chunk on the local node during a refill operation and store the information about the placement directly in the metadata section without querying the operating system. However, the run-time would have to take into account that such a predefined placement might fail if the targeted node cannot provide unused physical pages.

Also, the thresholds presented in the chapter allow the run-time to discriminate between large and small buffers for the benchmarks studied in this thesis. For a more generic approach that supports tasks with input buffers of different sizes, the actual values for the thresholds have to be determined more accurately or even the concept of using thresholds has to be revised in future work.

5 Dynamic single assignment

In the previous chapter, we introduced per-node memory pools that provide the run-time system with the ability to place input buffers accurately on NUMA nodes and that allow the run-time to determine the placement of an input buffer efficiently. These capabilities are a necessary condition for NUMA-aware scheduling and NUMA-aware data placement presented in Chapter 7 and Chapter 8. However, to benefit from these optimizations, the run-time must be able to determine which data is accessed by a task and to control its placement.

In this chapter, we introduce programming based on *dynamic single assignment* (*DSA*) on stream elements, which puts management of memory accessed by tasks under the responsibility of the run-time. This programming style fully exploits the concepts of data-flow tasks and is naturally supported by the OpenStream programming model. We show how programs based on dynamic single assignment meet the requirement above and illustrate the required implementation steps starting from a sequential program. We then discuss the influence of the control program on data locality and contention in these applications and show that sequential task creation can have a negative impact on these aspects. As a result of this analysis, we conclude that task creation by a parallel control program is often beneficial. The conditions for the parallelization of the control program are sketched at the end of the chapter.

The chapter is organized as follows. Section 5.1 introduces the basic concepts of dynamic single assignment and provides definitions for dynamic single assignment with respect to addresses and dynamic single assignment with respect to stream indexes. Section 5.2 provides a more formal view on deriving the working set of a task from the information made available to the run-time by using dynamic single assignment. Section 5.3 introduces an informal methodology for the implementation of dynamic single assignment using OpenStream, starting from a sequential implementation of an algorithm. The influence of sequential task creation on the memory footprint and data locality of an application is discussed in Section 5.4. The chapter finishes by sketching the conditions for the parallelization of the control program in Section 5.5.

5.1 Concepts of dynamic single assignment

Before we explain the principles and go into the details of dynamic single assignment, we first set the terminology that is used in the rest of the chapter.

5.1.1 Terminology

We define a *data element* as an entity of data that can be read and modified and refer to the range of addresses that is occupied by a data element as its *data location*. Whenever a data element is modified, a new *version* of the element is generated. The values of two versions of a data element do not necessarily have to be different. For example, a new version with the same value could be generated by assigning the return value of a function, which in a particular case yields the same value as the previous version of the data element. We illustrate the terms defined above on a simple example with a set of local integer variables `i`, `j` and `k` that are declared, initialized and manipulated by a function as in the following listing.

Listing 5.1: Illustration of the terminology for dynamic single assignment

```
1  void foo(void)
2  {
3    int i = 0;
4    int j = 0;
5    int k = 0;
6
7    while(some_condition) {
8      i = bar(i);
9
10     if(some_predicate(i)) {
11       j = baz(i, j);
12       k = doz(i, k);
13     }
14   }
15
16   return i + j + k;
17 }
```

As all variables are declared locally within the scope of the function their addresses belong to the program stack and are defined when `foo` is called. Their data locations are thus only defined during execution of the function. As `i` receives a new value at each iteration, a new version of `i` is also generated at each iteration. The updates of the other variables depend on some predicate of `i`, which might not be true for each value of `i`. Hence, there are not necessarily new versions of `j` and `k` at each iteration and the total number of generated versions of `i`, `j` and `k` might be different upon return from `foo`.

The example also shows that there is a tight coupling between data elements and data locations. The data location of an element is defined before its first reference when `foo` is entered and remains valid until the data element is discarded at the end of the function. Different versions of the data element are thus stored at the same data location.

5.1.2 Principles of dynamic single assignment

The main concept of dynamic single assignment is to use a different data location for each new version of a data element and thus to update each data location at most only once. This implies that a data element cannot be updated without changing its location and that in-place updates are not allowed. This decouples data elements from locations and allows the system to choose a new location at every update. In contrast to *static single assignment* (*SSA*), the number of versions is not necessarily known statically and might depend on values that are only known at execution time. The following manual example, which declares an array of integers with one array element per version of a data element `i`, illustrates this concept.

Listing 5.2: Example of manual dynamic single assignment

```
1  int i[n];
2  i[0] = 0;
3  i[1] = i[0]*5 + 3;
4  i[2] = i[1]*i[1];
5  i[3] = i[2]/3;
6  ...
```

At each update of `i` a new array index and thus a new data location is chosen to store the newly generated version. Note that dynamic single assignment in general does not require a specific

mapping of versions to data locations as in the example above, where versions are stored at locations of subsequent array indexes. As long as a location is only used at most for a single version, the mapping is correct. The following listing uses different indexes of the array to store the same versions as in the previous listing and is also a valid example for dynamic single assignment.

Listing 5.3: Example of manual dynamic single assignment with an irregular mapping of versions to data locations

```
1  int i[n];
2  i[0] = 0;
3  i[2] = i[0]*5 + 3;
4  i[715] = i[2]*i[2];
5  i[1] = i[715]/3;
6  ...
```

The formal restrictions for the mapping of versions to data locations can be defined as follows. Let $\mathcal{E}$ be the set of all possible data elements and let versions of an element be identified by a unique integer with zero identifying the initial version of an element. Let further $\mathcal{A} \subset \mathbb{N}_0$ be the set of all addresses of a flat address space and let $\text{loc} : \mathcal{E} \times \mathbb{N}_0 \to \mathcal{P}(\mathcal{A})$ be the function that maps each version of a data element to a data location. A data location is defined by a finite set of addresses at which the data of an element can be stored. For dynamic single assignment loc is restricted, such that:

$$\forall e, e' \in \mathcal{E} : \forall v, v' \in \mathbb{N}_0 : (e \neq e' \vee v \neq v') \Rightarrow \text{loc}(e, v) \cap \text{loc}(e', v') = \emptyset$$

Hence, an address is either not used at all or it belongs to a specific version of a specific data element.

5.1.3 Dynamic single assignment on streams

The set of streams and stream indexes of an OpenStream program can be seen as an unbounded, two-dimensional address space that allows to implement dynamic single assignment based on stream accesses, where each version of a data element is stored at a different set of stream indexes. Let $\text{loc}_s : \mathcal{E} \times \mathbb{N}_0 \to \mathcal{P}(S \times \mathbb{N}_0)$ be a function that maps each version of a data element to a set of stream indexes. For dynamic single assignment, this function must thus fulfill the same restriction as loc defined above:

$$\forall e, e' \in \mathcal{E} : \forall v, v' \in \mathbb{N}_0 : (e \neq e' \vee v \neq v') \Rightarrow \text{loc}_s(e, v) \cap \text{loc}_s(e', v') = \emptyset$$

Hence, every OpenStream program that passes all data elements through streams, i.e., each OpenStream program that does not use global variables or pointers to memory regions that are shared by multiple tasks, fulfills the restrictions for dynamic single assignment by construction. However, due to the execution model of OpenStream, the mapping of stream elements to addresses is not necessarily unique. Stream data is stored in input buffers and due to memory pooling these buffers can be reused. It is thus possible that the same address is used multiple times to store different stream elements. Thus, a program based on dynamic single assignment with respect to the address space formed by streams and stream indexes is not necessarily a program with dynamic single assignment with respect to the address space formed by memory addresses of the machine. We illustrate this aspect on a short example, given in the listing below.

Listing 5.4: Stream indexes and addresses in the context of dynamic single assignment

```
1  int istream[4] __attribute__((stream));
2
3  int i_in;
4  int i_out;
5  int i3;
6
7  /* Initialization task */
8  #pragma omp task output(istream[0] << i_out)
9  {
10   i_out = 5;
11 }
12
13 /* Task: t0 */
14 #pragma omp task input(istream[0] >> i_in) \
```

```
15    output(istream[1] << i_out)
16  {
17    i_out = i_in*i_in;
18  }
19
20  /* Task: t1 */
21  #pragma omp task input(istream[1] >> i_in) \
22    output(istream[2] << i_out)
23  {
24    i_out = i_in+3;
25  }
26
27  /* Task: t2 */
28  #pragma omp task input(istream[2] >> i_in) \
29    output(istream[3] << i_out)
30  {
31    i_out = i_in/3;
32  }
33
34  /* Termination task */
35  #pragma omp task input(istream[3] >> i_in)
36  {
37  }
```

As can be verified easily, all versions of `i` are associated to different streams and stream indexes: the first version of `i` is stored at the first index of `istream[0]`, the second version at the first index of `istream[1]` and so on. However, the input data of a task is stored within the input buffers associated to the task and the address of a stream element is defined by the input buffer that contains this element. The memory pooling mechanism of Section 3.4.4 and Section 4.4.2 allows an input buffer to be freed if it is not used any longer and, more importantly, to be reused by another task afterwards. Hence, in the example, the data-flow buffer of `t0` might be reused for `t2` and version 0 and version 2 might be stored at the same addresses, although they are associated to different stream elements.

In the remainder of this document, we use the term dynamic single assignment to refer to dynamic single assignment on streams. Furthermore, a version of an element is defined as the value of the stream elements that represent the data element. Intermediate versions that may be generated during execution of a task, but which do not correspond to the final values written to the stream, are not considered as versions. For example, if a task reads a stream element and copies its value to a task-local variable, modifies this variable multiple times and writes the result back to an element of an output stream, only the value of the element at the beginning and at the end of the task are considered as versions with respect to dynamic single assignment. The following listing provides an example of such a behavior.

Listing 5.5: Task-local modifications not counted as versions

```
1  int a_stream __attribute__((stream));
2  int another_stream __attribute__((stream));
3  int i_in, i_out;
4
5  #pragma omp task input(a_stream >> i_in) \
6      output(another_stream << i_out)
7  {
8    int i = i_in;
9
10   for(int j = 0; j < N; j++)
11     i += some_function(i);
12
13   i_out = i;
14 }
```

The intermediate versions generated in the loop body do not count as versions, while the initial value read from `a_stream` and the final value written to `another_stream` do.

5.2 Obtaining accurate information on data accesses

As stated at the beginning of this chapter, the main reason for the introduction of dynamic single assignment is to be able to determine the working set of a task accurately in order to optimize

the locality of memory accesses by scheduling the task near its data or by placing the data actively near the core that executes a task. In this section, we show how the working set of a task can be determined by the run-time system before execution of the task, based on the information provided by dynamic single assignment.

Let $\mathcal{M} \subset \mathbb{N}_0 \times \mathcal{A} \times \{R, W\}$ be the set of all possible memory accesses, where a triple $(\tau, a, u) \in \mathcal{M}$ represents a memory access at time τ to address a in mode u. Let $\tau_S(t)$ and $\tau_E(t)$ be the start and end of a t task with $\tau_S, \tau_E : T_\infty \to \mathbb{N}_0$. The set of memory accesses of a task t is defined as $\text{acc} : T_\infty \to \mathcal{P}(\mathcal{M})$ with $\forall(\tau, a, u) \in \text{acc}(t) : \tau_S(t) < \tau < \tau_E(t) \wedge |\{(\tau, a, u)\}| = 1$. The former condition specifies that all memory accesses of a task take place after start and before end of the task, while the latter specifies that only one memory access can take place at a time. We define the *working set* $\text{ws} : T \to \mathcal{P}(\mathcal{A})$ of a task as the set of distinct data locations accessed during execution of the task, i.e., $\text{ws}(t) = \{a | \exists(\tau, a, u) \in \text{acc}(t)\}$.

Determining the working set of a task before task execution implies that $\text{ws}(t)$ is known at a time $\tau < \tau_S(t)$. In Section 5.1.3 we defined that the values of each version of all data elements in dynamic single assignment are stored exclusively in streams, which implies that all relevant data is stored in input buffers. The only exception to this rule are the first and the final version, which are usually handled by initialization and termination tasks and which are stored in shared memory as explained below. However, in the benchmarks presented in Section 6.1, the number of versions generated by the main computation tasks is far higher than the two versions handled by the auxiliary tasks. Hence, for most of the tasks, the rules defined for dynamic single assignment apply: all data handled by the task is stored in streams. In practice, a small portion of data is still read from shared memory for convenience, especially parameters of the application or the parameters for the granularity are often accessed from shared memory. However, these accesses only represent a small fraction of the total number of memory accesses carried out by a task, such that access to shared memory can be neglected and the task can be considered as conforming to the restrictions of dynamic single assignment. The advantage of using dynamic single assignment on stream elements is that $\text{acc}(t)$ is restricted to accesses to input buffers and output buffers managed by the run-time and whose addresses and sizes are known when a task becomes ready.

Let $\text{addr} : T_\infty \times \mathcal{S} \times \mathbb{N} \to \mathcal{A}$ be a partially defined function that maps a stream element to its address from the perspective of a task. Note that there is no globally unique mapping of stream elements to addresses, as the same element can be available at multiple addresses when copied by a broadcast. Thus, the mapping is only unique from the perspective of each task. Using this definition, the working set of a task t is simply the union of all the addresses of all stream accesses $\text{sacc}(t)$ (defined in Section 3.1.2) of the task:

$$\text{ws}(t) = \bigcup_{\substack{(u,s,i) \\ \in \text{sacc}(t)}} \{\text{addr}(t, s, i)\}$$

Furthermore, it can be derived from the execution model that the elements which are made accessible by a view are mapped to consecutive addresses:

$$\forall t \in T_\infty : \forall(u, s, i_s, i_e) \in \text{views}(t) : \forall i \in \{i_s{+}1, ..., i_e\} : \text{addr}(t, s, i) = \text{addr}(t, s, i_s){+}(i{-}i_s){\cdot}\text{size}(s)$$

Hence, the working set of a task can be represented by a set of consecutive address regions $\text{ws}_C(t) \subset \mathcal{A} \times \mathcal{A}$, where each region is defined by its first and its last address:

$$\text{ws}_C(t) = \bigcup_{\substack{(u,s,i_s,i_e) \\ \in \text{views}(t)}} \{(\text{addr}(t, s, i_s), \text{addr}(t, s, i_e))\}$$

This set can easily be determined from dependence resolution of each view, since the starting address of each of the pairs in $\text{ws}_C(t)$ is the data pointer of the associated view and the end address can be determined by multiplying the horizon with the element size and by adding the result to the start address.

This makes profiling of the working set or deriving the working set from specific properties of the program structure (e.g., memory accesses in leaf tasks of divide-and-conquer algorithms) unnecessary and provides a reliable method for the prediction of a significant subset of a task's memory accesses before its execution.

5.3 Implementing an algorithm using dynamic single assignment

In this section, we illustrate the implementation of an algorithm using dynamic single assignment on the one-dimensional version of the *seidel* benchmark named *seidel-1d*, calculating the average of three neighboring elements at each iteration of the algorithm. We start from a sequential version and develop a task-parallel version that can be used as a drop-in replacement of the original implementation. The process can be summarized as follows:

1. Identification of the data elements and versions
2. Partitioning of the data elements
3. Mapping to stream elements and definition of the interface of tasks generating new versions
4. Definition of auxiliary tasks needed for initialization and termination
5. Implementation of all tasks
6. Parallelization of the control program

Parallel control programs have not been introduced earlier and require some explanation. In this section, we provide only an example of a parallel control program as a motivation and discuss the implications of a parallel control program and restrictions of the parallelization in Section 5.4 and 5.5.

5.3.1 Identification of data elements, versions and appropriate partitioning

The sequential implementation of *seidel-1d* is straightforward: at each iteration, each element of an array of double precision floating point values is updated according to its own value and the values of the its left and right neighbors. The elements at the first position and at the last position are treated as if their left and right neighbors, respectively had a constant value of zero. The following listing shows an implementation of the complete algorithm.

Listing 5.6: Sequential implementation of *seidel-1d*

```
1  void seidel_1d_seq(double* data, size_t N, int num_iter)
2  {
3    for(int iter = 0; iter < num_iter; iter++) {
4      /* Leftmost element */
5      data[0] = (0 + data[0] + data[1]) / 3.0;
6
7      /* Elements in the center */
8      for(size_t i = 1; i < N-1; i++)
9        data[i] = (data[i-1] + data[i] + data[i+1]) / 3.0;
10
11     /* Rightmost element */
12     data[N-1] = (data[N-2] + data[N-1] + 0) / 3.0;
13   }
14 }
```

Obviously, the data elements in this application are the elements of the array, for which each iteration yields a new version. The partitioning of the data determines the amount of data treated by each task and therefore indirectly determines how much work must be carried out per task. Parallelism is also conditioned by the partitioning, as it determines how many tasks can execute in parallel. In addition, the size of the data treated by a task can have an influence on how well caches are exploited. If the block size is bigger than the capacity of a cache and if data is referenced frequently within the task, the cache miss rate might be high. Hence, the size of a data block is often constrained by the cache size. As the characteristics of the hardware can differ from one machine to another it is possible that a partitioning that yields good performance on one system performs poorly on another system. Hence, using a static partitioning scheme might not yield the same performance across multiple machines.

A solution to this problem is to implement the application with variable granularity, whose actual value is defined at execution time. The granularity which yields minimal execution time among the possible values can then be determined experimentally on each machine without modification of the implementation. For the example of the one-dimensional stencil, the array can be partitioned into blocks whose size is specified at execution time. The sequential version with variable granularity is shown in the listing below.

Listing 5.7: Sequential, blocked implementation of a *seidel-1d*

```
1  void seidel_1d_seq_blocked(double* data, size_t N, size_t B, int num_iter)
2  {
3    for(int iter = 0; iter < num_iter; iter++) {
4      /* Leftmost element */
5      data[0] = (0 + data[0] + data[1]) / 3.0;
6
7      /* Leftmost block */
8      for(int i = 1; i < B; i++)
9        data[i] = (data[i-1] + data[i] + data[i+1]) / 3.0;
10
11     /* Blocks in the center */
12     for(size_t i = B; i < N-B; i += B)
13       for(size_t j = 0; j < B; j++)
14         data[i+j] = (data[i+j-1] + data[i+j] + data[i+j+1]) / 3.0;
15
16     /* Rightmost block */
17     for(int i = N-B; i < N-1; i++)
18       data[i] = (data[i-1] + data[i] + data[i+1]) / 3.0;
19
20     /* Rightmost element */
21     data[N-1] = (data[N-2] + data[N-1] + 0) / 3.0;
22   }
23 }
```

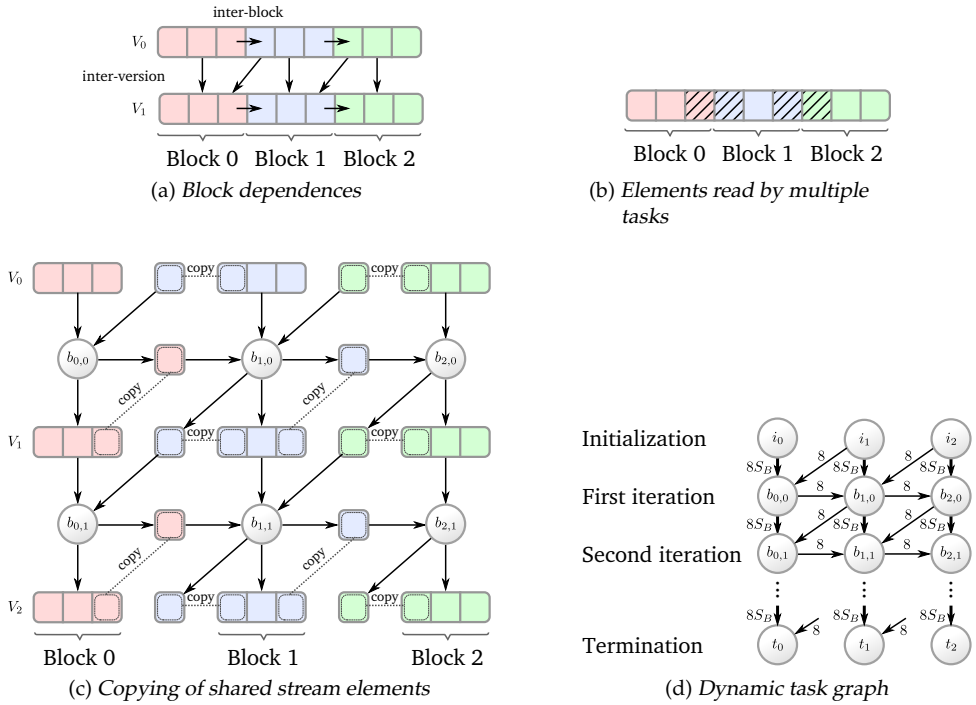
The new function `seidel_1d_seq_blocked` implicitly divides the array in blocks of `B` elements and treats each block individually during each iteration of the algorithm. The leftmost and the rightmost block must be treated differently from the others due to the missing neighboring elements on the left and the right, respectively. This separated treatment is done in lines 5 to 9 and 17 to 21. The remaining blocks are processed by the loop nest in lines 12 to 14.

5.3.2 Mapping of data elements to stream elements and definition of the interface of tasks generating new versions

The next step towards an implementation using dynamic single assignment is to develop a mapping between the versions of the data elements and stream elements based on the partitioning established before. If possible, this mapping should take advantage of the layout of stream data at execution time, e.g., using subsequent stream indexes for data elements that are processed sequentially. For our example this means that the elements of a block should be mapped to a set of contiguous stream indexes of the same stream. To avoid complex synchronization patterns in the parallelized control program regarding the matching of views on a stream, each stream is used only once, i.e., by a single producer and a single consumer (cf. Section 5.5).

After the determination of the mapping, the interface of the tasks can be defined according to the data dependences. A task that processes a block of data does not only depend on the values of the block itself from the previous iteration, but also on elements of the neighboring blocks. In addition, the number of neighbors of a block depends on the position of the block within the array. For example, a block in the center of the array, i.e., not at the rightmost or leftmost position, depends on the values of its left neighbor from the current iteration, its own values from the previous iteration and on the values of the right neighbor also from the previous iteration. For the blocks on the left or right of the array, there is one dependence less due to the missing neighbor either at the left or the right. Figure 5.1a illustrates these inter-block and inter-version dependences for an array with three blocks.

From a block's perspective, the data of a version of the block is read by up to two tasks. The striped elements in Figure 5.1b indicate elements that are read by two tasks, while the other elements are only read by the task of the same block at the next iteration. Hence, for one version



(a) *Block dependences*

(b) *Elements read by multiple tasks*

(c) *Copying of shared stream elements*

(d) *Dynamic task graph*

Figure 5.1: *Dependences in the dynamic single assignment version of* seidel-1d

of a block, there are multiple readers and multiple views on the same stream indexes would be required to provide the readers with access to the shared data. This pattern of communication can be implemented either through a broadcast, as described in Section 3.3, or by emitting shared data manually on multiple streams. As the number of readers is fixed for each data element and known at compile time and as optimizations of broadcasts are described after the optimizations for ordinary input and output views, we chose to emit data manually on multiple streams. In addition, each stream is only used to synchronize exactly two tasks in order to facilitate the creation of a parallel control program as explained in Section 5.5.5.

Figure 5.1c shows this principle for the first two iterations and three blocks. Each arrow from a task to a rectangle represents a write access to the elements of a unique stream and each arrow starting at a rectangle and ending in a task represents a read access on the same stream. Data that is read by more than one task is simply written twice to different streams, indicated by the dotted line labeled *copy*.

In summary, for each iteration, there is a set of $\frac{N}{B}$ streams used for dependences between tasks processing the same block, $\frac{N}{S_B} - 1$ streams for inter-block dependences within the same iteration and $\frac{N}{S_B} - 1$ streams for inter-block, inter-version dependences with N being the number of elements in the data array and S_B being the size of a block.

5.3.3 Definition of auxiliary tasks needed for initialization and termination

To replace the sequential algorithm of Listing 5.6, the implementation using dynamic single assignment must use exactly the same interface. However, as the initial version of the data elements is provided in a shared array and not within streams, there must be a set of initial tasks that copy data from shared memory to the streams. Likewise, the values of the final versions must be copied from streams to the shared array. The set of auxiliary tasks is thus composed of initialization tasks that copy data to the streams and termination tasks that write the results back to shared memory.

5.3.4 Implementation of all tasks

The following listing shows the dynamic single assignment implementation with a sequential control program., including main computation tasks and auxiliary tasks copying data from shared memory to streams and from streams to shared memory.

Listing 5.8: Parallel, dynamic single assignment implementation of *seidel-1d*

```
1  enum block_position {
2    POS_CENTER,
3    POS_LEFT,
4    POS_RIGHT
5  };
6
7  /* Update the values of one block according to the block's
8     position in the array. */
9  void process_block(enum block_position pos, size_t B,
10           double* center_in, double* center_out,
11           double* left_in, double* left_out,
12           double* right_in, double* right_out)
13 {
14   double vleft_in = 0.0;
15   double vright_in = 0.0;
16
17   /* Left neighbor? */
18   if(pos != POS_LEFT)
19     vleft_in = *left_in;
20
21   /* Right neighbor? */
22   if(pos != POS_RIGHT)
23     vright_in = *right_in;
24
25   /* Update first element of the block */
26   center_out[0] = (vleft_in + center_in[0] + center_in[1]) / 3.0;
27
28   /* Update elements in the middle of the block that
29      only depend on the block's own elements */
30   for(int i = 1; i < B-1; i++)
31     center_out[i] = (center_out[i-1] + center_in[i] + center_in[i+1]) / 3.0;
32
33   /* Update last element of the block */
34   center_out[B-1] = (center_in[B-2] + center_in[B-1] + vright_in) / 3.0;
35
36   /* Communicate the first value of the block to the left */
37   if(pos != POS_LEFT)
38     *left_out = center_out[0];
39
40   /* Communicate the last value of the block to the right */
41   if(pos != POS_RIGHT)
42     *right_out = center_out[B-1];
43 }
44
45 void seidel_1d_dsa(double* data, size_t N, size_t B, int num_iter)
46 {
47   size_t num_blocks = N/B;
48
49   /* Streams storing the values generated for each version */
50   double scenter[(num_iter+2)*num_blocks] __attribute__((stream));
51   double sleft[(num_iter+2)*num_blocks] __attribute__((stream));
52   double sright[(num_iter+2)*num_blocks] __attribute__((stream));
53
54   /* Indexes in the array of streams for input dependences */
55   #define LEFT_IN_IDX ((iter+1)*num_blocks+block-1)
56   #define RIGHT_IN_IDX (iter*num_blocks+block+1)
57   #define CENTER_IN_IDX (iter*num_blocks+block)
58
59   /* Indexes in the array of streams for output dependences */
60   #define LEFT_OUT_IDX ((iter+1)*num_blocks+block)
61   #define RIGHT_OUT_IDX ((iter+1)*num_blocks+block)
62   #define CENTER_OUT_IDX ((iter+1)*num_blocks+block)
63
64   /* Views on the stream elements */
65   double left_in, right_in, center_in[B];
66   double left_out, right_out, center_out[B];
67
68   /* Create tasks copying the initial version to the streams */
69   for(size_t block = 0; block < num_blocks; block++) {
70     /* Leftmost block */
```

```
71      if(block == 0) {
72        #pragma omp task output(scenter[block] << center_out[B])
73        {
74          memcpy(center_out, &data[B*block], B*sizeof(double));
75        }
76      }
77      /* Other blocks */
78      else {
79        #pragma omp task output(scenter[block] << center_out[B], \
80              sleft[block] << left_out)
81        {
82          memcpy(center_out, &data[B*block], B*sizeof(double));
83          left_out = data[B*block];
84        }
85      }
86    }
87
88    /* Create one task for each block and each iteration */
89    for(size_t iter = 0; iter < num_iter; iter++) {
90      for(size_t block = 0; block < num_blocks; block++) {
91        /* Leftmost block */
92        if(block == 0) {
93          #pragma omp task \
94            input(scenter[CENTER_IN_IDX] >> center_in[B], \
95                  sleft[RIGHT_IN_IDX] >> right_in) \
96            output(sright[RIGHT_OUT_IDX] << right_out, \
97                   scenter[CENTER_OUT_IDX] << center_out[B])
98          {
99            process_block(POS_LEFT, B,
100                   center_in, center_out,
101                   NULL, NULL,
102                   &right_in, &right_out);
103         }
104       }
105       /* Rightmost block */
106       else if(block == num_blocks-1) {
107         #pragma omp task \
108           input(scenter[CENTER_IN_IDX] >> center_in[B], \
109                 sright[LEFT_IN_IDX] >> left_in) \
110           output(sleft[LEFT_OUT_IDX] << left_out, \
111                  scenter[CENTER_OUT_IDX] << center_out[B])
112         {
113           process_block(POS_RIGHT, B,
114                   center_in, center_out,
115                   &left_in, &left_out,
116                   NULL, NULL);
117         }
118       }
119       /* Block in the center */
120       else {
121         #pragma omp task \
122           input(scenter[CENTER_IN_IDX] >> center_in[B], \
123                 sright[LEFT_IN_IDX] >> left_in, \
124                 sleft[RIGHT_IN_IDX] >> right_in) \
125           output(sright[RIGHT_OUT_IDX] << right_out, \
126                  sleft[LEFT_OUT_IDX] << left_out, \
127                  scenter[CENTER_OUT_IDX] << center_out[B])
128         {
129           process_block(POS_CENTER, B,
130                   center_in, center_out,
131                   &left_in, &left_out,
132                   &right_in, &right_out);
133         }
134       }
135     }
136   }
137
138   /* Create tasks copying the final version back to shared memory */
139   for(size_t block = 0; block < num_blocks; block++) {
140     /* Leftmost block */
141     if(block == 0) {
142       #pragma omp task input(scenter[num_iter*num_blocks+block] >> center_in[B])
143       {
144         memcpy(&data[B*block], center_in, B*sizeof(double));
145       }
146     }
147     /* Other blocks */
148     else {
149       #pragma omp task \
150         input(sleft[num_iter*num_blocks+block] >> left_in, \
```

```
151                 scenter[num_iter*num_blocks+block] >> center_in[B])
152         {
153           memcpy(&data[B*block], center_in, B*sizeof(double));
154         }
155       }
156     }
157
158     #pragma omp taskwait
159   }
```

The listing starts with the definition of `process_block` in lines 9 to 43, which performs one iteration of the stencil on a single block. Values from the previous iteration as well as values received from neighbors are passed as pointers to the respective data regions. The position of the block is indicated by a value from an enumeration, such that `process_block` can carry out the necessary steps depending on the block's position.

Lines 50, 51 and 52 define arrays of streams with one stream for each dependence of a block, each block and each iteration. The stream that is used to exchange data between two tasks is determined through proper indexation of these arrays. For example, two tasks processing the jth block at iterations k and $k+1$, the data generated by the first task is passed through the $((k+1)\cdot B+j)$th stream of `scenter`. Indexation of the other arrays of streams is done in a similar way. The preprocessor definitions of lines 55 to 62 serve as macros that facilitate the indexation of the arrays of streams in the input and output clauses of the main tasks, based on the iteration (`iter`) and the block identifier (`block`).

The initial tasks that copy data from shared memory to streams are created by the loop in lines 69 to 86. Depending on the position of a block, the initial data only needs to be copied to one stream or to two streams by the same task. The actual copying is carried out by a simple call to `memcpy` with the target address corresponding to the base address of the appropriate view. All main computation tasks are created by the loops in lines 89 to 136. Again, depending on the position of the block, the interface of the associated task varies. Also, the task body is adapted to the position and passes appropriate values to `process_block`. The terminal tasks are created by the loop in lines 139 to 156.

The resulting task graph is shown in Figure 5.1d. The weights of edges between tasks processing the same block at two different interations is $8S_B$ and corresponds to the size of a double precision floating point value multiplied with the number of elements per block. All other weights correspond to dependences of a single double precision floating point value and have a weight of only eight.

5.3.5 Parallelization of the control program

In the last step, the control program is parallelized. This step is necessary since it reduces the memory footprint of the application and increases parallelism, as discussed in Section 5.4. Depending on the complexity of the structure of the task graph, this step can be more or less complicated. For algorithms with regularly structured task graphs, e.g., with a same set of tasks that is instantiated at multiple iterations as in the example, the principles for parallelization are simple: each task creates its indirect successor generating the version after next of the same data elements, as illustrated in Figure 5.2. The initial tasks (i_0 to i_2), as well as the tasks treating the blocks for the first iteration ($b_{0,0}$ to $b_{2,0}$), are created directly by the root task. Afterwards, each task creates its indirect successor on the same block, i.e., a task i_j creates $b_{j,1}$, $b_{j,k}$ creates $b_{j,k+2}$ and so on, until $b_{j,n-2}$ creates t_j, with n being the number of iterations. The end of task creation is reached before the last iteration and neither $b_{j,n-1}$ nor t_j create follow-up tasks.

As the root task does not create all of the tasks, the taskwait construct at the end of `seidel_1d_dsa` in Line 158 of Listing 5.8 does not synchronize the root task with all tasks anymore. Therefore, the task graph contains an additional task d that is created by the root task and that reads a single integer from each of the terminal tasks. When d is ready for execution, all other tasks besides the root task have terminated. By synchronizing with d using a taskwait construct, the root task can thus synchronize indirectly with all tasks of the task graph.

Due to the size of the code, we do not show the entire listing of the implementation with a

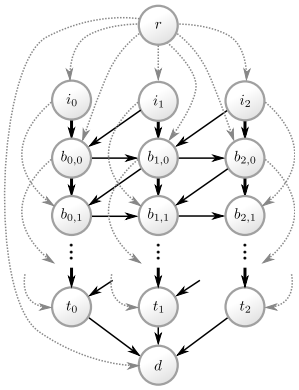


Figure 5.2: *Parallel control program of* seidel-1d

parallel control program and only sketch the actual code. In a first step, each task creation in Listing 5.8 is moved to a separate function. Next, a function named `create_followup_task` is defined and called from the body of each task. The parameters of `create_followup_task` describe the exact instance of the task from which the function was called, i.e., the task type (initialization or computation task), the block number and the iteration. Depending on the values of these parameters, the function determines whether a follow-up task needs to be created and, if this is the case, calls the appropriate function for task creation with the correct parameters. The following listing shows illustrates this principle for the tasks processing blocks in the center of the array:

Listing 5.9: Sketch of *seidel-1d* with a parallel control program

```
1  ...
2  enum task_type {
3    INIT_TASK,
4    MAIN_TASK
5  };
6
7  ...
8
9  void create_followup_task(enum task_type caller_type, int iter, int block,
10                           double* data, size_t N, size_t B, int num_iter)
11 {
12   if(caller_type == MAIN_TASK) {
13     /* Does this task have an indirect succesor of the same type? */
14     if(iter+2 < num_iter)
15       create_main_task(data, N, B, num_iter, iter+2, block);
16     /* If not, does it have an indirect successor that is a terminal task? */
17     else if(iter == num_iter-2)
18       create_terminal_task(data, N, B, block);
19
20     /* Otherwise: Nothing to do, task creation stopped */
21   } else if(caller_type == INIT_TASK) {
22     /* Regular case: the indirect successor is a main task */
23     if(num_iter > 1)
24       create_main_task(data, N, B, num_iter, 1, block);
25     /* For a single iteration the indirect successor is a terminal task */
26     else
27       create_terminal_task(data, N, B, block);
28   }
29 }
30
31 void create_init_task(double* data, size_t N, size_t B, int num_iter, int block)
32 {
33   ...
34 }
35
36 void create_terminal_task(double* data, size_t N, size_t B, int num_iter, int block)
37 {
38   ...
39 }
40
41 void create_main_task(double* data, size_t N, size_t B, int num_iter,
```

```
42                          int iter, int block)
43  {
44    int num_blocks = N/B;
45    ...
46
47    /* Leftmost block */
48    if(block == 0) {
49      ...
50    }/* Rightmost block */
51    else if(block == num_blocks-1) {
52      ...
53    }
54    /* Block in the center */
55    else {
56      #pragma omp task \
57        input(scenter_ref[CENTER_IN_IDX] >> center_in[B], \
58              sright_ref[LEFT_IN_IDX] >> left_in, \
59              sleft_ref[RIGHT_IN_IDX] >> right_in) \
60        output(sright_ref[RIGHT_OUT_IDX] << right_out, \
61               sleft_ref[LEFT_OUT_IDX] << left_out, \
62               scenter_ref[CENTER_OUT_IDX] << center_out[B])
63      {
64        process_block(POS_CENTER, B,
65                center_in, center_out,
66                &left_in, &left_out,
67                &right_in, &right_out);
68        create_followup_task(ITER_TASK, iter, block, data, N, B, num_iter);
69      }
70    }
71  }
72
73  void seidel_1d_dsa_parctrl(double* data, size_t N, size_t B, int num_iter)
74  {
75    ...
76    int dfbarrier_tokens[blocks];
77    int sdfbarrier __atribute__((stream));
78    ...
79
80    /* Create tasks copying the initial version to the streams */
81    for(size_t block = 0; block < num_blocks; block++) {
82      create_init_task(data, N, B, num_iter, block);
83      create_main_task(data, N, B, num_iter, iter, block);
84    }
85
86    /* Task synchronizing with all terminal tasks */
87    #pragma omp task input(sdfbarrier >> dfbarrier_tokens[blocks])
88    {
89    }
90
91    #pragma omp taskwait
92  }
```

The enumeration defined in Lines 2 to 5 associates one constant for each type of tasks in the task graph that potentially creates another task: `INIT_TASK` refers to an initialization task and `MAIN_TASK` stands for a task that processes a block of data. These constants are referenced in `create_followup_task`, starting at Line 5 of the listing. The function contains several tests that help determine whether a follow-up task needs to be created and which kind of task this is. A main task can either create another main task for the iteration after the next iteration (Line 15) or a terminal task (Line 18). Which of these tasks must be created depends on the iteration of the calling task, tested in Lines 14 and 17. In most cases, the task created by an initial task is a main task of the second iteration, as in Line 24. However, if there is only a single iteration, the indirect successor of an initial task in the task graph is a terminal task (Line 27).

The functions `create_init_task`, `create_terminal_task` and `create_main_task` are responsible for the creation of initial tasks, terminal tasks and main tasks, respectively. As in the previous listing, there are three types of main tasks with different sets of input and output views: one for the leftmost block, one for blocks in the center and one for the block at the rightmost position. Listing 5.9 omits the code for the creation of main tasks processing blocks at the left or the right of the array and only details the task for blocks in the center in Lines 56 to 69. The first difference to the previous listing is the use of stream references rather than streams in the input and output clauses. Instead of indexing the arrays of streams `scenter`, `sright` and `sleft`, the clauses refer to arrays of stream references named `scenter_ref`, `sright_ref` and `sleft_ref`.

This is necessary due to the technical restriction that streams can only be declared in the local scope of a function and thus cannot be referenced directly from multiple functions. In the part not shown in the listing, references to the locally created streams are stored within global arrays of stream references that can be accessed from any function. The second difference with the previous listing consists in the additional function call to `create_followup_task` after the call to `process_block` in Line 68, which effectively implements the parallel control program. Note that this call must be issued from within the task body, otherwise the creation of subsequent tasks would still be carried out by the root task.

5.4 Implications of dynamic single assignment on the control program

As all data of dynamic single assignment tasks is stored in input buffers and as these buffers are managed by the run-time, memory allocation for most of the data used by an application based on dynamic single assignment is under the responsibility of the run-time system. While this has the advantage that the run-time can use optimized algorithms and data structures to manage these allocations, dynamic single assignment can have a significant impact on the application's memory footprint as well as on the locality of data accesses. If input buffers cannot be reused, e.g., if tasks are created rapidly, such that none of the tasks has terminated before an allocation takes place, multiple buffers with different versions of the same data elements are kept at the same time, even though not all of them might be referenced simultaneously. However, even for extensive reuse of buffers there is a minimal number of versions that must exist at the same time. For each data element for which multiple versions are produced throughout the execution of the application and for which each version depends on the previous version, there are at least two versions present in data-flow frames when a new version is being generated: one is the current version, whose values are written and the second one corresponds to the previous version that serves as a base to calculate the new values.

The memory footprint of an application depends on the maximum number of input buffers that co-exist and is influenced by different factors. In particular, these are (1) the structure of the control program (sequential or parallel) and (2) dependences between tasks and their order of creation. In this section, we examine both points by unrolling the steps involved in task creation, allocation and de-allocation according to the execution model of OpenStream for simple examples. We also emphasize how the reuse of input buffers influences the locality of accesses to main memory with respect to NUMA.

5.4.1 Allocations of a sequential control program

The input buffers of a task are allocated when the task is created and remain in use until it terminates. Whether these allocations increase the memory footprint of the application or not depends on the state of the free list of the memory pool from which the buffers are allocated. If an allocation requires a refill operation caused by an empty free list, the footprint increases, while the footprint remains the same if a buffer from an earlier refill operation can be reused. However, in order to be reused by a subsequent allocation, an input buffer must be freed to the same memory pool as the pool used for the subsequent allocation. In the following part, we illustrate that a sequential control program either inhibits reuse or leads to poor data locality due to the allocation and de-allocation mechanism used for NUMA-aware memory pooling.

A sequential control program causes all allocations of input buffers to be carried out by a single task, namely the root task. As this task is executed by a single worker, all allocations are made within the same memory pool. Let w_0 be the worker that executes the control program. The tasks created by w_0 eventually become ready and, if w_0 has not finished executing the root task, these tasks are stolen or activated and executed by other workers. Upon termination of a task, the worker that executed the task frees the task's input buffers to the memory pools associated to the nodes on which the input buffers are located. There are two main scenarios for the placement and thus

for the de-allocation and reuse of an input buffer. First, if the size of a page is smaller than the size of a buffer, buffer placement is determined by the initial writer of the buffer as explained in Section 4.2.2. Reuse of such an input buffer can only take place if the writer is located on the same node as w_0, since only in this case the buffer is freed to the memory pool used by w_0. However, due to the small number of cores per node compared to the total number of cores of a many-core system, it is more likely that the writer executes on a core of a different node and the buffer is never used again. Second, if the size of a page of memory is larger than or equal to the size of the input buffer, the buffer is allocated on the node of w_0 as physical allocation and thus data placement have already been triggered during the refill operation (cf. Section 4.2.1). In this case, the buffer is freed to the memory pool used by w_0 and can rapidly be reused by subsequent allocations. However, this leads to the use of buffers that are all placed on the node of w_0, which results in poor data locality and high contention on the respective memory controller.

The following examples illustrates the two cases for the reuse of buffers when using a sequential control program.

Large memory footprint resulting from sequential task creation

Figure 5.3 illustrates the first scenario on the task graph, shown in Figure 5.3a. Every single task $t_0, \ldots, t_7$ as well as $t_0^i, \ldots, t_7^i$ is created by the root task r. For simplicity, we assume that t_0^i to t_7^i are all created before any of the tasks t_0 to t_7 is created, such that a steal of a task t_j^i results in the execution of t_j by the same worker, since every task t_j only has a single dependence and is thus activated right after the execution of t_j^i. We also assume that w_0, w_1 and w_2 execute on different NUMA nodes and thus use different memory pools. Furthermore, the size of all input buffers in the example is identical. Note that t_0^i to t_7^i do not have predecessors in the task graph and thus do not have input buffers. The Figures 5.3b to 5.3r show the state of the free list associated to the size of the input buffers of the memory pool of each worker after each step explained below.

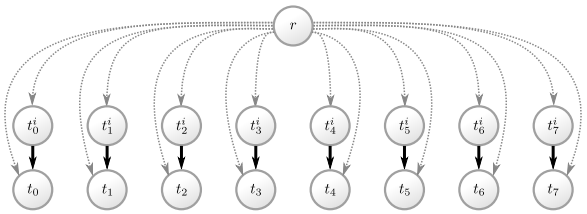
Initially, all lists are empty, as shown in Figure 5.3b. Let w_0 be the worker that executes the control program. Upon the creation of t_0, the memory pool needs to be refilled and new buffers are allocated. In the example, each refill operation allocates only two frames at once (cf. Figure 5.3c). The creation of t_0 activates the previously created task t_0^i due to the restriction that all consumers of a task must have been created before the task can execute. Let w_1 be the worker that steals t_0^i and which thus becomes the owner of t_0 after its execution, as shown in Figure 5.3d. Similarly, w_2 becomes the owner of t_1 after its creation and a steal of t_1^i (Figure 5.3e). When t_0 and t_1 terminate, their input buffers are freed to the memory pools of w_1 and w_2, respectively, as shown in Figure 5.3f. These buffers cannot be reused by w_0 for the creation of t_2 and another refill operation in the memory pool of w_0 becomes necessary (Figure 5.3g). The newly created tasks unblock t_2^i and t_3^i and cause w_1 and w_2 to execute t_2 and t_3 after the steals of t_2^i and t_3^i (Figures 5.3h and 5.3i). The input buffers are freed to the memory pools of w_1 and w_2 (Figure 5.3j), which makes them unavailable to w_0. This process repeats until the last two tasks terminate (Figure 5.3k to 5.3r).

In summary, w_0 cannot reuse any of the buffers as all tasks were stolen by workers from remote nodes. This results in the allocation of a total of eight input buffers by four refill operations.

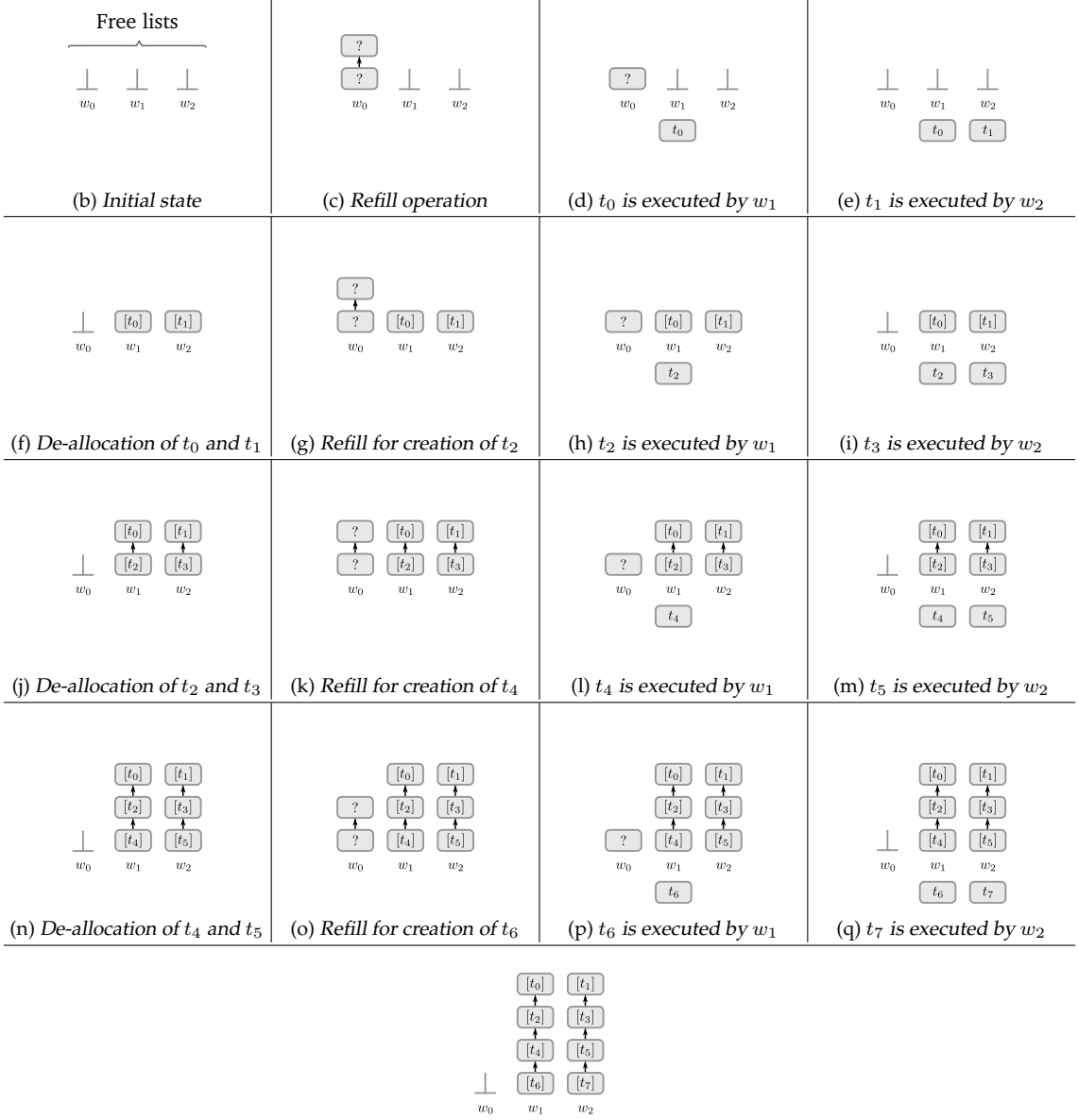
Extensive reuse with poor data locality and high contention resulting from sequential task creation

Figure 5.4 illustrates the events related to buffer allocation for the same application with huge pages causing input buffers to be placed on the allocating node. The first four steps, shown in Figure 5.4a to Figure 5.4d, are identical to the previous case using small pages. The first difference appears when the input buffers if t_0 and t_1 are freed, shown in Figure 5.4e. Instead of freeing them to the memory pools of w_1 and w_2, they are handed back to the memory pool of w_0. Hence, when t_2 is created, the free list of the memory pool of w_0 contains unused buffers and a refill operation is not necessary [1]. Figure 5.4f shows the state of the memory pools when t_2 is executed by w_1 after the steal of t_2^i by the same worker. Similarly, the buffer formerly used for t_0 can be reused for t_3 as

1. In practice, task creation is often faster than task execution, such that t_2 would likely be created before the input buffers of t_0 and t_1 are freed. However, the reuse of input buffers would take place for later task creations starting at the first de-allocation of an input buffer. To keep this example simple, we assume that buffers are reused immediately.



(a) *Task graph with sequential task creation*



(b) *Initial state*
(c) *Refill operation*
(d) t_0 *is executed by* w_1
(e) t_1 *is executed by* w_2
(f) *De-allocation of* t_0 *and* t_1
(g) *Refill for creation of* t_2
(h) t_2 *is executed by* w_1
(i) t_3 *is executed by* w_2
(j) *De-allocation of* t_2 *and* t_3
(k) *Refill for creation of* t_4
(l) t_4 *is executed by* w_1
(m) t_5 *is executed by* w_2
(n) *De-allocation of* t_4 *and* t_5
(o) *Refill for creation of* t_6
(p) t_6 *is executed by* w_1
(q) t_7 *is executed by* w_2
(r) *De-allocation of* t_6 *and* t_7

Figure 5.3: *Memory footprint resulting from sequential task creation with small pages*

shown in Figure 5.4g. Upon de-allocation of t_2 and t_3, the free list of the memory pool of w_0 again contains two buffers which can be reused for future tasks (Figure 5.4h). This pattern of allocations and de-allocations repeats as shown in Figure 5.4i to Figure 5.4n until all tasks of the task graph have been executed. In total, only two buffers have been allocated resulting in a smaller memory footprint compared to the previous example. However, as all buffers are placed on the node of w_0, data locality is poor, resulting in high contention on a single memory controller.

5.4.2 Allocations of a parallel control program

We now show that using a parallel control program leads to a different pattern of allocations and can both significantly reduce the memory footprint of the application and provide improved locality of accesses to main memory. To illustrate this, we use the same tasks as in the previous example, but replace the sequential control program with a parallel control program, in which each task t_j creates a follow-up task t_{j+2} and each task t_j^i creates t_{j+2}^i for $j \in \{0, \ldots, 5\}$. Only t_0^i, t_1^i, t_0 and t_1 are still created by the root task r, as illustrated by the task graph of Figure 5.5a. To keep the example simple, we assume that small pages are used, which causes all input buffers to be placed on the node of the workers that perform the first write access. The behavior using huge pages would be similar to the steps below, as only the placement of the input buffers of t_0 and t_1 is affected by the page size.

The steps presented in Figure 5.5b to 5.5d are identical to the steps with a sequential control program, since the root task still creates t_0 and t_1. The first difference appears at the execution of t_0 by w_1, when the follow-up task t_2 is allocated using the memory pool of w_1 instead of the pool of w_0. This triggers a refill operation and results in the addition of two input buffers to the free list of the memory pool used by w_1 (Figure 5.5e). The first element from the free list is used for t_2 and upon termination of t_0 the input buffer of t_0 is added at the front of the list, resulting in the state shown in Figure 5.5f. Similarly, w_2 performs a refill operation during execution of t_1 (Figure 5.5g), removes an input buffer for t_3 and pushes the old buffer of t_1 onto the free list (Figure 5.5h). Figure 5.5i shows how previously allocated buffers are reused: the old buffers of t_0 and t_1 are used as the input buffers for t_4 and t_5. Similarly, the old buffers of t_2 and t_3 are used for t_6 and t_7 in Figure 5.5j and the old buffers of t_4 and t_5 are added to the free list. When all tasks terminate, only 6 buffers have been allocated in total (Figure 5.5k).

Note that in contrast to the sequential control program with huge pages, the footprint increases, but data locality is similar to the sequential control program with small pages as w_1 and w_2 mostly operate on input buffers allocated from their own memory pools.

This example shows that work-stealing is an essential mechanism that spreads the execution of the parallelized control program over the machine and leads to task creations by other workers than the one executing the root task. This causes refill operations to be carried out on multiple memory pools and thus results in better distribution of the data across memory controllers and increases data locality. However, where an input buffer is allocated and whether an existing buffer can be reused varies with the total number of tasks and workers as well as on the timing of events at execution time as these have a strong influence on work-stealing. Hence, it is difficult to predict the exact memory footprint of an application only based on information about the task graph, the control program and the machine.

5.4.3 Estimation of the memory footprint

Although the exact memory footprint is difficult to predict, it is possible to provide upper and lower bounds for the number of buffers that are allocated throughout the execution for a given task graph. We illustrates this on the task graphs of Figure 5.3a and Figure 5.5a.

Let n_r be the number of buffers allocated by a single refill operation and let n_t be the number of tasks in a program with the same characteristics as the program of the previous examples. For a sequential control program and small pages, the total number of buffers $N_{\text{seq}}^{\text{small}}$ allocated by refills

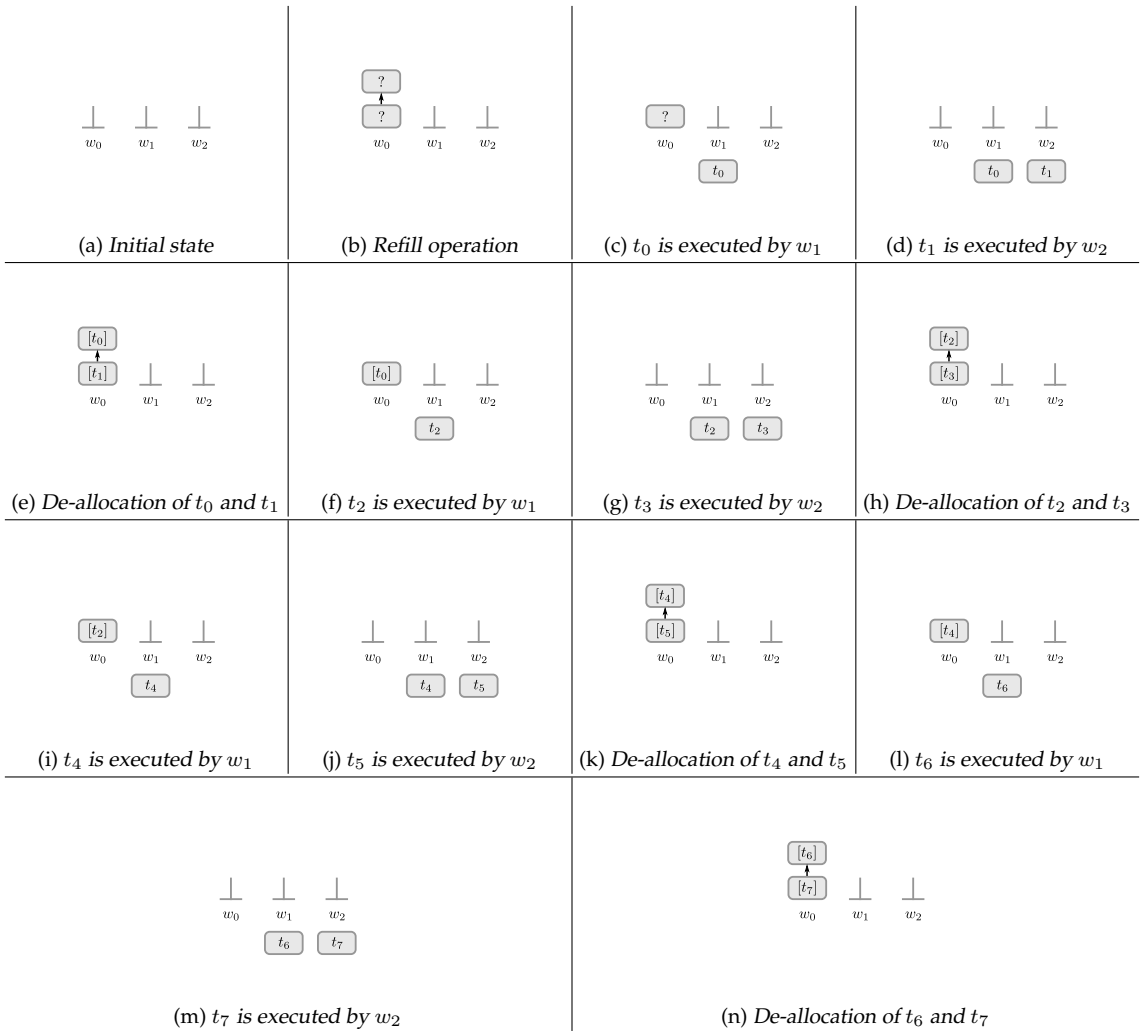
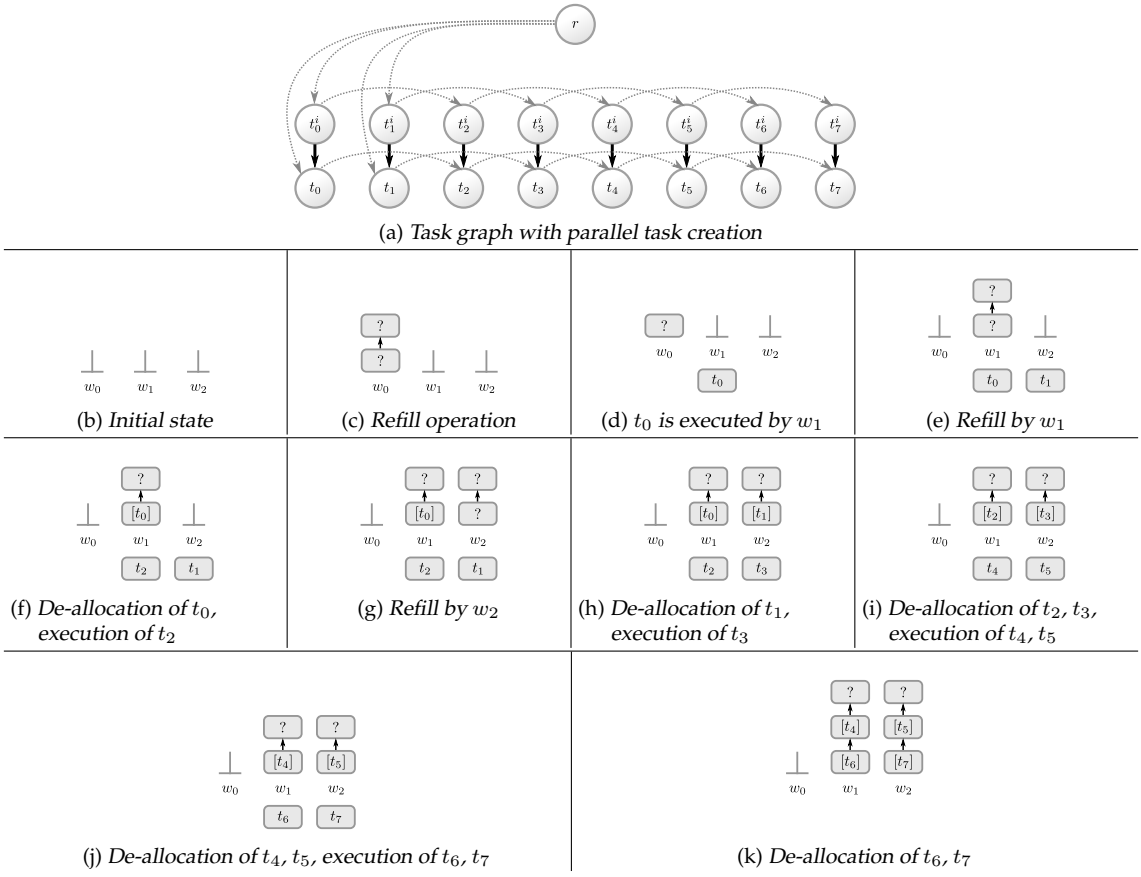


Figure 5.4: *Memory footprint resulting from sequential task creation with huge pages*



(a) *Task graph with parallel task creation*

(b) *Initial state*

(c) *Refill operation*

(d) t_0 *is executed by* w_1

(e) *Refill by* w_1

(f) *De-allocation of* t_0*, execution of* t_2

(g) *Refill by* w_2

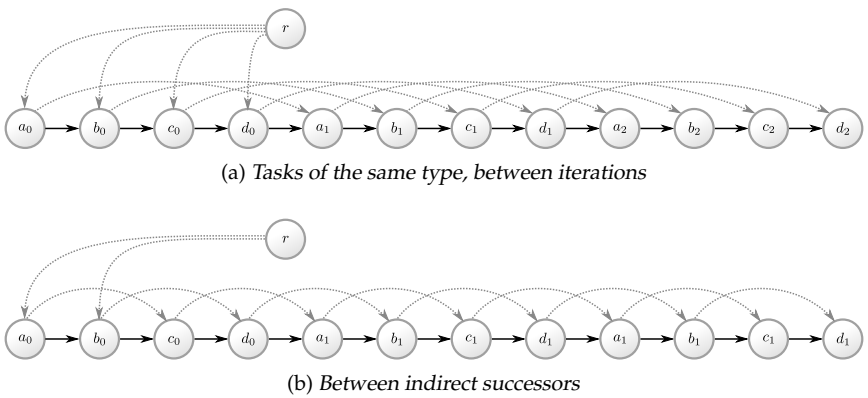
(h) *De-allocation of* t_1*, execution of* t_3

(i) *De-allocation of* t_2*,* t_3*, execution of* t_4*,* t_5

(j) *De-allocation of* t_4*,* t_5*, execution of* t_6*,* t_7

(k) *De-allocation of* t_6*,* t_7

Figure 5.5: *Memory footprint resulting from parallel task creation*



(a) *Tasks of the same type, between iterations*

(b) *Between indirect successors*

Figure 5.6: *Order of task creations in a parallel control program*

is constrained as follows:

$$n_r \leq N_{\text{seq}}^{\text{small}} \leq \left\lceil \frac{n_t}{n_r} \right\rceil \cdot n_r$$

The minimal number of allocations is achieved when all tasks created by the root task are stolen by workers operating on the same node as the worker executing the root task and if all buffers are freed in time right before an allocation, resulting in maximal reuse. The upper bound corresponds to a scenario where all tasks are stolen by workers that use a different memory pool, preventing any buffer from being reused at task creation. Note that it is more likely that the footprint reaches the upper bound, since the number of workers per node is generally much lower than the total number of workers. For sequential control programs and huge pages the bounds are identical, but can result from different situations:

$$n_r \leq N_{\text{seq}}^{\text{huge}} \leq \left\lceil \frac{n_t}{n_r} \right\rceil \cdot n_r$$

For the minimal footprint, it is no longer required that only workers of the same node steal tasks and it is sufficient that buffers are handed back to the memory pool of the creating worker in time. The maximal footprint occurs for the worst possible timing, where none of the buffers is freed before creation of the last task. In general, it is unlikely that the footprint reaches the maximum, since the duration of the root task is usually higher than the duration of a task, which makes it likely that buffers are reused.

For a parallel control program, the number of buffers allocated by refills varies with the number of parallel chains of task creation n_{chains}. To avoid limiting the parallelism of the application, this number should be equal or greater than the number of workers n_w. For $n_{\text{chains}} = n_w$ and small pages the number of buffers is:

$$n_r \leq N_{\text{par}}^{\text{small}} \leq \underbrace{\left\lceil \frac{n_{\text{chains}}}{n_r} \right\rceil \cdot n_r}_{\substack{\text{Sequential creation} \\ \text{of the heads of} \\ \text{each chain}}} + \underbrace{(n_w - 1) \cdot n_r}_{\substack{\text{One refill for} \\ \text{each remaining} \\ \text{worker}}} = \left(\left\lceil \frac{n_{\text{chains}}}{n_r} \right\rceil + n_w - 1 \right) \cdot n_r = \left(\left\lceil \frac{n_w}{n_r} \right\rceil + n_w - 1 \right) \cdot n_r$$

The minimal number of allocations is reached if (1) every chain is stolen by a worker of the same node as the worker executing the root task, (2) $n_r \geq 2$ and thus sufficient buffers for the stolen chain and the next task in the chain exist and (3) all tasks of the chain terminate before the head of the second chain started by the root task. The maximum number of buffers is allocated if every chain is stolen by a different worker and if all workers operate on different nodes. The upper and lower bounds for huge pages are identical:

$$n_r \leq N_{\text{par}}^{\text{huge}} \leq \left(\left\lceil \frac{n_w}{n_r} \right\rceil + n_w - 1 \right) \cdot n_r$$

Similar to the sequential control program with huge pages, the minimal number does not require that steals are carried out by workers of the same node as the worker executing the root task. This is due to the circumstance that every worker hands the buffers allocated by the root task back to the same memory pool. As far as the upper bound is concerned, it is reached whenever the last chain is created before any of the previous chains has terminated.

While the upper bounds for $N_{\text{seq}}^{\text{small}}$ and $N_{\text{seq}}^{\text{huge}}$ are constrained by the number of tasks, the upper bounds of $N_{\text{par}}^{\text{small}}$ and $N_{\text{par}}^{\text{huge}}$ are constrained by the number of workers. As the number of tasks is generally much higher than the number of workers, a parallel control program thus yields a lower memory footprint.

5.4.4 The order of task creations in a parallel control program

In the previous examples, we have neglected inter-task dependences and assumed that all tasks besides pairs of tasks formed by t_j and t_j^i for $0 \leq j \leq 7$ are independent. However, the dependence

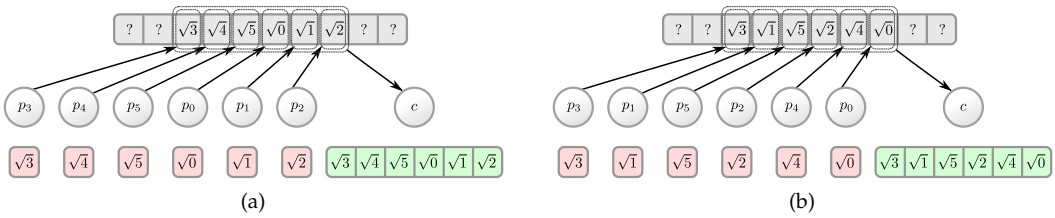


Figure 5.7: *Concurrent task creation with different matching of the views*

pattern in conjunction with the order of task creation determines how many tasks remain blocked. For example, in Figure 5.6, there are four task types a, b, c and d, which are instantiated for three iterations. In the task creation pattern of Figure 5.6a, each task creates its successor for the next iteration, skipping three tasks in between. The task of the next iteration can only become ready when all tasks in between have terminated. For example, a_2 can only start execution when b_1, c_1 and d_1 have terminated. During that time, the input buffers of b_2, c_2 and d_2 are allocated by b_1, c_1 and d_1, respectively. As the created tasks remain blocked, their input buffers are not available for reuse and the number of input buffers that coexist depends on the distance between two iterations.

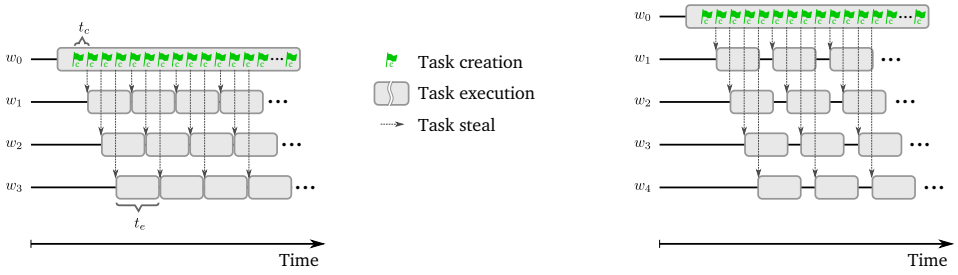
In Figure 5.6b the creation scheme is different. Instead of instantiating a task of the same type for the next iteration, each task allocates its indirect successor. As the distance between the creating task and the task that is being created is smaller than in the example before, the number of data-flow frames that coexist is lower, leading to a smaller memory footprint. Due to the restriction that the consumers of a task must be created before the task can start execution this distance cannot be reduced further.

Implementing a parallel control program in which all tasks create their indirect successors is often a lot more complicated than developing a pattern with creations between iterations. However, depending on the actual task graph the creation of tasks with the minimal distance can reduce the memory footprint significantly. A good example for such a program is the *bitonic* sorting network presented in Section 6.1.4, where the number of tasks per iteration increases with each iteration.

5.5 Parallelizing the control program

After the examples of parallel control programs and the discussion of the implications on data locality and the memory footprint we now discuss the restrictions that a parallel control program is subject to. In Section 3.1, we assumed that the control program creates all tasks sequentially. This limitation guarantees reproducible results for the mapping of views to stream elements, which ensures that the same producers are matched with the same consumers for each execution with deterministic results. If task creations and matchings of views would take place concurrently without any restriction, the set of stream elements a view provides access to could vary between two executions, depending on the exact timing. As a result, the order of values in a stream could vary from one run to another. For example, if the producers of Figure 3.5a on page 39 are created concurrently, the elements in the input buffer of the consumer are not necessarily stored in ascending order as specified for the sequential control program and can be shuffled, such as in Figure 5.7a and 5.7b.

In this section, we first point out the performance drawbacks not related to data locality or the memory footprint resulting from sequential task creation and sketch how a parallel control program increases performance. We then define the conditions under which parallel task creation preserves deterministic mappings of views to stream elements and sketch how a parallel control program can be derived from a sequential one.



(a) *Four workers* (b) *Five workers*

Figure 5.8: *Sequential control program with a different number of workers*

5.5.1 Rate of task creation

In the execution model of OpenStream all workers are created at the very beginning of program execution. When all workers are ready, one of them starts execution of the root task. In case of a sequential control program the control program is part of the root task and is thus executed by the same worker. The remaining workers are initially idle and try to obtain tasks through work-stealing. Thus, it is very likely that a newly created task that has become ready is immediately stolen and executed by a worker in parallel with the execution of the control program. Ideally, ready tasks are provided as fast as possible after the start of the control program, such that idle time of the remaining workers on startup is minimized and the arrival rate of these tasks is sufficiently high to provide enough tasks for execution afterwards.

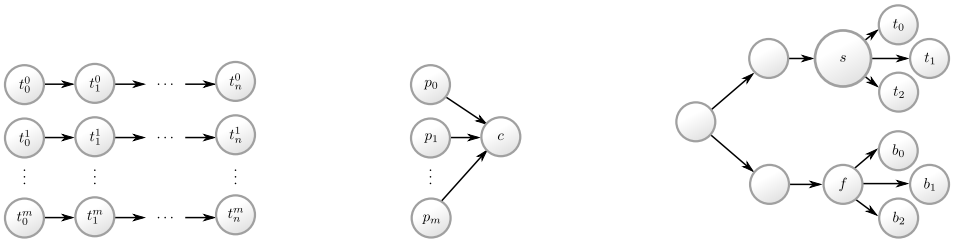
Let t_c be average time that is necessary to create a new task, i.e., the duration that it takes the run-time to set up its data structures and to perform calls to `resolve_dependences` for each of its views. Let t_e be the average time for the execution of a task and let t_r be the time on average between the moment when a task has been created and the moment when it becomes ready. If a task has neither input dependences nor output dependences, t_r is zero and the task becomes ready immediately after its creation. Let N denote the number of workers and, as a matter of simplicity, assume that all tasks are entirely independent. As long as the control program creates tasks at a higher rate than the remaining $N - 1$ workers execute them, the fact that the control program is sequential does not have an impact on the performance of the application. However, if tasks are executed faster than the rate of creation, workers become idle after the execution of a task, resulting in under-utilization of the hardware resources.

In the initial and terminal phase of an application at the beginning and at the end of the control program, only a subset of workers are busy. If these phases are neglected, the creation rate can be considered sufficiently high if the inequation $\frac{t_e}{t_c} > N - 1$ holds. If the inequation does not hold, there are idle phases between task executions and the creation rate is too low. Figure 5.8 illustrates these situations. In Figure 5.8a, the control program creates tasks fast enough, such that none of the four workers becomes idle. However, as can be seen in Figure 5.8b with the same values for t_c and t_e, the rate of creation is too low to keep an additional worker busy and idle phases occur.

For a huge number of workers, sequential task creation can even dominate the execution time. Let M be the number of tasks. The time t_{seq} needed for sequential execution of the entire program is:

$$t_{\text{seq}} = M \cdot t_c + M \cdot t_e = M \cdot (t_c + t_e)$$

For parallel execution, the longest sequential part is either the control program or the duration to execute the tasks on the critical path. Let t_{max} be the duration of the slowest task. For independent tasks and a sufficiently high number of processors, the minimal execution time is $\max\{M \cdot t_c, t_{\text{max}}\}$ according to Amdahl's Law. For a large number of workers and a large number of tasks, it is thus likely that sequential task creation dominates the execution time of the parallel program.



(a) *Multiple chains* (b) *Tree-like structure* (c) *Different durations*

Figure 5.9: *Examples of task graphs for which the order of task creation has an influence on performance*

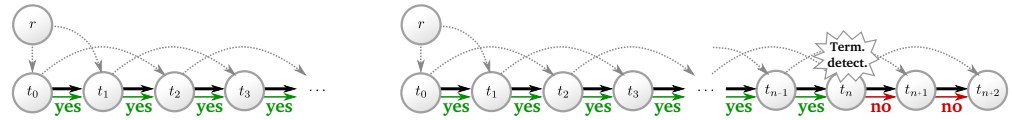
5.5.2 Order of task creations

In the discussion above, we assumed that tasks are completely independent and can therefore start execution immediately after creation. While this assumption is suited to illustrate the relationship between the task creation rate and performance, it is unrealistic for real-world applications as these usually have more complicated task graphs. Besides a few initialization tasks copying data from shared memory to streams at the beginning of the execution, the tasks of all the benchmarks presented in the next chapter have at least one input dependence. The order in which these tasks are created defines how fast they can become ready and is therefore crucial for performance. There are two major issues that should be taken into consideration when the order of task creation by the control program is determined.

First, the structure of the task graph is important and should be taken into account. Tasks with shorter paths from a task whose dependences have already been satisfied are good candidates for creation, as they are likely to become ready sooner than others. For example, the tasks t_i^j with $i \in \{0, \ldots, n\}$ and $j \in \{0, \ldots, m\}$ in Figure 5.9a with chain-like dependences should be created column-wise from left to right and not from right to left or row-wise. However, it must be taken into account that output dependences also have an effect on the readiness of a task. As output data is written to the consumers' input buffers, a producer cannot start execution before the creation of its dependent tasks. Hence, strict column-wise creation of the tasks in the example, i.e., creating t_i^0 to t_i^m before t_{i+1}^0 to t_{i+1}^m, delays the execution at the beginning, since t_0^0 is only ready upon creation of t_1^0 with $m-1$ task creations in between. To unblock tasks more rapidly, it would be preferable to start by first creating pairs of tasks t_0^j and t_0^{j+1} and to proceed strictly column-wise afterwards. Figure 5.9b shows an example of a task graph in which the creation of a consumer is required to unblock multiple producer tasks. None of the producers p_0 to p_m can execute before c has been created. Hence, instead of creating p_0 to p_m before c, the control program should create c first, such that an additional producer becomes ready at each subsequent task creation.

The second issue we would like to discuss is related to the duration of each individual task. The length of a path in the task graph to a task that is ready for execution does not necessarily reflect the duration until activation. For example, in Figure 5.9c, the task labeled f executes faster than the task labeled s, indicated by the size of the tasks in the figure. Thus, creating b_0, b_1 and b_2 before t_0, t_1 and t_2 unblocks tasks faster than the other way around.

Creating a sequential control program with optimal order with respect to the task graph that is to be constructed is often a non-trivial task. In addition, how fast an application executes tasks and how fast it makes progress within the task graph can be difficult or even impossible to predict. Parallelizing the control program, such that tasks are able to create their indirect successors in the task graph can thus be advantageous, as task creation and task execution progress together.



(a) *Start of the execution with an unknown number of tasks*

(b) *Termination*

Figure 5.10: *Parallel control program with termination detection*

5.5.3 Dynamic dependence patterns and termination detection

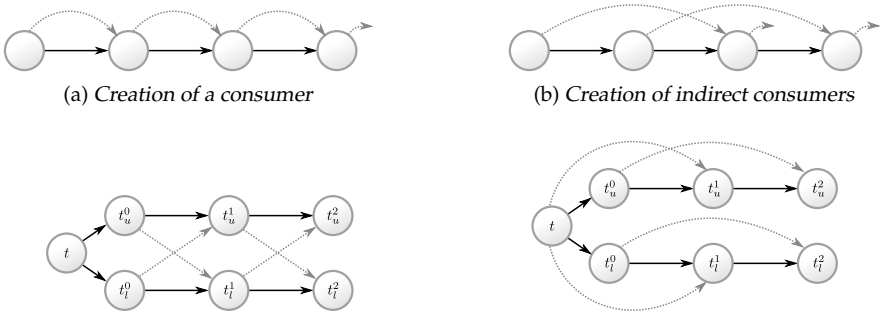
In some cases, it is even impossible to implement an application with a strictly sequential control program that does not synchronize with the tasks it creates using a taskwait barrier. For example, if the number of tasks of the program is finite, but unknown at the beginning of the execution, the control program cannot determine when task creation should stop. An example of such an application is the *k-means* benchmark presented in Section 6.1.6, whose control program must create a certain number of tasks for each iteration of the algorithm. The number of iterations, however, depends on the actual input data and is only known upon termination detection that takes place in the course of the execution. To stop the creation of tasks for future iterations, the task that detects the termination of the algorithm must synchronize with the control program. However, this cannot be done by passing data to the root task through a stream, since input data of a task can only be provided before a task is executed.

A parallel control program, in which tasks create their indirect successors is able to stop task creation based on the information that is available during execution. Figure 5.10 illustrates this concept on a simple task graph composed by a chain of tasks. In addition to a producer-consumer relationship between t_i and t_{i+1} for the actual data the graph also contains control dependences whose data indicates whether task creation should be stopped or if it should continue. At the beginning of the execution shown in Figure 5.10a the total number of tasks of the chain is unknown, but each task is capable of detecting whether another task is needed. Initially, the root task creates t_0 and t_1 and t_0 becomes ready for execution. Each task that does not detect that the application should terminate indicates to its successor that task creation should continue, as indicated by the edges labeled *yes*. Eventually, one of the tasks detects that the algorithm has finished. Let this task be t_n shown in Figure 5.10b. As the successor of t_n, t_{n+1}, was created before the execution of t_n, the chain of tasks cannot end with t_n. In addition, t_n cannot stop task creation neither, since t_{n+1} can only execute when its successor has been created due to the task's output dependence. Hence, an additional task t_{n+2} must be created whose input view matches the output view of t_{n+1}. If this task was not created, the application would deadlock and the program would not terminate. The task t_{n+2} forms the end of the chain and therefore does not have an output dependence. When t_{n+1} is executed, it first checks the value received through the control dependence and detects that task creation has stopped. This prevents an additional task t_{n+3} from being created and leads to proper termination.

5.5.4 Conditions for the parallelization of the control program

As discussed above, using a parallel control program can have a positive impact on performance as well as on the memory footprint of the application. However, as shown at the beginning of this section, parallelization of the control program can lead to indeterministic behavior. In the following part, we show that determinism can be preserved if the parallelization verifies certain restrictions.

The condition for deterministic behavior of a parallel control program is that for all possible executions, the views of each task provide access to the same streams and the same elements. Starting from a sequential control program, this means that the parallel control program must yield the exact same matchings as the sequential control program for each possible execution.



(a) *Creation of a consumer* (b) *Creation of indirect consumers*

(c) *Crossed creation* (d) *Non-crossed creation*

Figure 5.11: *Deadlocking and non-deadlocking parallel task creation*

Let $\tau_C, \tau_M, \tau_R : T_\infty \to \mathbb{N}_0$ be functions that indicate when a task is created (τ_C), when all of a task's views have been matched to a set of stream indexes (τ_M) and when a task becomes ready (τ_R). Due to the order of these events in the life cycle of a task, the inequation $t \in T_\infty$: $\tau_C(t) < \tau_M(t) < \tau_R(t)$ holds for all tasks $t \in T_\infty$. Note that $\tau_M(t)$ and $\tau_R(t)$ are not necessarily the same. For example, if a task has an output view on a stream and the consumer on this stream has not been created yet, the indexes of the elements that are accessible by the view are known upon the call to `resolve_dependences`, but the task only becomes ready when the input view of the consumer is matched.

Let further $T_{\text{seq}} = \langle t_0, t_1, t_2, \ldots \rangle$ be the totally ordered set of tasks created by a sequential control program with $\tau_C(t_i) < \tau_C(t_{i+1})$. To keep the definitions simple, we assume that a task can reference a stream at most in one of its views. Let $\tau_S : T_\infty \times \mathcal{S} \to \mathbb{N}_0$ be a partial function that indicates when a task's view accessing a stream has been matched. For each stream $s \in \mathcal{S}$ there are two totally ordered sets of tasks $T^s_{\text{seq},R} = \langle t_1^R, t_2^R, \ldots \rangle$ and $T^s_{\text{seq},W} = \langle t_1^W, t_2^W, \ldots \rangle$ with $\tau_S(t_j^R, s) < \tau_S(t_{j+1}^R, s)$ and $\tau_S(t_k^W, s) < \tau_S(t_{k+1}^W, s)$. These sets can be obtained by selecting in order only the tasks from T_{seq} that read from or write to s. We define that T_{seq} and one of its permutations T'_{seq} are equivalent if for all streams s the ordered sets $T^s_{\text{seq},R}$ and ${T^s_{\text{seq},R}}'$ as well as $T^s_{\text{seq},W}$ and ${T^s_{\text{seq},W}}'$ are identical.

Let $\mathcal{T}_{\text{par}} = \{p_1 = \langle t_1^1, t_2^1, \ldots \rangle, p_2 = \langle t_1^2, t_2^2, \ldots \rangle, \ldots\}$ be the set of all possible orders of task creations that can result from the execution of a parallel control program. If for all $T_{\text{par}} \in \mathcal{T}_{\text{par}}$ and $s \in S$ the equations $T^s_{\text{seq},R} = T^s_{\text{par},R}$ and $T^s_{\text{seq},W} = T^s_{\text{par},W}$ hold, then the parallel control program is equivalent to the sequential control program. As all matchings are identical to those of the sequential control program, deterministic execution is thus preserved.

5.5.5 Sketching deterministic parallel task creation

The development of a method for the construction of a parallel control program from a sequential control program is beyond the scope of this thesis. Hence, we only provide a sketch of how we have parallelized the control programs of most of the applications used in the experimental evaluation of this thesis and leave the development of a general method as a perspective for future work. In order to make concurrent matchings of views on the same stream impossible, each stream is used to synchronize only two tasks: one task takes the role of the producer on the stream and the other task is the consumer. For any order of calls to `resolve_dependences` of the two views on a stream, i.e., calling `resolve_dependences` for the input view before calling the functions for the output view or vice-versa, the input and output view provide access to the same set of stream elements. Using the definitions above, this results in $\forall s \in \mathcal{S} : |T^s_{\text{par},R}| = |T^s_{\text{par},W}| = 1 \vee |T^s_{\text{par},R}| = |T^s_{\text{par},W}| = 0$. This makes it impossible that the order of values of the elements of a stream varies between executions. This also implies, that the order of creation of these tasks does not have any influence on the matching on the streams, which finally

facilitates the development of a parallel control program.

However, although the order of calls to `resolve_dependences` can be arbitrary while preserving the order of stream elements, the task creation relationships, i.e., which task creates another task, is constrained. In particular, the control program must ensure that there are no deadlocks resulting from the restriction that a task can only become ready when all of its consumers have been created. Figure 5.11a and 5.11c illustrate task graphs and parallel control programs that result in a deadlock due to this restriction. In the first case, shown in Figure 5.11a, each task creates its direct consumer. However, the creation of a consumer takes place when the producer executes, but this requires that the producer has become ready, which in turn requires that the consumer has already been created. This phenomenon is not limited to direct producer-consumer relationships. For example, in the task graph and control program of Figure 5.11c there are no task creations between direct successors in the task graph, but the structure of the dependences still leads to a deadlock. The task t_u^0 requires that t_u^1 has been created and creates t_l^1. However, t_u^1 is created by t_l^0, which in turn requires that t_l^1 has been created. The solution for the problems shown in the figures is given in Figure 5.11b and 5.11d. In Figure 5.11b, each task creates its indirect successor in the task graph, as seen for *seidel-1d*. In Figure 5.11d, the creation of t_u^0 and t_l^0 is now done by t. Furthermore, t_u^0 and t_l^0 create their indirect successors.

These examples illustrate that the dynamic task graph resulting from the sequential control program must be analyzed carefully in order to avoid deadlocks when creating tasks with output dependences. Hence, providing a generic method for the construction of a parallel control program is a non-trivial task.

5.6 Summary

In this chapter, we introduced dynamic single assignment, which allows the run-time to determine the working set of a task and allows it to control where the data accessed by a task is placed through the allocating of input buffers. We provided an informal methodology for the implementation of programs based on dynamic single assignment and illustrated this methodology on a simple one-dimensional stencil code. We examined the influence of sequential control programs on the memory footprint, data locality and contention on memory controller and motivated that applications using dynamic single assignment benefit from the implementation of a parallel control program. The conditions for the parallelization of a control program are out of the scope of this thesis and could thus only be outlined.

In the next chapter, we present a set of high performance scientific benchmarks based on dynamic single assignment and describe the experimental setup for the experiments conducted in this thesis.

6 Experimental Setup

To demonstrate that the optimizations presented in this thesis apply to real-world applications and thus to show that they are practically relevant, we evaluate our concepts on a set of applications executing on machines with contemporary hardware architectures. The purpose of this chapter is to provide an overview of these applications as well as on the hardware environment used for evaluation. We introduce a set of high performance, scientific applications implemented using the language extensions of OpenStream of Chapter 3 and dynamic single assignment described in Chapter 5 and describe the memory hierarchy of the many-core systems used in our experiments. Furthermore, we provide a methodology for measurements and show which events are quantified.

The chapter is structured as follows. In Section 6.1 we provide an overview of the benchmarks used for evaluation as well as details on their implementation using dynamic single assignment presented in the previous chapter. Section 6.2 presents the different baselines for the evaluation and introduces shared memory programming using tokens for synchronization used in one of the baselines. The methodology for the measurement of the execution time and the collection of statistics using hardware performance counters is explained in Section 6.2.3, which introduces the definition of the measurement interval. Details about the hardware environment are given in Section 6.3 that describes the two test platforms used for the execution of the benchmarks. The parametrization of the benchmarks, e.g., the size of input data and the granularity defining the amount of work per task is presented in Section 6.4. To estimate which benchmarks are most sensitive to the locality of memory accesses, Section 6.5 provides an overview of the characteristics of the applications with respect to the memory hierarchy. The chapter finishes with an analysis of the scalability of the applications of the shared memory baseline in Section 6.6 to show that interleaved allocation across all nodes is essential for performance. *Parts of this chapter were previously published in [46].*

6.1 Benchmarks

For the experimental evaluation of the concepts presented in Chapters 7, 8 and 9, we have implemented a set of high performance, scientific benchmarks. These applications can roughly be divided into four categories: *stencil computations* (*seidel*, *jacobi*, and *blur-roberts*), *integer sorting* (*bitonic*), *clustering* (*k-means*) and *linear algebra* (*cholesky*). In this section, we provide an overview of these benchmarks and briefly describe their implementations.

6.1.1 Seidel

The *seidel* benchmark implements the Gauß-Seidel method, which iterates a five-point stencil over a two-dimensional, dense $N \times N$ matrix of double precision floating point elements with N being a power of two. Similar to the one-dimensional stencil presented in Section 5.3, the value $v^i_{x,y}$ of an element of an iteration i at position (x, y) in the matrix is calculated by taking into account values from the previous iteration $i - 1$ as well as the current iteration, but in two dimensions:

$$v^i_{x,y} = \frac{1}{5}\left(v^i_{x-1,y} + v^i_{x,y-1} + v^{i-1}_{x,y} + v^{i-1}_{x+1,y} + v^{i-1}_{x,y+1}\right)$$

Elements at the border of the matrix are treated as if the values of the missing neighbors were zero. For example, the element in the corner at position $(0, 0)$ is updated as follows:

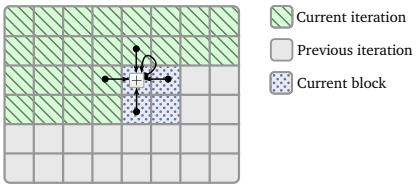
$$v^i_{0,0} = \frac{1}{5}\left(0 + 0 + v^{i-1}_{x,y} + v^{i-1}_{x+1,y} + v^{i-1}_{x,y+1}\right) = \frac{1}{5}\left(v^{i-1}_{x,y} + v^{i-1}_{x+1,y} + v^{i-1}_{x,y+1}\right)$$

As processing of the elements of the entire matrix in a single task would limit parallelism and thus lead to poor performance, the matrix is tiled into blocks of $S_B \cdot S_B$ elements, each treated by a separate task performing a single iteration on the tile. The number of tasks per iteration is thus $\frac{N^2}{S_B^2}$. However, not all of these tasks can execute in parallel, since each block relies on data from its neighborhood, as illustrated in Figure 6.1a. A subset of the dynamic task graph showing the producers and consumers of a task $b^i_{X,Y}$, calculating the values of the ith iteration on the block at coordinates X, Y, is given in Figure 6.1b. The values generated by this task belong to the elements that are within the square-shaped block of the matrix, which are $\{v^i_{x,y} | X \cdot S_B \leq x < (X + 1) \cdot S_B \wedge Y \cdot S_B \leq y < (Y + 1) \cdot S_B\}$. The graph contains two types of dependences: heavy dependences between tasks that treat the same block at different iterations (e.g., between $b^{i-1}_{X,Y}$ and $b^i_{X,Y}$ or between $b^i_{X,Y}$ and $b^{i+1}_{X,Y}$) and light dependences between neighboring blocks of the same iteration (e.g., between $b^i_{X,Y}$ and $b^i_{X+1,Y}$ or between $b^i_{X,Y}$ and $b^i_{X,Y+1}$) or neighboring blocks across iterations (e.g., between $b^i_{X,Y}$ and $b^{i+1}_{X-1,Y}$ or between $b^i_{X,Y}$ and $b^{i+1}_{X,Y-1}$). The heavy dependences correspond to data dependences for entire blocks, which consist of $S_B \cdot S_B$ elements, while light dependences represent data dependences for the borders of blocks and only comprise S_B elements. The value S_{dbl} in the task graph stands for the size of a double precision floating point value of eight bytes.

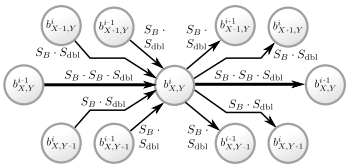
The data of a block is processed by two nested loops iterating over the x and y coordinates of the elements of a block. Throughout the execution of these two loops, each data element is read multiple times. It is therefore crucial for performance that a block fits into the cache of the processor to avoid repetitive accesses to main memory for the same elements. Hence, S_B must be chosen such that $S_B \cdot S_B \cdot S_{\text{dbl}}$ is smaller than the cache capacity associated to a single core.

Similar to the implementation of the one-dimensional version of the benchmark, elements at the borders of a block are read by more than one task and need to be copied to two streams. The first stream is used to pass the whole block to the task treating the same block at the next iteration and the other stream is used to pass the elements to the task treating the neighbor block. This is shown in Figure 6.1c. Figure 6.1d combines the illustrations of Figure 6.1b and 6.1c and shows the data dependences of a single task, the elements of a block and the copied elements at the borders.

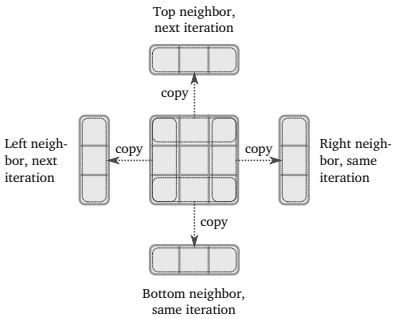
The parallel control program is also similar to the one-dimensional version of the benchmark as each task $b^i_{X,Y}$ creates its indirect successor along the path of heavy dependences, i.e. the task processing $b^{i+2}_{X,Y}$. Figure 6.1e shows a three-dimensional illustration of a task graph that includes the scheme for task creation for 16 blocks and four iterations. The vertical axis in this illustration represents the iterations, while the other two axes indicate the block coordinates of the block treated by a task. Note that neither the tasks in the center of the cube nor the dependences from and to these tasks are shown in order to keep the figure readable. To be used as a drop-in replacement for a sequential version operating on a global matrix in shared memory, i.e. whose elements are not stored in streams, the benchmark requires two types of auxiliary tasks. The first type corresponds



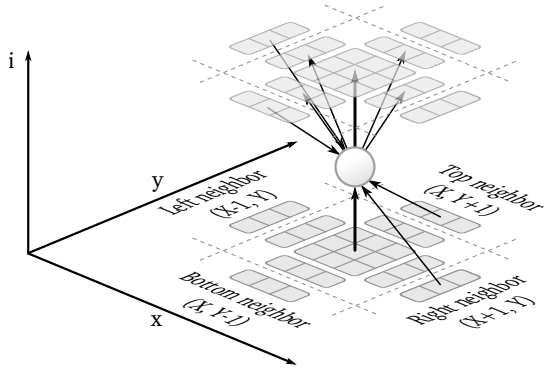
(a) *Illustration of the stencil pattern*



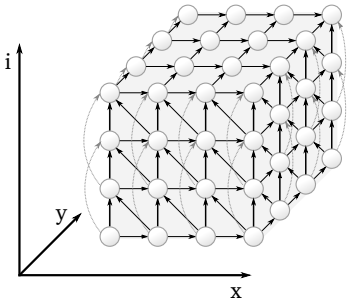
(b) *Dynamic task graph (excerpt)*



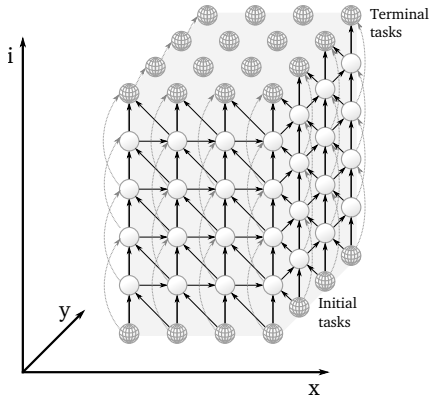
(c) *Data at the borders copied to multiple streams*



(d) *Tiling and data dependences*



(e) *Illustration of the control program*



(f) *Illustration of the control program including auxiliary tasks*

Figure 6.1: *Seidel: two-dimensional five-point stencil*

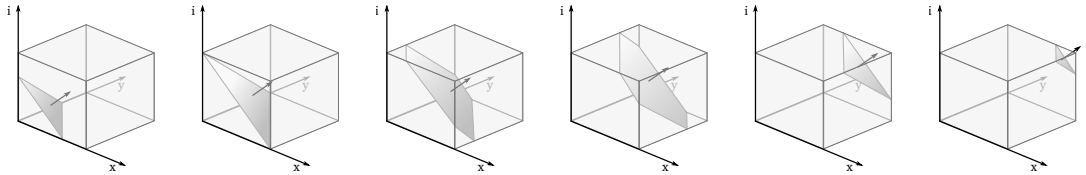


Figure 6.2: *Seidel: progress within the task graph*

to initial tasks that copy data from shared memory to streams and that execute before the tasks that carry out the actual computation of the stencil. The second type consists of terminal tasks that copy the data back to shared memory and which are thus needed at the end of the execution. Figure 6.1f shows the tasks of Figure 6.1e, but also includes the auxiliary tasks. The purpose of the root task (not shown in the graph) is the creation of the initial tasks as well as the tasks for the first iteration. If the number of iterations is smaller than two, the root task also creates the terminal tasks, as these tasks do not have indirect predecessors.

Due to the dependences between tasks within and across iterations, execution starts at the lower left corner of the matrix at block coordinates $(0, 0)$ at the front of the three-dimensional illustration. Afterwards, execution progresses along the dependences, from left to right, from front to the rear and from the bottom to the top as shown in Figure 6.2. Hence, in a first phase, the number of tasks that are ready for execution increases, resulting in growing parallelism (Figure 6.2a to 6.2c). Once the maximum number of tasks ready for execution has been reached, parallelism declines until the task at the upper right corner at the rear of the three-dimensional representation is executed at the very end (Figure 6.2d to 6.2f).

6.1.2 Jacobi

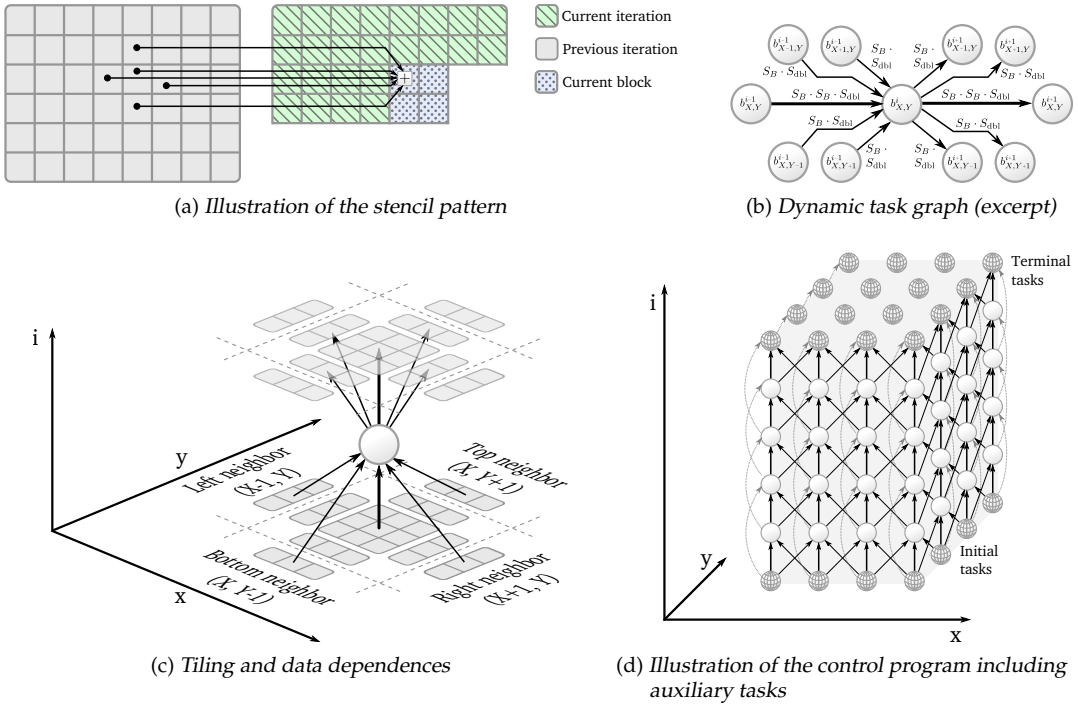
The two-dimensional version of the *jacobi* benchmark, *jacobi-2d*, is a five-point iterative stencil operating on a dense, $N \times M$ matrix of double precision floating point values with N and M being powers of two. The code of this benchmark is inspired by an implementation from the POLYBENCH [75] suite with characteristics similar to the *seidel* benchmark presented above. The matrix is processed in tiles of size $S_{B,N} \times S_{B,M}$ with $S_{B,N}|N$ and $S_{B,M}|M$, resulting in $\frac{N \cdot M}{S_{B,M} \cdot S_{B,N}}$ tasks per iteration. However, in contrast to *seidel*, *jacobi-2d* does not have intra-iteration dependences and each value generated for the ith version of an element only depends on values from the previous iteration $i - 1$:

$$v^i_{x,y} = \frac{1}{5}\left(v^{i-1}_{x-1,y} + v^{i-1}_{x,y-1} + v^{i-1}_{x,y} + v^{i-1}_{x+1,y} + v^{i-1}_{x,y+1}\right)$$

Figure 6.3a illustrates the principles of this calculation. The division into blocks is identical to *seidel* with similar dependences for each task, as shown in Figure 6.3b and Figure 6.3c. The absence of intra-iteration dependences in Figure 6.3c manifests as the absence of arrows pointing from the top towards the task in the middle. The parallel control program is identical to *seidel* and each task creates its indirect successor in the task graph along the iteration dimension, i.e. the task processing the same block at the iteration $i + 2$. Figure 6.3d illustrates the control program for 16 blocks and four iterations, including auxiliary tasks copying data from shared memory to streams as well as auxiliary tasks writing the results back to shared memory.

Although the task graph of *jacobi-2d* is similar to *seidel*, execution progresses differently within the task graph during execution due to the missing intra-iteration dependences. In contrast to the triangle-shaped wavefront of *seidel*, the wavefront of *jacobi-2d* can have a rectangular shape and the program can advance iteration by iteration as shown in Figure 6.4. However, this pattern of progress is not unique and is only likely to occur for a high number of workers. If the number of workers is lower than the number of blocks, it is more likely that tasks of later iterations execute in parallel with tasks of earlier iterations, leading to dynamic pipelining effects. Examples of such cases are given in Figure 6.5, showing triangular-shaped wavefronts similar to *seidel* (Figure 6.5a and 6.5b) as well as progress in a pyramid-like fashion (Figure 6.5c and 6.5d). Which progress pattern occurs at execution time depends on the number of workers, on the order of the creation of auxiliary tasks as well as on the timing of task execution due to dynamic events such as work-stealing.

In addition to the two-dimensional version, we have also implemented a one-dimensional version, *jacobi-1d* implementing a three-point stencil as well as a three-dimensional version of the benchmark, *jacobi-3d* implementing a nine-point stencil with similar characteristics.



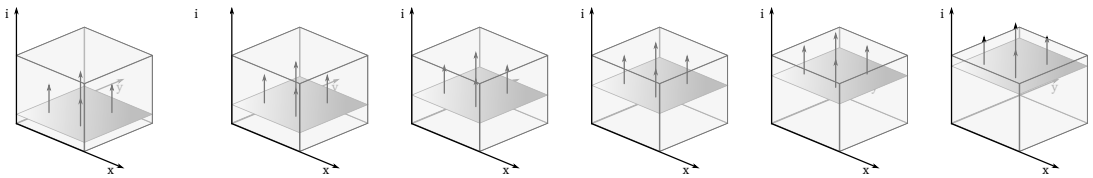
(a) *Illustration of the stencil pattern*

(b) *Dynamic task graph (excerpt)*

(c) *Tiling and data dependences*

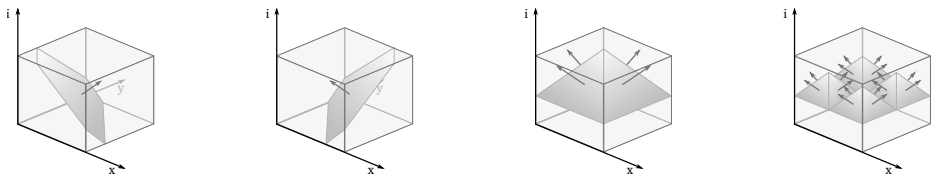
(d) *Illustration of the control program including auxiliary tasks*

Figure 6.3: *Jacobi-2d: two-dimensional five-point stencil*



(a) (b) (c) (d) (e) (f)

Figure 6.4: *Jacobi-2d: progress within the task graph for a high number of workers*



(a) (b) (c) (d)

Figure 6.5: *Jacobi-2d: progress within the task graph depending on the timing*

6.1.3 Blur-roberts

The *blur-roberts* benchmark [59] carries out the stencil computations of two kernels used in image processing on a dense $N \times M$ matrix of double precision floating point elements, processed in blocks of size $S_{B,N} \times S_{B,M}$, with $S_{B,M}|N$ and $S_{B,N}|M$. The application first applies a blur filter on each element of the two-dimensional input matrix, averaging the values of the eight neighbors surrounding the element and the element itself. In a second step, the algorithm applies the Roberts Cross Operator used for edge detection involving the lower left element, the element right below, the left element and the element itself. At the end of this operation, the result is written back to the original matrix. An illustration of the two steps for a single block is given in Figure 6.6a and Figure 6.6b. According to the description of the two steps the final value $v''_{x,y}$ of an element at the position (x, y) in the output matrix is calculated as follows:

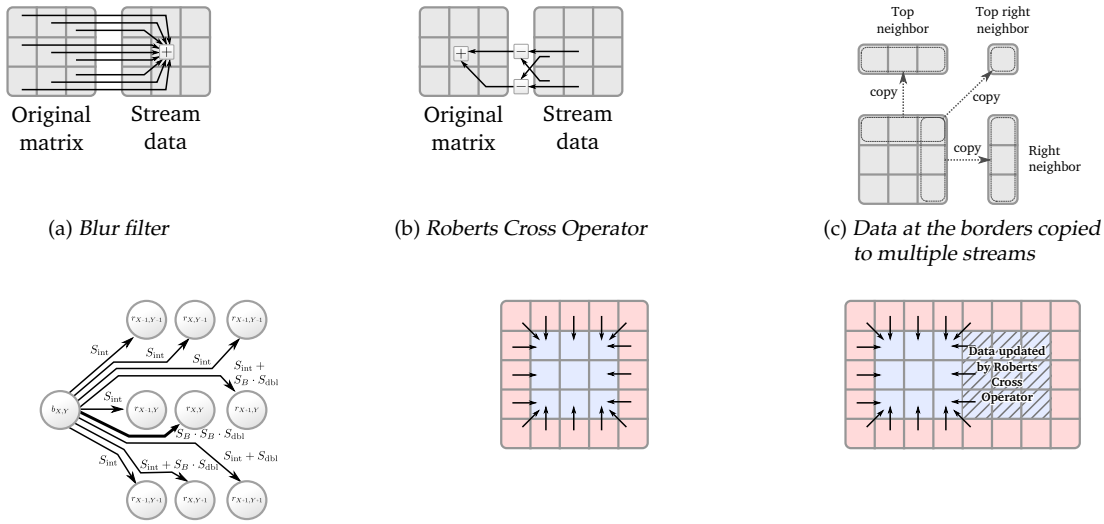
$$v''_{x,y} = v'_{x,y} - v'_{x-1,y+1} + v'_{x,y+1} - v'_{x-1,y} \quad \text{with}$$
$$v'_{x,y} = \frac{1}{9}\left(v_{x-1,y-1} + v_{x,y-1} + v_{x+1,y-1} + v_{x-1,y} + v_{x,y} + v_{x+1,y} + v_{x-1,y+1} + v_{x,y+1} + v_{x+1,y+1}\right)$$

In contrast to *seidel* and *jacobi*, *blur-roberts* only performs a single iteration on each block. To limit the overhead related to the execution of auxiliary tasks, the application does not use dedicated tasks to copy data from shared memory to streams and from streams back to shared memory. Instead, initial data is read from shared memory directly by blur tasks and final data is written back by tasks applying the Roberts Cross Operator. Hence, streams are only used to exchange data between the blur tasks and the tasks applying the Roberts Cross Operator. Similar to *seidel* and *jacobi*, data needed by multiple tasks is copied to several streams. From a block's perspective these are the elements at the top row, the upper right corner and the right column as shown in Figure 6.6c.

One of the key characteristics of *blur-roberts* are the short dependence paths including only two tasks per block, namely the blur filter and the Roberts Cross Operator. Figure 6.6d shows the dependences from the perspective of a single task of the blur filter. The task $b_{X,Y}$ designates a blur task operating on block (X, Y) and $r_{X,Y}$ is a task that applies the Roberts Cross Operator on the block with the block coordinates (X, Y). Besides the main dependence between $b_{X,Y}$ and $r_{X,Y}$ for the block data of size $S_B \cdot S_B$ multiplied with the size of a double precision floating point value S_{dbl}, the graph also contains dependences for the right and top border of a tile, i.e. a dependence of S_B elements between $b_{X,Y}$ and $r_{X+1,Y}$ and a dependence of the same size between $b_{X,Y}$ and $r_{X,Y+1}$, as well as a dependence for the upper right corner of the tile of a single element between $b_{X,Y}$ and $r_{X+1,Y+1}$. The remaining dependences of a single integer of size S_{int} ensure that the values of the input matrix are not overwritten before they have been read by the corresponding tasks carrying out the blur filter as explained below.

When the blur filter is applied to a block, the task associated to this block both reads values from within the block and from the direct neighbors, as shown in Figure 6.6e. As the final data is written back to the original matrix, it must be ensured that the Roberts Cross tasks operating on neighboring blocks do not overwrite the original values before the blur task has finished reading all of the required elements. Without protection, it would be possible that the blur filter operates with data already updated by a Roberts Cross Task as shown in Figure 6.6f. To avoid these early updates of the original matrix, each Roberts Cross Operator task requires the permission of the neighboring blur tasks, which is modeled as a data dependence of a single integer element.

Due to the absence of dependences between blur tasks, all blur tasks can execute in parallel. The available parallelism at the beginning of the execution is thus only limited by the rate of task creation and progress can be made on any part of the matrix. As far as the Roberts Cross Operator tasks are concerned, parallelism is limited by the number of completed blur tasks as well as the location of the blocks associated to these tasks.



(a) *Blur filter*

(b) *Roberts Cross Operator*

(c) *Data at the borders copied to multiple streams*

(d) *Dynamic task graph (excerpt)*

(e) *Data dependences of a blur task*

(f) *Early update of the original matrix*

Figure 6.6: *Blur-roberts: consecutive applications of two stencils*

6.1.4 Bitonic

The *bitonic* benchmark implements a bitonic sorting network [16], capable of sorting an array of 2^N arbitrary 64-bit signed integer values. The sorting process is divided into N stages, each performing a series of compare-and-exchange operations on the elements of the array. At each stage, chunks of the array with a fixed size are sorted with a doubling chunk size from one stage to another. That is, at the end of the kth stage, all chunks of size 2^{k+1} are sorted internally. Hence, the first stage sorts pairs, the second stage all chunks of size four and so on, until the entire array is sorted at the end of stage $N-1$. An advantage of the bitonic sorting algorithm is that data can be treated in fixed-size blocks by a sorting network of a fixed structure, keeping parallelism and the amount of work per task constant on average.

Figure 6.7a shows a sorting network for arrays with eight elements divided into blocks of two elements. During stage 0, each block is sorted internally by applying a compare-and-swap operation to the pair that forms the block. This means that the element with a lower index in the array is swapped with its neighbor if its value is greater. If this is not the case, both values remain at their current positions. Let v_j^k be the element at position j of the array, resulting from stage k. The values v_j^0 of the array at the end of stage 0 are sorted, such that $v_{2i}^0 \leq v_{2i+1}^0$ for $0 \leq i \leq 2^{N-1}-2$. The next stage sorts quadruples and yields v_j^1 with $v_{4i}^1 \leq v_{4i+1}^1 \leq v_{4i+2}^1 \leq v_{4i+3}^1$ for $0 \leq i \leq 2^{N-2}-4$. During the last stage, the entire array is sorted with $v_i^2 \leq v_{i+1}^2$ for $0 \leq i < 2^N - 1$.

Figure 6.7b reveals that there are two types of operations, which can be seen best in stage 2. The triangle-shaped operations merge two chunks by comparing the elements v_{c+i} with $v_{c+s-i-1}$, where c is the base index of a chunk and s the size of a chunk (e.g., for $s=2$ the base index of the third chunk is $c=4$). The second type of operation sorts a chunk internally by comparing and swapping the elements of the upper and the lower half of the chunk for a successively refined chunk size, indicated by the rectangles in Figure 6.7b. The compare-and-swap operations of both types of operations can be grouped, such that a task that executes these operations takes either one block as its input and produces exactly one block of output or such that it takes two blocks on input and generates two output blocks. Figure 6.7c provides an example of such a grouping. Note that this property is independent of the block size and the size of the array, as long as both are a power of two.

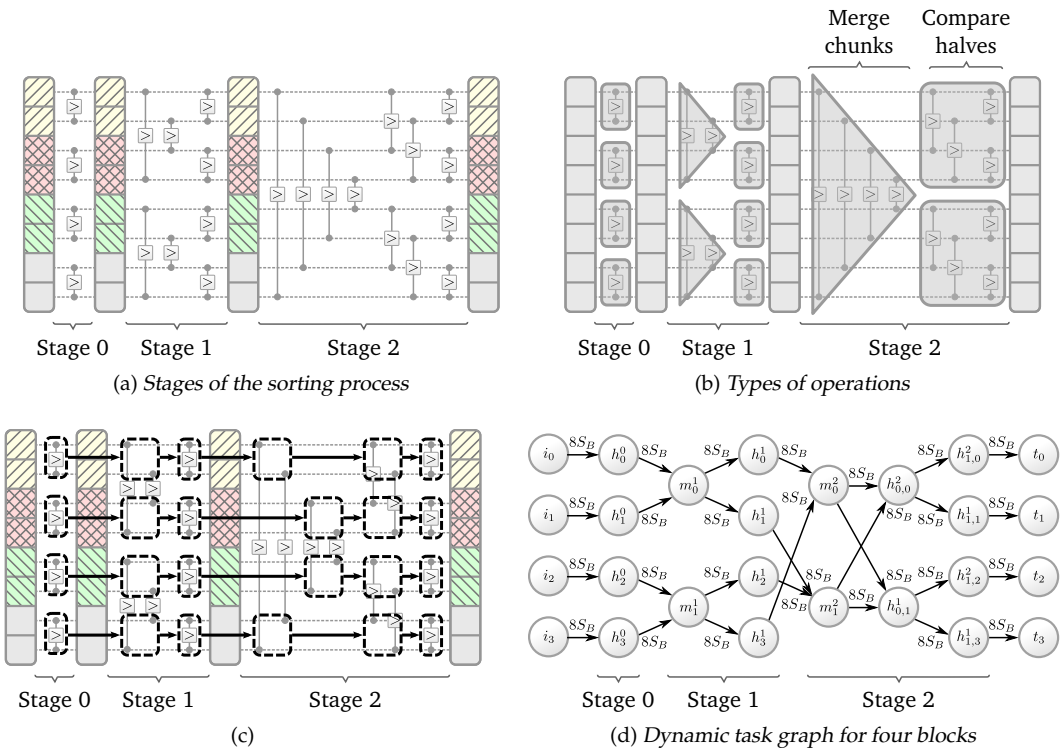


Figure 6.7: *Bitonic: bitonic sorting network*

If a task is associated to each group of operations in the grouping scheme above, this results in a task graph given in Figure 6.7d for the example. The task graph also includes initialization tasks i_0 to i_3 that copy data from shared memory to streams and t_0 to t_3, which write the result back to shared memory. The tasks named h_j^0 execute the operations necessary for the comparisons of the halves of the blocks at stage 0. The first merging tasks, labeled m_j^1, appear at the beginning of stage 1 and provide data for the set of comparison tasks of the same stage, h_j^1. The value S_B indicates the number of elements per block. As each element is a 64-bit integer, the weight of each dependence is thus $8S_B$.

The parallel control program follows the same pattern as the previous applications, in which each task creates an indirect successor in the task graph. However, *bitonic* is less regularly structured than the stencil computations and task creations can span the boundaries of two stages. Figure 6.8 shows an example of a larger sorting network with task creations.

The structure of the task graph of *bitonic* guarantees that the number of tasks ready for execution always ranges between the number of blocks $N_B = \frac{2^N}{S_B}$ and the number of blocks divided by two $\frac{N_B}{2} = \frac{2^{N-1}}{S_B}$ throughout the execution of the program if progress is made from left to right and if all tasks with the same distance to the initialization tasks are executed in parallel. Figure 6.9 illustrates this property on the task graph of the previous figure. If all tasks in a dashed rectangle execute in parallel, parallelism never drops below $\frac{2^{N-1}}{S_B}$ and is at most $\frac{2^N}{S_B}$. For each compare halves operation of a given stage, a constant number of tasks can execute in parallel and although the merge operations operate on data from multiple chunks, they do not act as global barriers synchronizing all tasks before the operation. The upper bound of tasks that can execute in parallel is reached at the beginning of the execution and at the end of each stage, while the lower bound applies to all other intermediate steps. However, this property for parallelism is only valid if progress in the task graph is made stage by stage and if the number of workers is greater than or equal to $\frac{N_B}{2}$. For a lower number of workers, it is more likely that the overall progress spans

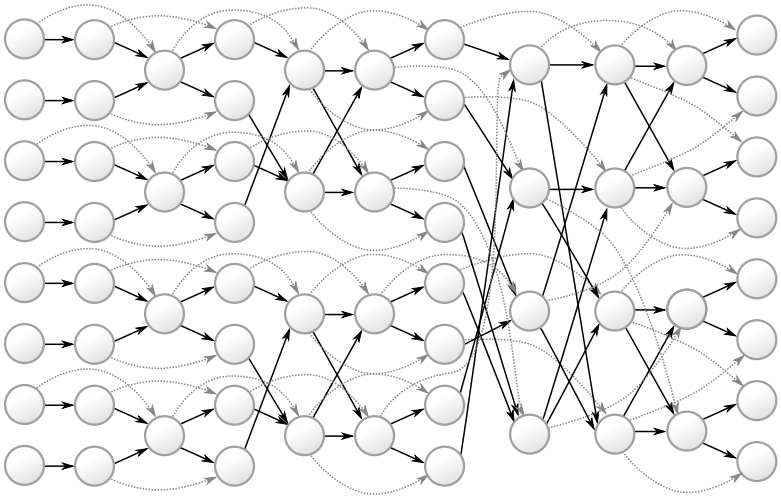


Figure 6.8: *Parallel control program of a bitonic sorting network*

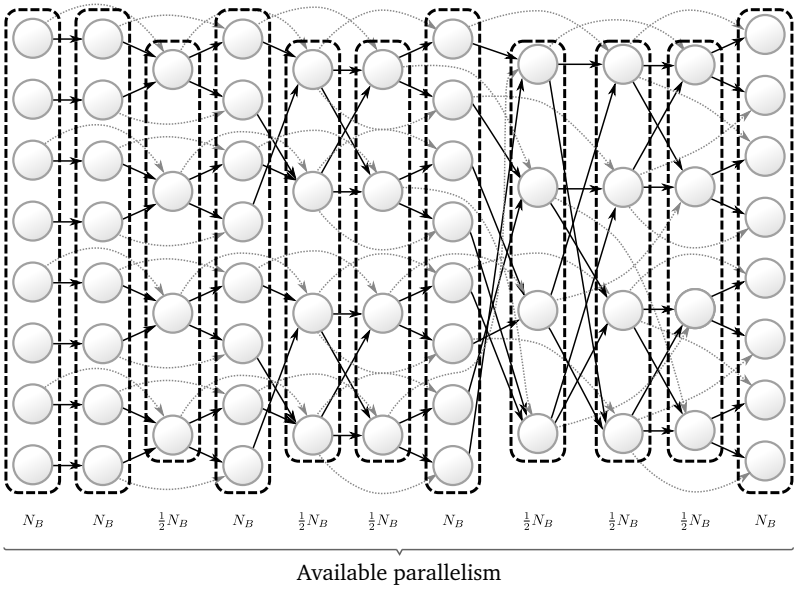


Figure 6.9: *Available parallelism during execution of a bitonic sorting network*

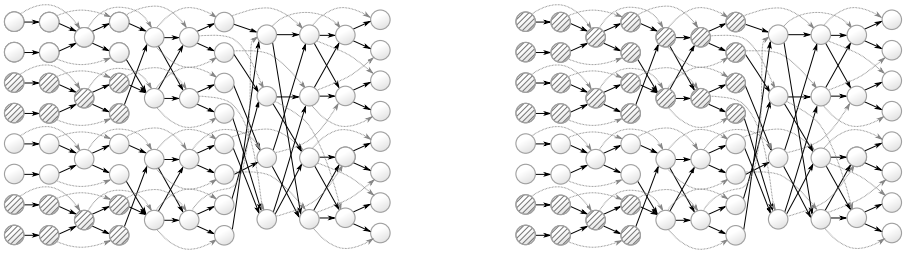
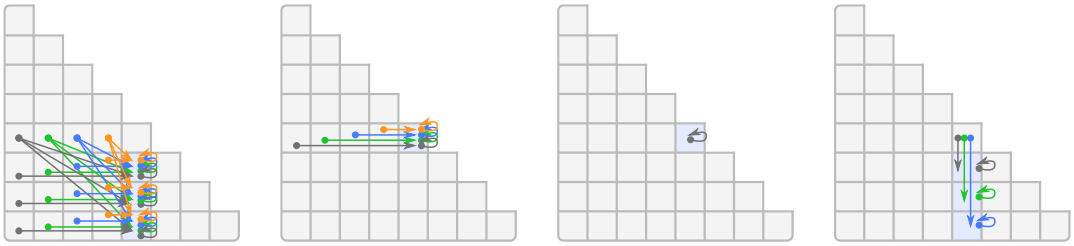


Figure 6.10: *Bitonic: examples of progress within the task graph*



(a) *Matrix multiply* (b) *Symm. rank k update* (c) *Cholesky Factorization* (d) *Equation solving*

Figure 6.11: *Cholesky: Block-wise updates of the matrix*

multiple stages, such as in Figure 6.10a and Figure 6.10a, where tasks that have terminated are marked with a striped pattern.

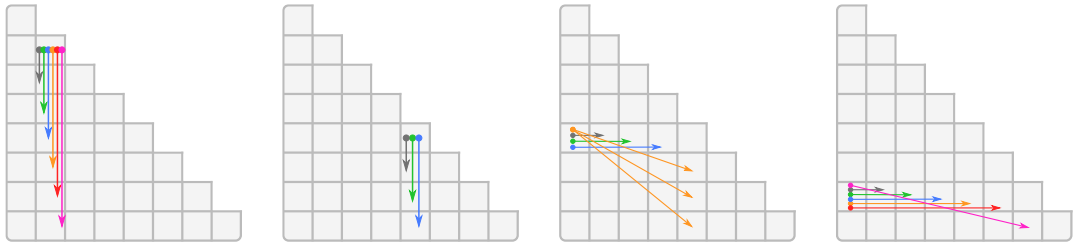
6.1.5 Cholesky

Cholesky is a linear algebra kernel that calculates the lower triangular matrix L of a dense, symmetric, positive definite matrix A, such that $A = L \cdot L^T$. The $N \times N$-matrix A is divided into $S_B \cdot S_B$ sub-matrices $A_{x,y}$, each of size $S_B \times S_B$ with $A_{x,y} = (a_{ij})_{x,y}$ and $x \cdot S_B \leq i < (x+1) \cdot S_B$, $y \cdot S_B \leq j < (y+1) \cdot S_B$. To calculate the Cholesky Factorization of A, it is necessary to apply different operations to the sub-matrices and to propagate updated values accordingly. The principles are illustrated in Figure 6.11, showing the updates of sub-matrices in the fifth column of a matrix decomposed into 64 blocks. The order of the operations is the following:

1. Each block below the block on the diagonal is successively updated by calculating $A'_{x,y} = -A_{i,x} \cdot A^T_{i,y} + A_{x,y}$ and setting $A_{x,y} := A'_{x,y}$ after each step for $0 \leq i < x$ (cf. Figure 6.11a).
2. A symmetric rank k update is applied on the block on the diagonal, i.e., $A'_{x,x} = -A^2_{i,x} + A_{x,x}$ for $0 \leq i < x$ and $A_{x,x} := A'_{x,x}$ after each update (cf. Figure 6.11b).
3. A Cholesky Factorization is performed on the diagonal block, such that $A'_{x,x} = L_{x,x}$ with $A_{x,x} = L_{x,x} \cdot L^T_{x,x}$. As specified for the previous operations, $A_{x,x}$ is set to $A'_{x,x}$ afterwards (cf. Figure 6.11c).
4. The result from the Cholesky Factorization is propagated vertically to all blocks below the diagonal by solving: $A^T_{x,x} \cdot X_{x,y} = A_{x,y}$ and by setting $A_{x,y} := X_{x,y}$ afterwards (cf. Figure 6.11d).

Each of the operations is carried out by a highly optimized sequential function using the interface from BLAS [20] and LAPACK [12], namely `dgemm` for the matrix multiplication, `dsyrk` for the symmetric rank k update, `dpotrf` for the Cholesky Factorization and `dtrsm` for solving the equation from step 4. The implementation of these interfaces is provided by Intel's MATH KERNEL LIBRARY (MKL [2]) with optimized code for high performance processors of the x86 family.

Due to its complexity, the task graph for Cholesky Factorization cannot be shown here. However, the dependences between the blocks as shown in Figure 6.11 give an idea of the structure of the task graph. Each operation, `dgemm`, `dsyrk`, `dpotrf` and `dtrsm` on each block is carried out by a separate task. Data dependences of the factorization algorithm require that the results produced by a task might be communicated to multiple subsequent tasks. The exact number of readers depends on the operation that the producer carries out, the number of blocks as well as on the position of the associated block within the matrix. For example, the result of the `dpotrf` operation on block $(1, 1)$ in Figure 6.12a is read by six tasks calling `dtrsm` on the blocks below. As the number of blocks below the diagonal decreases for blocks towards the left, the number of readers for the same operation on block $(4, 4)$ is only three as shown in Figure 6.12b. For the `dtrsm` operation the number of readers is constant for all blocks in the same column, but the partitioning across `dgemm` and `dsyrk` readers changes as shown in Figure 6.12c and 6.12d. However, the number of readers of each version of a block is known at execution time as soon as the block size and the size



(a) *dpotrf to dtrsm, block* $(1, 1)$ (b) *dpotrf to dtrsm, block* $(4, 4)$ (c) *dtrsm to dgemm and dsyrk, block* $(0, 4)$ (d) *dtrsm to dgemm and dsyrk, block* $(0, 6)$

Figure 6.12: *Cholesky: varying number of readers depending on the operation and the block position*

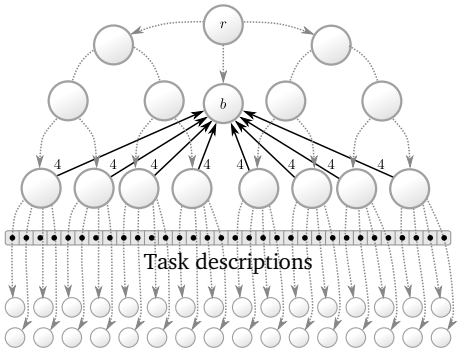


Figure 6.13: *Cholesky: parallel control program*

of the matrix are determined. Hence, it is possible to use the broadcast mechanism involving peek clauses and the tick construct introduced in Section 3.2.3 and 3.2.4. As almost all blocks need to be communicated to multiple tasks our implementation makes heavy use of broadcasts.

Due to the complex dependences and due to the fact that ticks must take place after the creation of all readers of a broadcast, it is difficult to develop a parallel control program that creates tasks throughout the entire execution of the application. Therefore, we have implemented a less complex control program that creates all tasks in parallel at the beginning of the execution. Figure 6.13 shows the principles of this control program [1]. In a first step, the root task allocates an array with a description including the operation and the coordinates of the block for each task to be created. These descriptions are read by the leaf tasks of a tree of tasks whose root is formed by the root task and where each leaf creates the set of tasks that correspond to the portion of the array assigned to it. A barrier task b reads a single integer token from each of the leaf tasks and is thus activated when all tasks for the Cholesky Factorization have been created. The root task synchronizes with this barrier task using a taskwait construct to make sure that the tick operations that are necessary to enable broadcasts all take place after the creation and matching of all readers.

Progress on the factorization is made starting with the upper left block of the matrix in vertical and horizontal direction. The available parallelism at the beginning of the execution is low, but increases until the maximum is reached towards the middle of the execution. Afterwards, parallelism decreases until the block at the lower right is processed and execution finishes. Instead of using auxiliary tasks for the transfer of data from shared memory to streams, the data is simply copied by the first task accessing a block. In contrast to this, the transfer from streams back to shared memory is done by dedicated tasks. The reason for this implementation scheme is that a transfer at the end of the execution of a broadcasting task would delay its termination, which might

1. The number of 32 task descriptions in this example does not represent an actual value of an instance of *cholesky* and only serves as an example for the illustration of the principles of the control program. Moreover, the actual data dependences for the tasks implementing the factorization are not shown.

result in delaying the start of a potentially high number of reading tasks. By adding a dedicated task to the readers of the broadcast the transfer of the result to shared memory can be carried out in parallel.

6.1.6 K-means

K-means is a data-mining benchmark that partitions a set of n d-dimensional points into k clusters using a naive implementation of the K-means clustering algorithm. The algorithm starts with a random selection of k points from the points to be clustered as the cluster centers and then calculates for each point p_i and each cluster center c_q the euclidean distance $\delta_{i,q}$:

$$\delta_{i,q} = \sqrt{\sum_{j=0}^{d}(v_{i,j} - c_{q,j})^2} \quad \text{with} \quad p_i = (v_{i,0}, \ldots, v_{i,d-1}) \quad \text{and} \quad c_q = (c_{q,0}, \ldots, c_{q,d-1})$$

Afterwards, each point is associated to the cluster with the minimal distance [2]. The cluster centers for the next iteration are updated by calculating the barycenters of their associated points of the current iteration:

$$c'_q = \frac{1}{|A_q|} \sum_{p_i \in A_q} p_i \quad \text{where} \quad A_q = \{p_i | \nexists \delta_{i,r} : \delta_{i,r} < \delta_{i,q} \wedge r \neq q\}$$

The algorithm stops when the number of points that changed their associated clusters falls below a user-defined threshold. The principal data structures used by the implementation of the clustering algorithm are a global array with the multi-dimensional points to be clustered, an array with one point per cluster representing its current center and an array storing the cluster membership of each point. While the array with the points is referenced only in read mode, both the cluster center array and the membership array are updated at each iteration. The parallelization is straightforward: the array containing the points to be clustered is divided into equal-sized blocks of S_B elements and during each iteration, each of these blocks can be processed in parallel. The new barycenters are calculated by aggregating intermediate values of the sum successively in a tree-like fashion.

The structure of the task graph for the clustering of an array with eight blocks is given in Figure 6.14. For readability not all task creation relationships and not all data dependences and weights are shown. The creation and execution of this graph can roughly be divided into four phases.

During the first phase, the initial tasks as well as the tasks for the first and second iteration of the algorithm are created by the tree-like structure on the left side formed by $c_{0,0}$, $c_{1,0}$, $c_{1,1}$, $c_{2,0}$, …, $c_{2,3}$. The leaves of this tree, i.e., $c_{2,0}$ to $c_{2,3}$, are responsible for the creation of the initial tasks i_0 to i_7 reading data from shared memory and writing it to streams as well as for the tasks k_0^0 to k_7^0 of the first iteration of the clustering algorithm and k_0^1 to k_7^1 of the second iteration. This ensures that maximum parallelism is achieved quickly after the start of the application [3].

The second phase consists in the execution of i_0 to i_7, which read the blocks of multi-dimensional point data, the current cluster centers and the membership information from shared memory and write this data to streams.

In the third phase, the application performs as many iterations as needed until the number of points whose assigned cluster changes falls below a threshold. Except the first and last iteration, each iteration is characterized by three steps. The first step consists in propagating the updated cluster centers by a tree-like subgraph formed by the tasks $p_{j,o}^i$, where i stands for the iteration, j for the depth of the propagating task below the root $p_{0,0}^i$ of the subgraph and o for the identifier of the task at depth j. In the second step, each task k_j^i calculates the distance of each point of block j

2. The actual implementation leaves out the calculation of the square root, since the squared euclidean distance $\delta_{i,q}^2$ is sufficient to determine which of the cluster centers is nearest to a point.

3. The tree-like structure is implemented slightly differently in the actual implementation. However, the principle of parallel task creation remains the same.

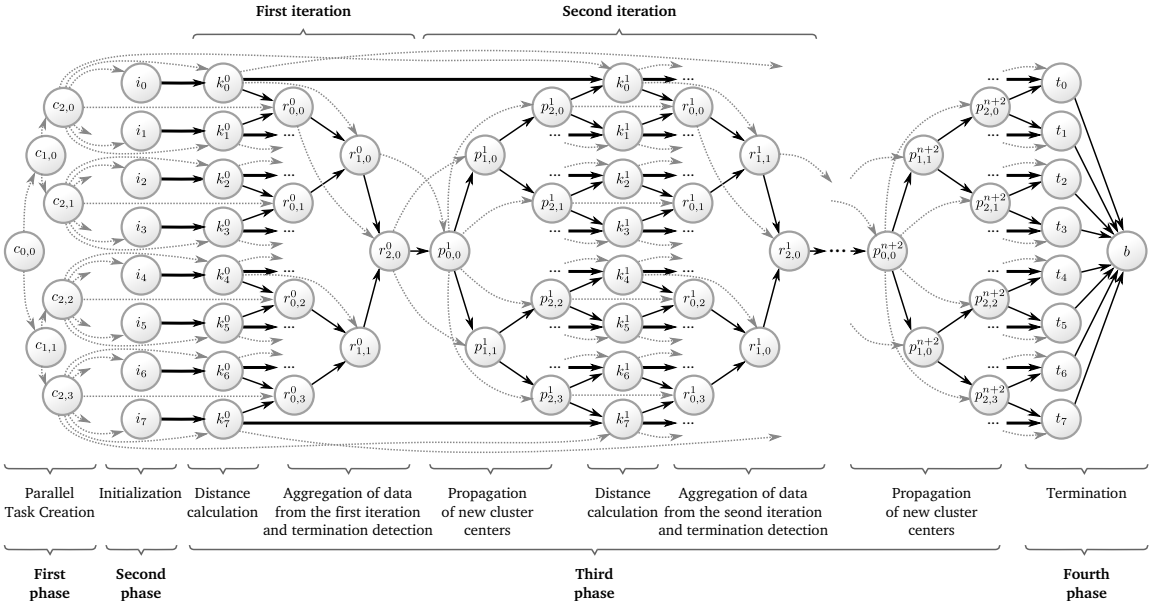


Figure 6.14: *K-means: clustering of multidimensional data*

to the cluster centers and updates cluster membership accordingly. In the final step of an iteration, the new cluster centers are calculated based on this membership. Furthermore, statistics about the number of points that changed cluster membership are aggregated. This is done by the tasks $r^i_{j,o}$, where i designates the number of the iteration whose results are analyzed, j stands for the depth of the task in the aggregating tree starting with the leaves and o corresponds to the number of the task among all tasks at the same level of the aggregation. The last of the tasks in the tree finally detects whether the algorithm has terminated (e.g., $r^1_{2,0}$ in the figure).

The fourth and final phase consists in the termination of the algorithm. Due to the structure of the control program, in which each task k^i_j creates k^{i+2}_j, and the fact that termination detection takes places after termination of all tasks of an iteration, the application terminates two iterations after the first iteration for which the number of points that changed membership is below the threshold. Let q be the iteration at which termination is detected. The tasks k^{q+1}_j of the following iteration are not necessary, but cannot be canceled and the algorithm cannot terminate instantly. To bypass this problem, each task k^i_j receives a value from a propagating task that indicates whether the algorithm is terminating or whether execution continues. In the first case, the task skips all calculations and simply forwards all data to the terminal task t_j. In the latter case, the task performs a single iteration on the associated block. The application terminates when all terminal tasks have finished execution. This can be detected by the root task by synchronizing using the taskwait construct with a task b reading a token from all tasks t_j.

The *k-means* benchmark also highlights an issue related to the implementation of broadcasts in OpenStream. The data of the points to be clustered is only read and never written by any of the tasks. Hence, a broadcast from the initial task reading a block from shared memory to all tasks that read the block would be preferable (e.g., from i_0 to k^0_0, k^1_0, …, k^q_0). However, as the number of readers of a block depends on the number of iterations q, which is only known when termination detection takes place, and as the number of readers of a broadcast must be known before it can take place, it is impossible to use the broadcast mechanism of OpenStream to pass the point data to all tasks k^i_j. Thus, to be able to determine the working set of a task and its memory accesses in the run-time all read-only data must be copied manually from one iteration to another. To this end, each task k^i_j performs a call to `memcpy` before termination, copying the entire block of points from an input view to an output view of the same size.

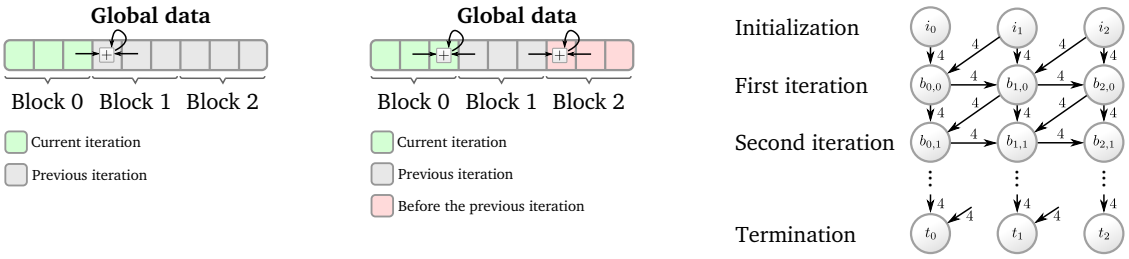
As a result of the aggregation mechanism which features a single task towards which all dependence paths converge, progress in the task graph is made iteration by iteration. At each iteration, parallelism increases exponentially from $p^i_{0,0}$ until all tasks k^i_j have become ready for execution. Afterwards, parallelism decreases exponentially until the last task of the reduction executes. Note that the depth of the tree of reduction and propagation tasks in the examples corresponds to the logarithm to the basis two of the number of blocks and that the amount of work for these tasks is proportional to the number of cluster centers. Usually, the number of cluster centers is several orders of magnitude lower than the number of points to be clustered, such that the impact on execution time is small. However, due to the short execution time of reduction and propagation tasks the relative overhead for task creation, synchronization and destruction can be high. Therefore, the implementation of *k-means* allows the specification of an arbitrary number f of aggregations at each step of a reduction or a propagation as long as the number of blocks is a power of f. The amount of work of each task is proportional to the product of the number of clusters and f. Thus, for a higher value for f both the relative overhead as well as the depth of reduction and propagation trees decreases. However, at the same time parallelism between two iterations decreases. During parametrization of the benchmark for a specific machine parallelism and overhead must be traded off by adjusting f and the block size.

6.2 Baselines and measurement

The evaluation of the concepts presented in this thesis highlights different aspects related to data locality and performance. For the comparison in our analyses we use two parallel baselines as well as a sequential baseline. The first parallel baseline corresponds to the dynamic single assignment version of the benchmarks presented in the previous section, executed by the NUMA-aware implementation of the OpenStream run-time described in Chapter 4. This run-time uses complete random work-stealing and does not include any of the optimizations for scheduling and memory allocation presented in Chapters 7 and 8. The second parallel baseline consists in shared memory implementations of the benchmarks that do not follow the principles of dynamic single assignment and that use streams only for token-based synchronization. This baseline highlights the benefits and drawbacks of using dynamic single assignment and allows for the comparison with more conventional shared memory programming. The principles of this programming style are given in the following section. Last, the sequential baseline consists of a set of sequential implementations of the benchmarks not using any OpenStream-specific constructs and executing without the OpenStream run-time system.

All implementations of the benchmarks use the same sequential optimizations, i.e. tiling of matrices and arrays into blocks, manual loop unrolling etc., and all parallel baselines use identical control programs. Furthermore, the data sets processed by the benchmarks implementing the same algorithms are identical. In some cases, the shared memory and sequential implementation differ sightly from the dynamic single assignment version. This is the case for in-place updates of memory locations which require that older versions of elements which have not been read by all dependent calculations are saved in a separate memory location. For example, the shared memory versions of the *jacobi* benchmarks require the data at the borders of a tile to be saved for the treatment of neighboring blocks before overwriting the current block.

With the exception of *cholesky*, all benchmarks have been implemented in three versions: dynamic single assignment, token-based synchronization and as a sequential algorithm. Instead of comparing the dynamic single assignment version of *cholesky* to a shared memory version with token synchronization, we compare its performance to highly optimized, state-of-the-art linear algebra codes. This comparison in provided in Chapter 9 dealing with the optimization of broadcasts.



(a) *Division into blocks* (b) *Pipelining on blocks* (c) *Dynamic task graph*

Figure 6.15: *1d stencil synchronizing with tokens*

6.2.1 Synchronization using tokens

As an example of shared memory programming with OpenStream tasks, consider the 1-dimensional Seidel stencil introduced in Section 5.3. At each iteration, every element is updated by calculating the average of the current value of the left neighbor, the value of the previous iteration of the right neighbor and the value of the element itself (Figure 6.15a). Elements of different iterations that are not direct neighbors can be processed in parallel. For example, the jth iteration of the element at index i can take place at the same time as iteration $j-2$ of the element at index $i+2$.

The shared memory implementation of this application with token synchronization uses a global array to store the data elements, implicitly partitioned into multiple blocks as shown in Figure 6.15b. Tasks that operate on the same or neighboring blocks are synchronized through streams. Similar to the dynamic single assignment version, a task for each block and each iteration is created. The benchmark also creates initialization and termination tasks that are necessary to set up and terminate synchronization, shown in Figure 6.15c. As in the dynamic single assignment version, the dependences in the task graph reflect the data dependences of Figure 6.15a: horizontal arrows represent dependences to the right neighbors, vertical arrows represent dependences to the same sets of element from previous iterations and diagonal arrows indicate dependences to the left neighbors. However, in contrast to the dynamic single assignment version, the weight associated to each edge is the size of an integer element of four bytes rather than the actual size of the data that corresponds to the dependence. As all data is stored in the global matrix, the dependences between tasks are only control dependences that ensure the correct order of execution of the tasks. The stream elements that correspond to these control dependences can be seen as tokens that are passed from one task to another for task activation, hence the name token-based synchronization. The following code listing shows the implementation of this token-based synchronization scheme.

Listing 6.1: One-dimensional stencil using tokens for synchronization

```
1  int main(int argc, char** argv)
2  {
3    int num_iters = 2;
4    size_t N = 1024*1024;
5    size_t B = 256*1024;
6    size_t num_blocks = N/B;
7    double* data = malloc(N*sizeof(double));
8
9    /* Streams with tokens for synchronization */
10   int tokens_center[(num_iters+2)*num_blocks] __attribute__((stream));
11   int tokens_left[(num_iters+2)*num_blocks] __attribute__((stream));
12   int tokens_right[(num_iters+2)*num_blocks] __attribute__((stream));
13
14   /* Indexes in the array of streams for input tokens */
15   #define LEFT_IN_IDX ((iter+1)*num_blocks+block-1)
16   #define RIGHT_IN_IDX (iter*num_blocks+block+1)
17   #define CENTER_IN_IDX (iter*num_blocks+block)
18
19   /* Indexes in the array of streams for output tokens */
20   #define LEFT_OUT_IDX ((iter+1)*num_blocks+block)
```

```
21    #define RIGHT_OUT_IDX ((iter+1)*num_blocks+block)
22    #define CENTER_OUT_IDX ((iter+1)*num_blocks+block)
23
24    for(size_t i = 0; i < N; i++)
25      data[i] = random_double();
26
27    /* Token views */
28    int left_in_token, right_in_token, center_in_token;
29    int left_out_token, right_out_token, center_out_token;
30
31    /* Create tasks writing initial tokens */
32    for(size_t block = 0; block < num_blocks; block++) {
33      /* Leftmost block */
34      if(block == 0) {
35        #pragma omp task output(tokens_center[block] << right_out_token)
36        { }
37      }
38      /* Other blocks */
39      else {
40        #pragma omp task output(tokens_center[block] << right_out_token, \
41            tokens_left[block] << left_out_token)
42        { }
43      }
44    }
45
46    /* Create one task for each block and each iteration */
47    for(size_t iter = 0; iter < num_iters; iter++) {
48      for(size_t block = 0; block < num_blocks; block++) {
49        /* Leftmost block */
50        if(block == 0) {
51          #pragma omp task \
52            input(tokens_center[CENTER_IN_IDX] >> center_in_token, \
53                  tokens_left[RIGHT_IN_IDX] >> right_in_token) \
54            output(tokens_right[RIGHT_OUT_IDX] << right_out_token, \
55                   tokens_center[CENTER_OUT_IDX] << center_out_token)
56          {
57            process_block(data, block, num_blocks);
58          }
59        }
60        /* Rightmost block */
61        else if(block == num_blocks-1) {
62          #pragma omp task \
63            input(tokens_center[CENTER_IN_IDX] >> center_in_token, \
64                  tokens_right[LEFT_IN_IDX] >> left_in_token) \
65            output(tokens_left[LEFT_OUT_IDX] << left_out_token, \
66                   tokens_center[CENTER_OUT_IDX] << center_out_token)
67          {
68            process_block(data, block, num_blocks);
69          }
70        }
71        /* Block in the center */
72        else {
73          #pragma omp task \
74            input(tokens_center[CENTER_IN_IDX] >> center_in_token, \
75                  tokens_right[LEFT_IN_IDX] >> left_in_token, \
76                  tokens_left[RIGHT_IN_IDX] >> right_in_token) \
77            output(tokens_right[RIGHT_OUT_IDX] << right_out_token, \
78                   tokens_left[LEFT_OUT_IDX] << left_out_token, \
79                   tokens_center[CENTER_OUT_IDX] << center_out_token)
80          {
81            process_block(data, block, num_blocks);
82          }
83        }
84      }
85    }
86
87    /* Create tasks consuming output tokens of the final iteration */
88    for(size_t block = 0; block < num_blocks; block++) {
89      /* Leftmost block */
90      if(block == 0) {
91        #pragma omp task input(tokens_center[num_iters*num_blocks+block] >> center_in_token)
92          { }
93      }
94      /* Other blocks */
95      else {
96        #pragma omp task \
97          input(tokens_left[num_iters*num_blocks+block] >> left_in_token, \
98                tokens_center[num_iters*num_blocks+block] >> center_in_token)
99          { }
100     }
```

```
101     }
102
103     #pragma omp taskwait
104
105     free(data);
106
107     return 0;
108   }
```

As tokens are represented by simple integers, all streams are typed as integer streams. Lines 10, 11 and 12 define arrays of streams with one stream for each dependence of a block, each block and each iteration. The stream that a token is passed through for synchronization is determined using the same macros as for the dynamic single assignment version of Section 5.3.4, defined in lines 15 to 22.

The shared array containing all data is allocated in line 7 with an ordinary call to `malloc` and initialized in line 25. Input and output views on the synchronization tokens are declared in lines 28 and 29. The views only serve for proper declaration of the stream accesses of each task, but are not accessed within a task.

The initial tasks, created by the loop starting at line 32, only generate tokens and do not perform any useful computation. Therefore, their task bodies do not include any statement. The same applies for the terminal tasks created in lines 88 to 101, which consume the tokens produced by the tasks of the final iteration. The function `process_block` carries out the actual computation and is called from the main tasks in lines 51 (leftmost block), 62 (rightmost block) and 73 (blocks not at the border of the array). Note that the position of the block in the global array determines the number of tokens the associated task needs in order to become activated: the block on the very left only depends on the previous value of the block at the same position, while all the other blocks also depend on the value of their left neighbors of the current iteration.

Before freeing the global array, it must be ensured that all tasks operating on its elements have terminated. This is achieved by the taskwait construct in line 103 that blocks execution of the main function until all tasks have completed.

As mentioned in the previous section, all shared memory implementations of benchmarks are provided with a parallel control program that is identical to the dynamic single assignment version. However, for simplicity, the listing above only contains a sequential control program.

6.2.2 Generic optimizations for load balancing across memory controllers

The sequential initialization of the matrix in Listing 6.2.1 causes all pages of the matrix to be placed in the memory of the NUMA node of the worker executing the root task. Hence, during execution of the benchmark, all memory accesses target the same memory controller, leading to high contention resulting in high latency of memory accesses and low throughput. For benchmarks with regular data structures, such as arrays and matrices, load balancing can be addressed simply by distributing the pages of the data structures over multiple nodes. On Linux systems, LIBNUMA provides support for interleaved allocation of data on a set of nodes that can be specified by the application. By passing all of the system's nodes as the list, it is thus possible to distribute data over the entire machine. The interleaving is implemented at page granularity and allocates pages in a round-robin fashion by cycling over the nodes from the list. Figure 6.16 shows this principle for a an array allocated in an interleaved fashion on a system with n nodes in total.

In order to take advantage of this load balancing technique, we have provided all parallel benchmarks with calls to a run-time function implementing interleaved allocation. This does not only apply to the global data structures of the shared memory parallel baseline, but also to the shared memory data structures for input and output data of the dynamic single assignment versions.

After this description of the concepts and implementation of the benchmarks in the previous sections, the next section focuses on measurement of indicators for performance and data locality.

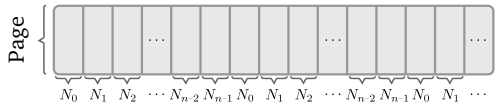


Figure 6.16: *Interleaved allocation on n nodes*

6.2.3 Execution phases and measurement interval

Depending on the benchmark, not all of the executed instructions and micro-architectural events are relevant for the experimental evaluation and it does not always make sense to measure the time and all events that occur during the entire execution. For example, the initialization of the input matrices of *seidel*, *jacobi* or *blur-roberts* is not part of the actual algorithm that the benchmark implements and thus should not be taken into account for any evaluation. However, these steps are essential for the execution of the benchmark and cannot be omitted.

Therefore, we have instrumented all benchmarks with calls to run-time functions that indicate the beginning and the end of a relevant part of the execution. In the remainder of this thesis, we refer to the interval between these function calls as the *measurement interval*. The execution time and all data obtained using hardware performance counters refer exclusively to the measurement interval and exclude all events related to initialization and de-allocation of resources. Activities that are carried out by the run-time system outside the measurement interval are also excluded.

Figure 6.17 provides an overview of the activities that are included in and excluded from the measurement interval, respectively. On startup of an OpenStream application, control is transferred to the run-time system before any application-specific code is executed. The run-time starts by initializing the data structures needed by the first worker, which will later execute the root task of the application. Afterwards, memory pools are initialized, with one memory pool per NUMA node as described in Section 4.4.2, including the allocation of large chunks of memory explained in Section 4.5.1. Note that these chunks are only allocated logically, which results in fast initialization of the memory pools.

Once all memory pools are initialized, the run-time creates the remaining workers w_1 to w_{n-1}, where n corresponds to the number of cores. This is followed by the initialization of hardware performance counters through appropriate calls to the PAPI [80] library [4]. Depending on the type of hardware counters provided in the configuration of the run-time system, e.g., per-core counters or hardware counters that count micro-architectural events generated by multiple cores, all of the workers or only a subset of workers are involved in this part of the initialization. After this step, the run-time is set up and ready to start executing tasks. However, as no task has been created so far, all workers except the first worker executing the root task remain idle. The root task is created implicitly and starts with the `main` function of the application. Usually, all streams used by the program are declared in the local scope of the main function at the beginning, such that the first instructions of the application correspond to the creation of streams. Note that, although an application can request the creation of large arrays with tens of thousands of streams, the duration of this phase is usually negligible compared to the duration of the measurement interval.

The next step consists in the initialization of the benchmark, which includes the allocation of global data structures as well as the generation of initial data. The beginning of the measurement interval is marked by a call to `gettimeofday` that captures the current time followed by a call to `openstream_start_hardware_counters`, which starts counting micro-architectural events. All the activities of the benchmark itself as well as all activities of the run-time that occur within the measurement interval are included in the statistics. For the benchmarks studied in this thesis, this involves the creation of all tasks, the execution of all tasks including auxiliary tasks transferring data between global data structures and streams as well as the termination of the algorithm using a data-flow barrier. The end of the measurement interval is marked by a call to `openstream_`

4. The use of hardware performance counters can be entirely disabled in the configuration of the run-time, e.g., if only the execution time is measured.

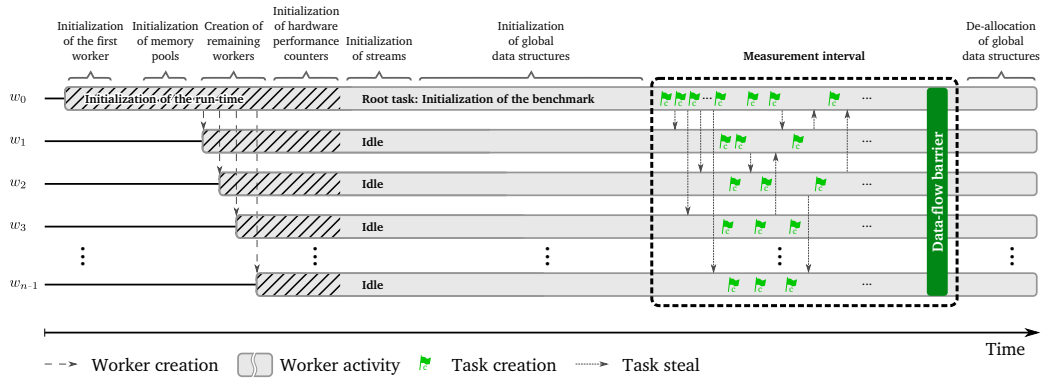


Figure 6.17: *Phases during execution of a benchmark*

`pause_hardware_counters` and a second call to `gettimeofday`. The wall clock execution time is calculated by subtracting the timestamps obtained through the two calls to `gettimeofday` and is written to standard output afterwards. Values for hardware performance counters are also written to the standard output, but are calculated by the run-time aggregating values obtained for all cores.

6.3 Hardware environment

For the experimental evaluation of our concepts we used two many-core systems: an AMD Opteron platform and an SGI system with Intel Xeon processors. In this section, we provide an overview of the memory hierarchy of these systems and determine the ratio of the latency of accesses to remote memory over the latency of accesses to local memory.

6.3.1 Opteron test platform

The first test system is a quad-socket AMD Opteron 6282 SE running at a clock frequency of 2.6 GHz. Figure 6.18a shows a hierarchical view of its basic components. At the coarsest level, the machine is composed of 4 physical packages called *multi-chip modules*. Each module contains two dies, each of which finally contains 8 cores organized as pairs of cores sharing some resources.

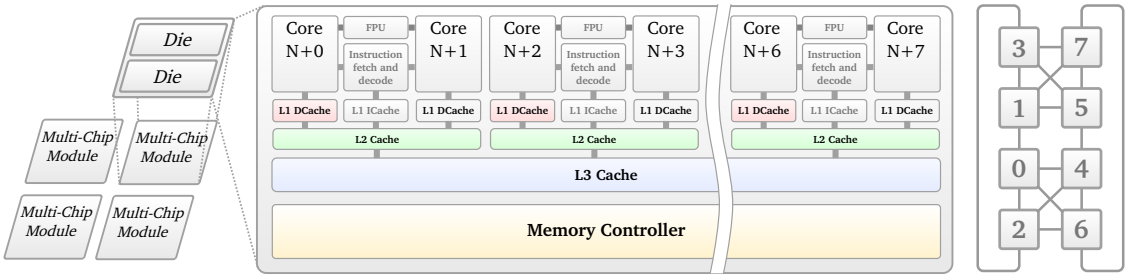
At the core pair level, the floating point unit, the instruction fetcher and decoder, the first-level instruction cache as well as the 2 MiB of second level cache are shared. The third level cache of 6 MiB and the memory controller are shared by all of the cores located on the same die. Among the private, per-core resources are the integer unit and the 16 KiB first level data cache.

As implied by the sharing of memory controllers, main memory is divided into 8 equal-sized NUMA domains of 8 GiB so that the total amount of main memory available is 64 GiB. Their distances as reported by the NUMACTL tool [58] is visualized in Figure 6.18b. For each node there are four neighbors at a distance of one hop and three neighbors at a distance of two hops.

The machine runs Scientific Linux 6.2 with kernel 3.10.1.

6.3.2 SGI test platform

The second system we have used is an SGI UV2000 with a total of 192 cores and 756 GiB RAM distributed over 24 NUMA nodes. The system is organized in *blades*, each of which contains two Intel Xeon E5-4640 CPUs running at 2.4 GHz. Each CPU has 8 cores and has direct access to a memory controller. The cache hierarchy consists of three levels with separate, core-private instruction and data cache at the first level, each with a capacity of 32 KiB, a core-private unified L2 cache of 256 KiB at the second level and a unified third level cache of 20 MiB, shared among all cores of the CPU. Each core supports two hardware threads through Intel's Hyperthreading technology, such that the system appears to have 384 processing units in total. For our experiments,



(a) *Organization of the memory hierarchy* (b) *Links*

Figure 6.18: *Architecture of the Opteron test system*

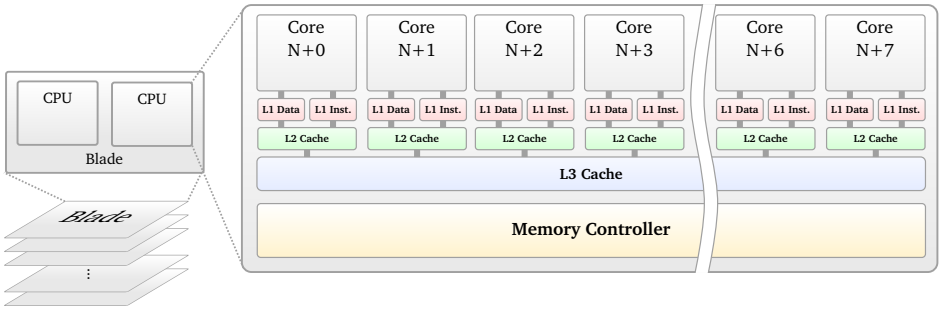


Figure 6.19: *Architecture of the SGI test system*

however, Hyperthreading has been disabled for a unique mapping of logical processor identifiers to physical cores.

Each blade has a direct connection to a set of other blades and an indirect connections to the remaining ones. From a core's perspective, a memory controller can be either local if associated to the same CPU, at 1 hop if on the same blade, at 2 hops if on a different blade that is connected directly to the core's blade or at 3 hops if on a remote blade with an indirect connection.

The system runs SUSE Linux Enterprise Server 11 SP3 with kernel 3.0.101-0.46-default.

6.3.3 Latency of memory accesses and NUMA factors

To measure the latency of memory accesses as a function of the distance in hops between the requesting core and the memory controller that satisfied the request, we have implemented a synthetic benchmark that allocates a buffer on a single node using LIBNUMA, initializes it and measures the execution time for a sequence of memory accesses to the buffer from a core of a specific node. Each sequence traverses the whole buffer from beginning to end in steps of $64\,\mathrm{B}$, such each cache line of the buffer is only accessed once. The buffer size was set to $1\,\mathrm{GiB}$ to make sure that data is evicted from the cache before it is reused and thus to measure only memory accesses that are satisfied by the memory controller and not by the hierarchy of caches.

The results are summarized in table 6.1 and 6.2. On both systems, the latency of memory accesses increases with the distance between the requesting core and the targeted memory controller. In addition, for each distance, write accesses are significantly slower than read accesses. The rightmost column of each table indicates how many times accesses are slower compared to accesses to a node's local memory. For read accesses on the Opteron system, these values range from 1.81 for on-chip accesses to a memory controller at a distance of one hop to a factor of 4.34 for off-chip accesses at a distance of two hops. For write accesses these values are lower, ranging only from 1.2 to 2.48 due to the higher initial latency of local write accesses. Not surprisingly the factors for both read and write accesses are significantly higher on the larger SGI system. The highest factor of 7.48 corresponds to read accesses at a distance of three hops.

	Read accesses	Write accesses	Factor R/W
Local	1288.50 ± 1.22 ms	2256.94 ± 14.06 ms	1.00 / 1.00
1 hop (on-chip)	2328.38 ± 0.49 ms	2717.62 ± 12.16 ms	1.81 / 1.20
1 hop (off-chip)	2781.36 ± 0.56 ms	3934.62 ± 0.56 ms	2.16 / 1.74
2 hops	5601.58 ± 0.57 ms	5601.34 ± 0.55 ms	4.34 / 2.48

Table 6.1: *Average latency of read and write accesses as a function of the distance for the Opteron system*

	Read accesses	Write accesses	Factor R/W
Local	934.82 ± 5.74 ms	1307.40 ± 2.95 ms	1.00 / 1.00
1 hop	4563.10 ± 3.02 ms	5282.38 ± 1.56 ms	4.88 / 4.04
2 hops	5820.48 ± 2.11 ms	6473.38 ± 1.16 ms	6.23 / 4.95
3 hops	6991.24 ± 2.71 ms	7673.14 ± 0.92 ms	7.48 / 5.87

Table 6.2: *Average latency of read and write accesses as a function of the distance for the SGI system*

In conclusion, we expect that the optimization of the locality of accesses to main memory has a higher impact on the SGI system and that applications react more sensitively to the optimization of write accesses than to the optimization of read accesses.

6.4 Parametrization and tuning of the benchmarks

In this section, we describe which parameters we have chosen for the benchmarks presented in Section 6.1 and how these parameters have been determined. The second part of the section deals with the sequential optimization of the applications and presents the flags used for compilation as well as manual optimizations of the code.

6.4.1 Parametrization

The parameters for the benchmarks have been chosen carefully with respect to the architectures above. Four criteria were particularly relevant to the parametrization:

- The input size defining whether the entire data set can be held in caches or if accesses to main memory are necessary. The maximal reasonable size should prevent the system from swapping out contents of the main memory to the hard disk.
- The available parallelism depending on the input size and the granularity of tasks defined by the block size.
- The cache hit rate for sequential execution within a task constrained by the block size.
- The wall clock execution time mainly constrained by the size of the data set and the number of iterations for stencil computations.

Some of these criteria are dependent, e.g., the block size both influences the cache hit rate as well as the available parallelism during execution. The methodology we have used for the parameterization is the following. First, we defined the size of the data set. As this thesis focuses on the optimization of accesses to main memory, we have chosen an initial size for the data set for each benchmark that exceeds the overall cache capacity of both test platforms by several orders of magnitude. Next, we defined a number of iterations for the stencil computations, except for *blur-roberts* that always performs only a single iteration. To avoid that the execution of auxiliary tasks represents a significant fraction of the execution time, we set the number of iterations to 60, such that the number of main computation tasks is at least one order of magnitude higher than the number of auxiliary tasks. In the third step, we defined several block sizes and chose the size that yielded minimal execution time among them when executed using the OpenStream run-time with random work-stealing, i.e., without any of the optimizations proposed in Chapter 7 or Chapter 8. Finally, we analyzed the parallelism with our performance analysis tool Aftermath, presented in

	Matrix / Vector size			Block size	Iterations
Seidel	$2^{14} \times 2^{14}$			$2^8 \times 2^8$	60
Jacobi-1d	2^{28}			2^{16}	60
Jacobi-2d	$2^{14} \times 2^{14}$			$2^8 \times 2^8$	60
Jacobi-3d	$2^{10} \times 2^9 \times 2^9$			$2^6 \times 2^6 \times 2^6$	60
Blur-roberts	$2^{15} \times 2^{15}$ (Opteron) / $2^{16} \times 2^{16}$ (SGI)			$2^8 \times 2^8$	-
Bitonic	2^{28}			2^{16}	-
Cholesky	Up to $2^{15} \times 2^{15*}$			$2^8 \times 2^8$	-
	Points	**Block size**	**Dimensions**	**Clusters**	**Fan-out**
K-means	$40,960,000$	$10,000$	10	11	2

* Different matrix sizes are evaluated, see Chapter 9

Table 6.3: *Parameters for the benchmarks*

Chapter 10. If workers were idling during a significant fraction of the measurement interval, we increased the size of the data set until parallelism was sufficiently high.

Table 6.3 summarizes all parameters for the benchmarks. These apply to the dynamic single assignment implementations, the shared memory baseline as well as the sequential baseline. For *seidel*, *jacobi-1d*, *jacobi-2d*, *jacobi-3d* and *bitonic* the data set has a size of 2 GiB (2^{28} double precision floating point elements or 64-bit integers, respectively). For *blur-roberts*, we have chosen matrices with a size of 8 GiB for the Opteron system and 16 GiB for the SGI platform. The main reason for the difference between the size for the Opteron platform and the size for the SGI system is that the $2^{15} \times 2^{15}$ matrix is processed very fast on the SGI system, resulting in a wall clock execution time of far less than one second. Therefore, we have increased the size to $2^{16} \times 2^{16}$ elements on the SGI system. However, we could not use this size on the Opteron platform as the parallel baseline without optimizations for scheduling and data placement starts swapping out contents of the main memory to the hard disk. For *cholesky* we have evaluated different sizes of the symmetric matrix with up to 8 GiB. The *k-means* benchmark is executed with $40,960,000$ points with 10 double precision floating point dimensions, which corresponds to a data set of 3.125 GiB.

The block size yielding the minimal execution time for the majority of the benchmarks is 512 KiB. The only exceptions are *jacobi-3d*, for which a block size of 2 MiB performs best and *k-means* with a block size of 800 kB. The number of clusters and the number of dimensions for *k-means* have been set arbitrarily to 11 and 10, respectively. A value of two for the fan-out of propagations (equal to the fan-in for reductions) provided sufficiently short reduction and propagation phases.

All input data structures are initialized at the beginning of the execution, triggering physical allocation of the memory for the input data. The *seidel* benchmark sets all elements of the matrix to zero, except two elements near the upper left corner and the lower right corner, which are initialized with a value different from zero. As the latter values do not have any significant impact on the execution time we have chosen an arbitrary value of 500. The same procedure for initialization applies to *jacobi-2d*. In the one-dimensional version of the benchmark we initialize all elements to zero except two values at the beginning and the end of the array are, which are set to 500. Similarly, *jacobi-3d* initializes only an element in the upper left corner towards the front and one in the lower right corner towards the rear to 500. The *blur-roberts* application sets all elements to values obtained by calculating the sum of their coordinates within the matrix, i.e. $v_{x,y} = x + y$. For the initialization of the input vector of the bitonic sorting network, we chose to generate random integers using a linear congruential generator.

The initialization of the input matrix for *cholesky* is slightly more complicated. The matrix is first initialized by the `dlarnv` function of the LAPACK interface, generating a random double precision floating point value between between 0 and 1 for each element of the matrix. To make sure that the matrix is positive definite as required for Cholesky Factorization, a diagonally dominant matrix is derived from the random matrix by adding N to the elements on the diagonal, where N is the number of elements in a row.

	Relevant compiler flags	Manual optimizations
Bitonic	`-O2`	-
Cholesky	`-O2 -ffast-math`	Calls to an optimized library (MKL)
Jacobi-1d	`-O3 -ffast-math`	-
Jacobi-2d	`-O3 -ffast-math`	-
Jacobi-3d	`-O3 -ffast-math`	-
K-means	`-O3`	-
Blur-roberts	`-O3 -ffast-math`	-
Seidel	`-O3 -ffast-math`	16× loop unrolling of the innermost loop

Table 6.4: *Compiler flags and manual optimizations for the benchmarks*

The points for the *k-means* benchmark are generated through random walks around 11 randomly chosen cluster centers. Similar to *bitonic*, all random data is generated using a linear congruential generator.

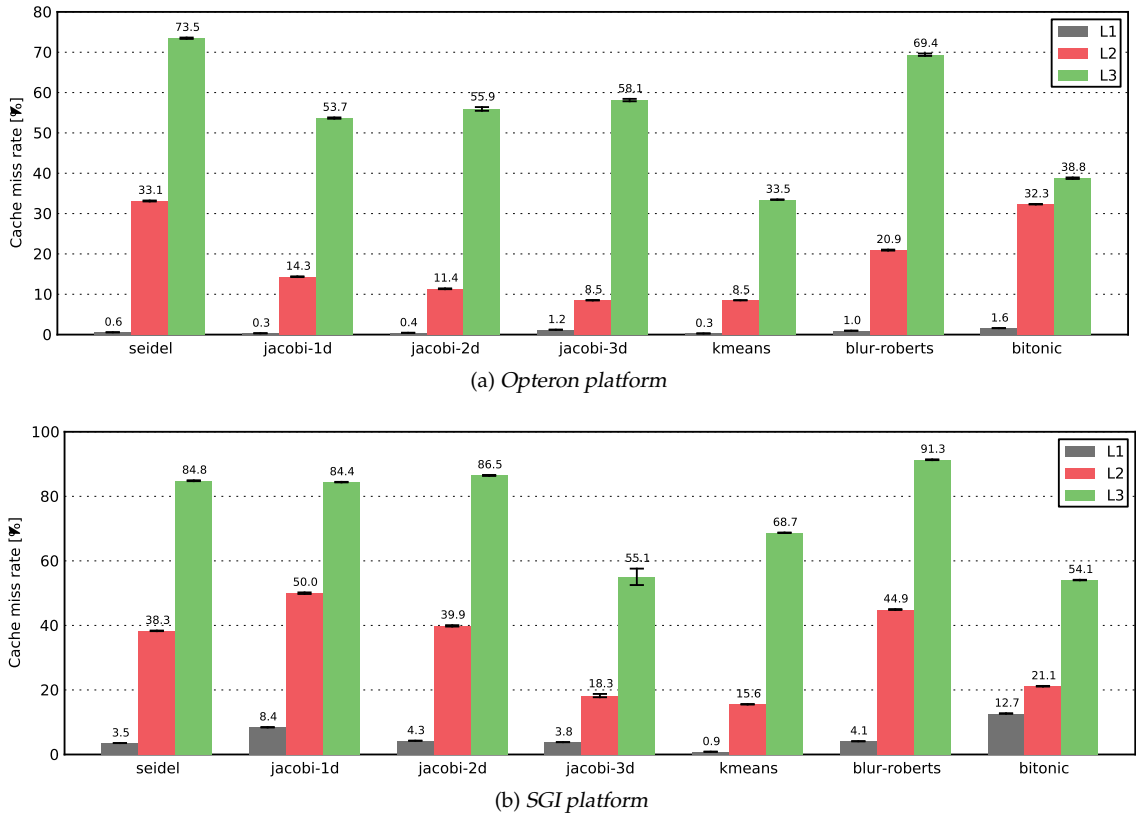
6.4.2 Compiler flags and manual optimizations

All benchmarks have were compiled using the OpenStream compiler based on GCC 4.7.0 [79]. Table 6.4 summarizes the relevant compiler flags and the manual optimizations applied to the source code used on both hardware platforms. The majority of the benchmarks was compiled using aggressive optimizations enabled by the `-O3` flag and using faster floating point operations enabled by `-ffast-math`. In addition, we have unrolled 16 iterations of the innermost loop of the loop nest for the main computation in *seidel*. The *cholesky* benchmark is compiled using the less aggressive `-O2` switch. However, as described in Section 6.1.5, we have configured *cholesky* to use the highly optimized MATH KERNEL LIBRARY for all BLAS and LAPACK functions, such that the compiler flags above only have little influence on the performance of the benchmark. The *bitonic* benchmark has also been compiled using the `-O2` flag as the use of `-O3` did not improve performance.

6.5 Characterization of memory accesses

To understand the impact of data locality on performance in the experimental evaluation of the following chapters, it is important to understand the behavior of the benchmarks with respect to the memory hierarchy. In this section, we provide an overview of the cache miss rates and analyze the frequency of memory accesses of the parallel baseline using dynamic single assignment on both test systems presented above. To characterize the behavior for the execution with maximal parallelism, the number of workers used in the experiments is equal to the number of cores of the systems. The goal of this section is to classify the applications according to the frequency of memory accesses into a set of memory-bound applications that are sensitive to the latency of memory accesses and thus sensitive to data and task placement and a set of cache-bound applications for which placement does not have a significant impact on performance.

Figure 6.20 shows the cache miss rates for the three cache levels of the memory hierarchy of the Opteron and the SGI platform for the execution of the different benchmarks, except cholesky. The reason why we do not provide an analysis of *cholesky* is that this benchmark relies on optimizations for broadcasts that are introduced in Chapter 9. Without these optimizations, the memory footprint is excessively high, such that a presentation of the cache miss rate would not provide insight on inherent characteristics of the benchmark. The frequency of last level cache misses for *cholesky* will be presented in Section 9.4.2. The cache miss rate is defined as the number of cache misses divided by the number of accesses to the cache. Due to the lack of an appropriate counter for the number of accesses to the first level cache on the SGI system, we use the number of load instructions as an approximation for the number of accesses. As this counter does not capture all accesses to the first level cache, it represents an over-estimation of the actual number of accesses. At the first



(a) *Opteron platform*

(b) *SGI platform*

Figure 6.20: *Cache miss rates of the dynamic single assignment versions*

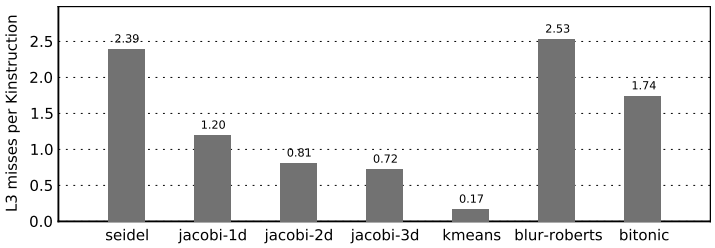
level of the cache hierarchy both machines have separate data and an instruction caches. As we focus on data accesses, we have collected statistics on the first level cache for the data cache only [5]. Each bar of the graph represents the median for 50 executions and error bars indicate the standard deviation.

On the Opteron system, the fraction of accesses to the first level cache generating a miss is below 2% for all of the benchmarks. The miss rates of the second level cache on this platform are significantly higher and range from 8.5% for *jacobi-3d* and *k-means* to more than 30% for *bitonic* and *seidel*. The third level cache has the highest cache miss rates with at least 33.5% for *k-means* to up to 73.5% for *seidel*.

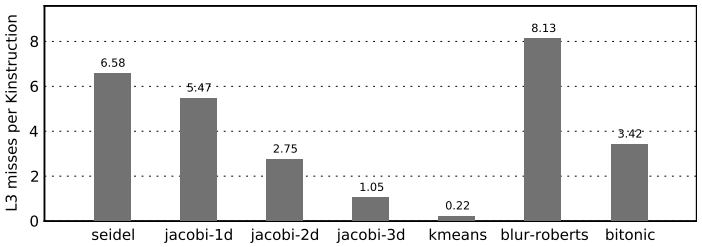
Execution on the SGI system yields higher cache miss rates in general due to the fact that remote memory accesses have a significantly higher latency, resulting in less cache lines that are brought to the cache in time by the hardware prefetching mechanism. The approximated first-level cache miss rates range from 0.9% for *k-means* to as much as 12.7% for *bitonic*. The minimal miss rate for the second level cache is also achieved by *k-means* and reaches its maximum again for *jacobi-1d*. The last level cache miss rates on this system are all higher than 50% and reach more than 80% for four of the benchmarks, namely *seidel*, *jacobi-1d*, *jacobi-2d* and *blur-roberts*.

Although the miss rates of the second level and third level caches are high, the total number of accesses missing all levels remains reasonable due to the low miss rates of the first level cache. However, cache miss rates help characterize the benchmarks with respect to the cache hierarchy and indicate the *fraction* of cache accesses resulting in misses, but do not provide information on

5. The actual PAPI hardware counter names used for data collection are *PAPI_L1_DCM*, *PAPI_L1_DCA*, *PAPI_L2_DCM*, *PAPI_L2_DCA* for the per-core counters of the first and second level as well as the per-northbridge counters *L3_CACHE_MISSES:ALL* and *READ_REQUEST_TO_L3_CACHE:ALL* on the Opteron system and *PAPI_L1_DCM*, *PAPI_LD_INS*, *PAPI_L2_DCM*, *PAPI_L2_DCA*, *PAPI_L3_TCM* and *PAPI_L3_TCA* on the SGI platform.



(a) *Opteron platform*



(b) *SGI platform*

Figure 6.21: *Number of last level cache misses per thousand instructions*

how often accesses to main memory occur. As we focus on memory accesses to main memory in this thesis, we analyze their frequency below.

An access to main memory occurs when the data requested by the access is present in either of the system's cache. Hence, the number of accesses to main memory is equal to the number of last-level cache misses. We define the *frequency of last level cache misses* as the number of last level cache misses per thousand instructions. Figure 6.21 shows the frequency of last level cache misses for each benchmark except *cholesky*, executing on the two test systems. Each bar in Figure 6.21 represents the median number of last level cache misses of the respective benchmark divided by the median number of instructions for the same benchmark for 50 executions [6]. Without exception, all of the frequencies of the SGI system are higher than those of the Opteron system. This is due to the higher cache miss rates on the SGI system, which result in frequent accesses to the last level cache and hence a higher absolute number of misses.

A comparison of the frequencies of all benchmarks reveals that the characteristics are approximately the same on both platforms. Among all benchmarks, *blur-roberts* generates the highest number of last level cache misses per thousand instructions, followed by *seidel*. The ranking of *bitonic* and *jacobi-1d* differs between the systems: while *bitonic* has a higher frequency of last level cache misses than *jacobi-1d* on the Opteron platform, *jacobi-1d* generates more misses per instruction than *bitonic* on the SGI system. Furthermore, *jacobi-2d* and *jacobi-3d* yield approximately the same values on the Opteron system, while the frequencies of these benchmarks differ significantly on the SGI system. However, on both platforms *k-means* has by far the lowest frequency of last level cache misses among all benchmarks.

Hence, we expect *k-means* to be the benchmark with the least sensitivity to data and task placement and *seidel*, *blur-roberts* and *bitonic* to be among the benchmarks with the highest sensitivity. For the benchmarks in between the categorization is less clear.

6.6 Scalability of NUMA-agnostic shared memory benchmarks

To support our choice in Section 6.2.2 to use interleaved allocation for global data structures in shared memory, we analyze the scalability of the shared memory baseline with and without

6. The PAPI counter measuring the number of executed instructions on both platforms is *PAPI_TOT_INS*.

interleaving. Figure 6.22 shows the speedup over the sequential baseline of the benchmarks on the Opteron system for an increasing number of workers used for execution [7]. Due to the long execution time for a low number of workers, each data point represents the median of only 10 executions. As for the previous graphs, error bars indicate the standard deviation. The workers in the experiments have been assigned to cores in increasing order of logical processor identifiers, resulting in placements where all cores of a NUMA node are used before workers are placed on neighboring nodes. For example, a configuration with eight workers uses the cores with the identifiers 0 to 7, which all belong to the first NUMA node. To maximize the available bandwidth, interleaved allocation always involves all NUMA nodes independently from the number of cores.

The graphs for the Opteron system show that the maximum speedup over sequential execution without interleaving is reached when using six or seven workers for most of the benchmarks. For a higher number of workers, the performance either remains approximately the same (*seidel*, *jacobi-2d*, *jacobi-2d*, *blur-roberts*, *bitonic*) or performance drops (*jacobi-1d*). The *k-means* benchmark is the only application that scales beyond seven workers and achieves higher performance with each additional core. However, neither benchmark performs better with the default data placement scheme than with interleaved allocation for more than 24 cores. Using interleaved allocation scaling is near linear, although the slope of the curves is below a preferable slope of one.

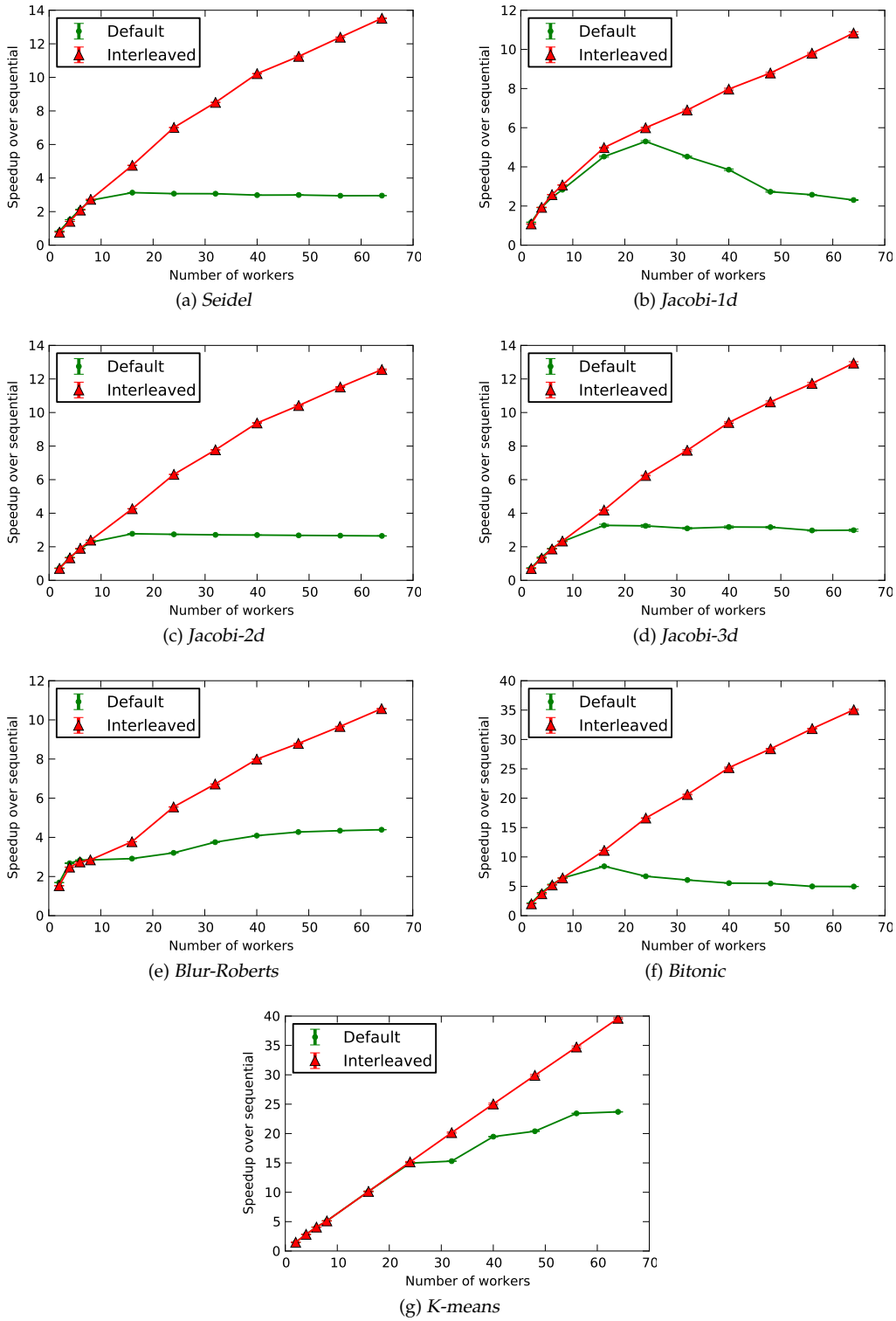
The results for the SGI system shown in Figure 6.22 are similar. With interleaved allocation, each additional core increases performance, except for *blur-roberts* whose performance drops for more than 128 cores. Though, the performance increase by each core becomes lower for a high number of cores for most of the benchmarks. An interesting observation is that for all benchmarks except *k-means* the default allocation scheme performs better than interleaved allocation for up to eight cores. This is due to the placement of workers, which associates the first eight workers to cores of the same node. With the default allocation scheme, all memory accesses are local, while most of the accesses are remote using interleaved allocation. However, as soon as the number of workers exceeds the number of cores per node, reduced contention compensates the reduced locality of interleaved allocation, resulting in higher performance compared to the default method for memory allocation. Another interesting result is that the performance of *blur-roberts* using interleaved allocation decreases when using more than 112 cores. Similarly, the performance of *seidel* drops for 160 cores, but reaches the maximum for 192 cores. In the case of *blur-roberts*, we suspect that the high number of requests to main memory creates contention on all memory controllers. An explanation for the performance drop of *seidel* for 160 cores seems more complicated. However, a detailed analysis of this behavior is out of the scope of this analysis. As the majority of benchmarks provides the highest performance when using all cores of the machine, the experiments for the evaluation use 64 and 192 cores, respectively.

The conclusion of the results above is that interleaved allocation significantly improves the scalability of the shared memory baseline and should therefore be used by default. This confirms our choice to use interleaved allocation for data structures in shared memory in our experiments.

6.7 Summary

In this chapter, we provided an overview of the applications and the hardware environment used for the evaluation of the concepts presented in this thesis and introduced the methodology used for measuring the execution time as well as the quantification of micro-architectural events based on hardware performance counters. The set of applications consists of eight high performance scientific benchmarks covering stencil computations (*seidel*, *jacobi-1d*, *jacobi-2d*, *jacobi-3d*, *blur-roberts*), integer sorting (*bitonic*), clustering (*k-means*) and linear algebra kernels (*cholesky*). All of the benchmarks except *cholesky* have been implemented as a sequential application, as a parallel application using dynamic single assignment and as a parallel application using shared memory programming with token synchronization. The *cholesky* benchmark has only been implemented using dynamic single assignment and will be compared to state-of-the-art parallel linear algebra

7. As *cholesky* has not been implemented using shared memory and token synchronization the graphs do not include results for this benchmark.



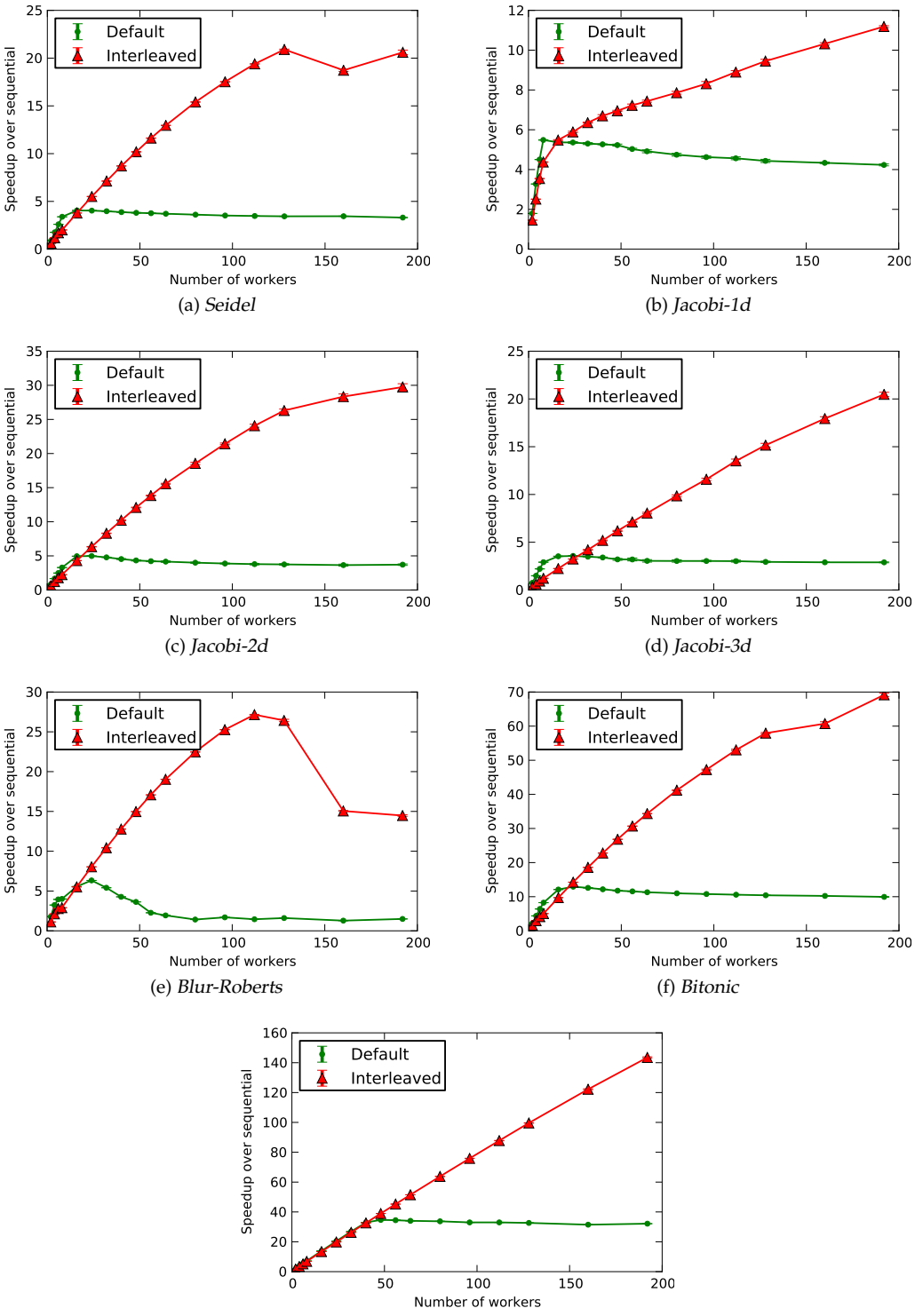
(a) *Seidel* (b) *Jacobi-1d*

(c) *Jacobi-2d* (d) *Jacobi-3d*

(e) *Blur-Roberts* (f) *Bitonic*

(g) *K-means*

Figure 6.22: *Scalability of shared memory benchmarks (Opteron platform with 64 cores)*



(a) *Seidel* (b) *Jacobi-1d*

(c) *Jacobi-2d* (d) *Jacobi-3d*

(e) *Blur-Roberts* (f) *Bitonic*

(g) *K-means*

Figure 6.23: *Scalability of shared memory benchmarks (SGI platform with 192 cores)*

codes in Chapter 9. For the dynamic single assignment versions, we provided details on the partitioning of work into tasks and the parallel control program as well as on the use of streams. Wherever possible, we kept the same partitioning, the same data structures and the same control programs for the shared memory baseline. A basic characterization of the behavior of the benchmarks with respect to the memory hierarchy showed that some benchmarks are likely to be more sensitive to remote memory accesses than others. Finally, we confirmed our choice to use interleaved allocation for all data structures in shared memory by analyzing the results for the scalability of the applications of the shared memory baseline.

The next two chapters introducing data-aware scheduling and deferred allocation build on the information of this chapter as they both refer to the applications and the hardware environment described above and make use of the measurement methodology.

7 Data-aware scheduling

In the previous chapters, we laid the foundations of data-aware scheduling by providing accurate and efficient methods to determine the placement of data and by constraining the structure of streaming applications, such that the working set of the majority of tasks is known before their execution. In this chapter, we show how this information can be used to improve the locality of accesses to main memory with respect to NUMA. We first analyze the effects of the default scheme for task activation on data locality and identify possible causes for accesses to remote memory. Based on these observations, we introduce *work-pushing*, a data-aware and topology-aware mechanism for task transfers between workers that is triggered at task activation. The goal of this technique is to avoid a mismatch of task ownership and the ownership of data and thus to favor execution of tasks on the nodes containing their data. We then analyze the influence of random work-stealing on data locality with matching task and data ownership and introduce *topology-aware work-stealing*, which attempts to steal tasks in a worker's, incrementally widening, neighborhood based on an abstract, static description of the memory hierarchy. We show that work-pushing and topology-aware work-stealing are complementary techniques that improve the locality of accesses to main memory significantly compared to the default task activation scheme and random work-stealing. For memory-bound applications, the increase of data locality results in a significant improvement of performance. *Parts of this chapter were previously published in [46].*

7.1 The influence of task activation on data locality

A task is activated when all of its producers have terminated and all of its consumers have been created. To illustrate the different scenarios for data locality resulting from task activation, we use the most general case for task dependences presented in Figure 7.1, showing a task t that depends on n producers p_0 to p_{n-1} and that provides input data for m consumers c_0 to c_{m-1}. The amount of data read from each producer is indicated by δ_i^0 to δ_i^{n-1} while δ_o^0 to δ_o^{m-1} refer to the amounts of output data of t written to the consumers. The task that creates t is labeled t_c and tasks $t_{c,o}^0$ to $t_{c,o}^{m-1}$ are the creators of the consumers of t. The figure also indicates for each task by which worker it is executed: the producers are executed by w_i^0 to w_i^{n-1}, t is executed by w_t, t_c is executed by w_c and the creators of the consumers are executed by $w_{c,o}^0$ to $w_{c,o}^{m-1}$. Note that the identifiers for the workers are only used as a shorthand to refer to the workers executing the task and different identifiers do not necessarily imply different workers. Throughout the entire chapter, we assume that the memory allocation techniques presented in Chapter 4 are used and that the size of each input buffer exceeds the threshold defined in Section 4.4.1, such that the placement of each buffer

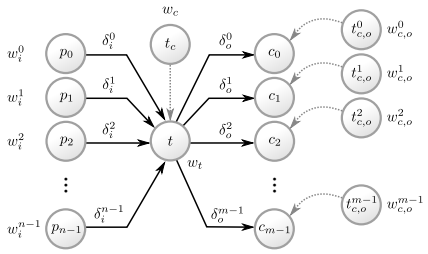


Figure 7.1: *A task with n producers and m consumers*

is determined by the run-time system. All applications are assumed to implement dynamic single assignment on streams.

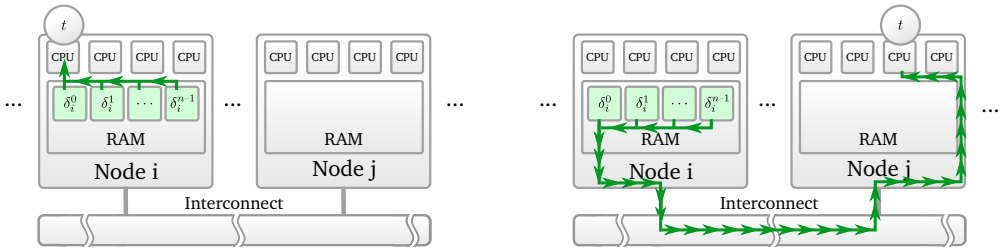
As stated in Section 3.4.1, the execution model of OpenStream defines that a task which becomes ready for execution is added to the single entry software cache of the worker that satisfies its last dependence. This is the worker that executed the producer which finished last among all producers of the task and which decremented the task's synchronization counter to zero. Unless the newly activated task is transferred to the worker's work-deque due to the activation of another task by the same worker and unless it is stolen by another worker, the task will also be executed by the worker that satisfied its last dependence. We say that a worker *owns* a task if the task is either in the worker's single entry software cache or in the worker's work-deque. For the task graph of the example, this means that the worker that owns t and which is likely to execute t afterwards is defined by the order of termination of p_0 to p_{n-1}. Depending on the timing at execution this could be any of the workers w_i^0 to w_i^{n-1}.

7.1.1 The locality of read accesses

All of the input buffers of a task are allocated at task creation and originate from the memory pool of the creating worker. Due to the use of per-node memory pools presented in Section 4.4.2, buffers allocated from the same memory pool are generally placed on the same node, which corresponds to the node associated to the memory pool. Hence, the input buffers of a task are all placed on the node of the creating worker. In the example, all of the input buffers of t are thus placed on the node associated to w_c. Whether access to these buffers during execution of t targets local or remote memory depends on the core on which w_t executes. If the core is associated to the same node as w_c, the accesses to input data of t all target local memory, as shown in Figure 7.2a. If w_t is a worker that executes on a CPU that is associated to a different node than the node of w_c, the accesses target a remote node, as shown in Figure 7.2b. However, this last configuration becomes more likely for a higher number of nodes and thus a higher number of workers if we assume that the probability for the execution of a producer task is equal for all workers. The default task activation scheme does neither have control over the timing of the execution of the producers nor does it define the set of workers w_i^0 to w_i^{n-1}. Thus, local accesses are the result of favorable circumstances at execution time.

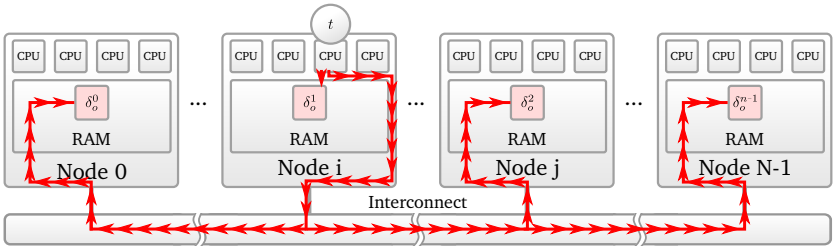
7.1.2 The locality of write accesses

The placement of the buffers receiving the output data of t can be more complex than the placement of input buffers. As the consumers are potentially created by different workers operating on different nodes, the input buffers of the consumers, which serve as the output buffers for t, can also be placed on different nodes. Hence, not only the worker executing t, but also the distribution of output buffers define whether write accesses during execution of t target local or remote memory. For a distribution of output buffers on more than one node a part of the write accesses of t inevitably targets a remote node, independently from the worker that executes t.

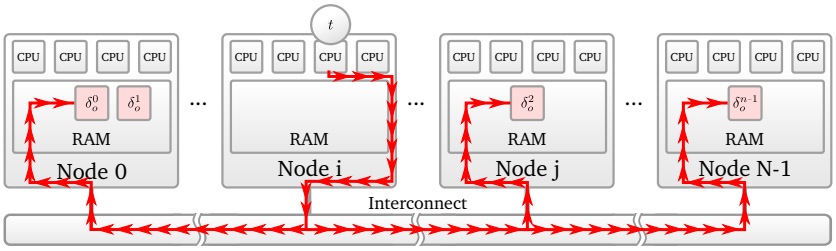


(a) *Local read accesses* (b) *Remote read accesses*

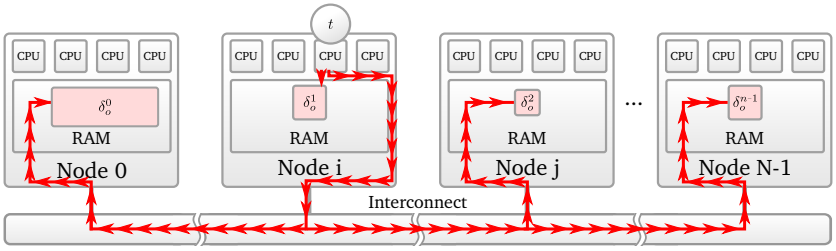
Figure 7.2: *Remote / local memory accesses to input buffers depending on activating worker*



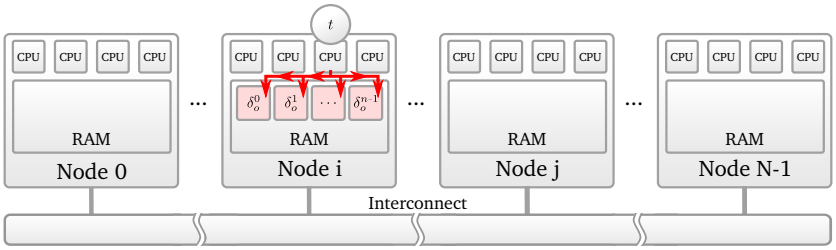
(a) *Remote accesses to multiple nodes, equal-sized buffers*



(b) *Imbalance between nodes due to buffers placed on the same node*

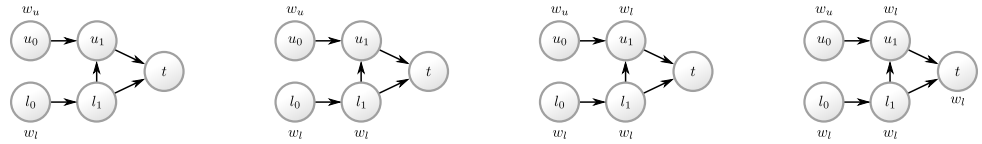


(c) *Imbalance between nodes due to a varying buffer size*



(d) *Local accesses only*

Figure 7.3: *Remote / local write accesses depending on the placement of output buffers*



(a) *Execution of u_0 and l_0* (b) *Execution of l_1* (c) *Execution of u_1* (d) *Execution of t*

Figure 7.4: *Different probabilities among workers for task ownership*

(a) *Remote read accesses* (b) *Local read accesses*

Figure 7.5: *Influence of task creation on the locality of read accesses*

Figure 7.3 illustrates different cases for the locality of write accesses resulting from the placement of the input buffers of c_0 to c_{m-1}. In Figure 7.3a, only one of the consumers' input buffers is placed on the node of the worker executing t and the large majority of write accesses targets remote memory independently from the node on which t is executed. For example, if t were executed on Node 0, access to the input buffer of c_0 would be local, but accesses to the input buffer of c_1 would be turned from local into remote accesses. Figure 7.3b and Figure 7.3c show different situations in which output buffers are also distributed over multiple nodes, but in which some of the nodes contain more output data than others. The imbalance of the distribution in Figure 7.3b results from the placement of multiple output buffers on the same node, while the imbalance in Figure 7.3c is due to the varying size of output buffers. Hence, some nodes are more favorable for execution than other as they contain a larger fraction of the memory regions for output. However to maximize data locality, ideally all output buffers of t are placed on a single node and t is executed on a core of the same node as in Figure 7.3d. This requires that all of the workers $w_{c,o}^0$ to $w_{c,o}^{m-1}$ as well as w_t are workers operating on cores of the same node. For higher numbers of nodes and workers this becomes less likely if the probability of execution of a task is equal for each worker.

As already mentioned above for input buffers, the default task activation scheme neither takes into account data placement nor does it control data placement actively. As a result, the fraction of local accesses to main memory heavily depends on the application and on parameters determined at execution time.

7.1.3 The influence of the task graph on task ownership

In the discussion above we assumed that the probability of a task to be executed by a worker is equal for each worker. However, in practice, the probability of task ownership often varies between the workers that execute the task's producers, as the order of execution of the producer tasks can be influenced by the structure of the task graph. Consider the task graphs shown in Figure 7.4a with five tasks u_0, u_1, l_0, l_1 and t. Assume that u_0 and l_0 are executed in parallel by two different workers w_u and w_l and have the same duration. At the end of their execution l_1 becomes ready and w_l becomes the owner of l_1 as shown in Figure 7.4b. The consumer of u_0 also depends on data from l_1 and therefore cannot become ready immediately at termination of u_0. At termination of l_1 the only worker that can become the owner of u_1 is w_l (Figure 7.4c). Thus, although t depends on two tasks it can only be activated by w_l (Figure 7.4d) and the probability of task ownership for w_u is zero.

Depending on the relationships of task creation and the initial distribution of tasks to workers,

the probabilities for task ownership resulting from the structure of the task graph can lead to remote or local memory accesses. For example, if w_u and w_l operate on different nodes and if the task creation in the task graph of the previous example is carried out as in Figure 7.5a, where u_0 creates t, read accesses during execution of t are guaranteed to target remote memory. In contrast to this, read accesses are guaranteed to be local if l_0 creates t as shown in Figure 7.5b. This would also be the case if w_u and w_l operated on the same node.

The implications of the structure of the task graph on task ownership and locality can be complex for sufficiently large task graphs. In addition, they depend on the initial distribution of tasks to workers as well as on the architecture of the system executing the application. Hence, it is generally impossible to control the locality of memory accesses through modification of the structure of a parallel application in this execution model.

7.1.4 Conclusion

The default method for task activation causes the worker that satisfies the last dependence of a task to become its owner, independently from the placement of input and output buffers of the newly activated task. This often leads to a mismatch between the ownership of tasks and the ownership of data, resulting in accesses to remote memory during execution of tasks. For data locality, ideally task and data ownership match and each task is executed on the node that contains its data or, if its data is distributed over multiple nodes, on the node with the minimal average distance to the data. In the following section, we introduce *work-pushing*, a task activation mechanism that improves data locality by transferring a task to a worker of an appropriate node.

7.2 Work-pushing

Improving the fraction of memory accesses targeting local memory is an important step to reduce the average latency of memory accesses and thus to increase performance. From this perspective, it is crucial to reduce the mismatch of task and data ownership explained in the previous section. To this end, we propose *work-pushing*, which transfers a task that becomes ready for execution to an appropriate worker based on information about the working set of the task derived from dynamic single assignment as well as information on the placement of this data. The selection of the target worker for such a transfer is essential for the locality of accesses to memory during task execution. Based on the observations of the previous section, we propose three simple heuristics for this selection:

- The first heuristic, which we refer to as *input only*, avoids a mismatch between task ownership and the ownership of input data by pushing a task to a worker operating on the node that contains the task's input buffers.

- The *output only* heuristic tries to avoid a mismatch between task ownership and the ownership of output buffers. However, as there can be several nodes containing the output buffers of a task, it might be necessary to choose between multiple nodes. To take into account these situations, the heuristic estimates the access time for each node and each output buffer and chooses a worker from the node with the smallest overall access time.

- Finally, we also evaluate a heuristic named *weighted*, which acts similarly to the *output only* heuristic, but which also includes input buffers when determining the most appropriate worker to reduce the time spent on memory accesses.

Efficient lock-free work-stealing deques, as the one used for work-stealing in OpenStream [37], cannot be used to *remotely push* tasks without changing the algorithm and incurring high synchronization costs. As a solution to this problem, the run-time can employ an additional work-sharing mechanism that uses a data structure dedicated to task transfers, such as a multi producer, single consumer FIFO queue (*MPSC FIFO*). In this solution, each worker is provided with an MPSC FIFO

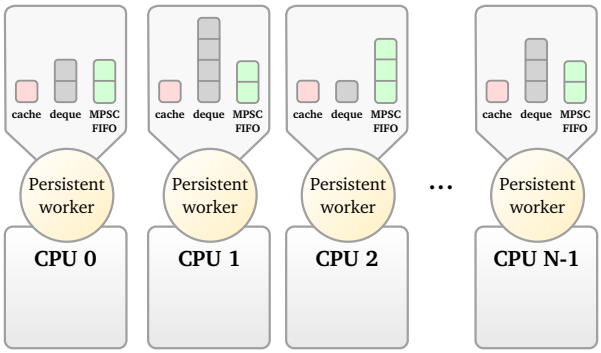


Figure 7.6: *Updated structure of the workers with MPSC FIFO*

whose only purpose is to receive tasks from remote workers and which does not interfere with the single entry software cache or the work-deque. Figure 7.6 shows the updated set of structures associated to each worker. When a task needs to be transferred from the activating worker to the target worker, it is simply added to the target worker's MPSC FIFO. The target worker checks after each execution of a task if new tasks have been added to the MPSC FIFO and transfers them to its single entry software cache and its work-deque.

Algorithm 3 shows how a worker w discovering that a task t is ready for execution transfers the task to another worker if necessary. This procedure called *last_dep_satisfied* is composed of four parts explained below.

In the first part from lines 3 to 42, the actual placement of input and output buffers of the task is determined and weighted according to the heuristic. Input views are only taken into account if the selected heuristic is either the *input only* heuristic or the *weighted* heuristic. For each of the input views the procedure first determines the node that contains the associated input buffer by calling *node_of* with the data pointer of the view in line 10. This function looks up the placement from the metadata cache of the input buffer introduced in Section 4.4.1. If the placement is known, the input buffer is taken into account and its size is added to the element of an array called *data*, which stores the amount of data for each node (line 15). The size of the view is also added to *data_size* in line 18, even if the placement could not be determined. This variable stores the overall size of the task's data relevant to the heuristic and is used to estimate whether a transfer to a remote worker is likely to be beneficial or not. Output views are taken into account if the selected heuristic is the *output only* or *weighted* heuristic. The statements of lines 25 to 42 are similar to the statements for input views with the exception that the *weighted* heuristic uses a weight of two when counting the amount of output data on a node. The reason for this choice is that write accesses to main memory are usually more critical for performance than read accesses as shown in Section 6.3.3. In case of an equal number of bytes of input and output data on different nodes, the *weighted* heuristic thus prioritizes the node for the output data.

In the second part in lines 43 to 47, the procedure estimates whether a transfer of the task to a remote worker is beneficial or not and, if this is the case, determines to which node the task should be transferred. To this end, the variable *data_size* is compared with an empirically determined threshold of 10 kB of relevant data [1]. The purpose of this check is to avoid that tasks with a small working set are transferred to remote workers, incurring an overhead for the transfer of the task without improvements on performance due to improved locality of data accesses. If the value of *data_size* is below the threshold, the task is added to the single entry software cache of the worker by calling *add_task_locally* (presented in Algorithm 2 of Section 17) and the procedure returns immediately.

1. For the benchmarks studied in this thesis the threshold of 10 kB is superior to the size of the working set of tasks that only access small buffers whose placement does not have an impact on performance. Whether this value is appropriate in general is to be determined with a broader range of applications.

Algorithm 3: last_dep_satisfied(w, t)

```
1  node_w ← local_node_of_worker(w)
2  data[0...N−1] ← ⟨0,...,0⟩
3  data_size ← 0
4
5  if heuristic = input only or
6     heuristic = weighted
7  then
8      for v ∈ t.input_views
9      do
10         n_d ← node_of(v.data)
11         s ← v.horizon
12
13         if n_d ≠ unknown
14         then
15             data[n_d] ← data[n_d] + s
16         end
17
18         data_size ← data_size + s
19     end
20 end
21
22 if heuristic = output only or
23    heuristic = weighted
24 then
25     for v ∈ t.output_views
26     do
27         n_d ← node_of(v.data)
28         s ← v.horizon
29
30         if n_d ≠ unknown
31         then
32             if heuristic = weighted
33             then
34                 s ← 2 · s
35             end
36
37             data[n_d] ← data[n_d] + s
38         end
39
40         data_size ← data_size + s
41     end
42 end
43 if data_size < 10 kB
44 then
45     add_task_locally(t, w)
46     return
47 end
48
49 node_min ← node_with_min_cost(node_w, data)
50
51 if node_min ≠ node_w
52 then
53     w_dst ← random_worker_on_node(node_min)
54     res ← push_back(w_dst.mpsc_fifo, t)
55
56     if res = failure
57     then
58         add_task_locally(t, w)
59     end
60 else
61     add_task_locally(t, w)
62 end
```

Algorithm 4: node_with_min_cost($node_w$, *data*)

```
1  costs[0...N-1] ← ⟨0,...,0⟩
2
3  for src ∈ {0,...,N-1}
4  do
5      for dst ∈ {0,...,N-1}
6      do
7          costs[n] ← costs[n]+
8                      data[dst] · cost(src, dst)
9      end
10 end
11
12 cost_min ← costs[node_w]
13 node_min ← node_w
14 other_min ← ⟨0,...,0⟩
15 num_other_min ← 0
16
17 for n ∈ {0,...,N-1}
18 do
19     if costs[n] = cost_min
20     then
21         other_min[num_other_min] ← n
22         num_other_min ← num_other_min + 1
23     end
24
25     if costs[n] < cost_min
26     then
27         cost_min ← costs[n]
28         node_min ← n
29         num_other_min ← 0
30     end
31 end
32
33 if node_min ≠ node_w and num_other_min ≠ 0
34 then
35     other_min[num_other_min] ← n_min
36     num_other_min ← num_other_min + 1
37
38     r ← rand(num_other_min − 1)
39     node_min ← other_min[r]
40 end
41
42 return node_min
```

Algorithm 5: scheduler_loop(w)

```
1  while true do
2      t ← w.cached_task
3      if t = null then
4          t ← pop_bottom(w.deque)
5      end
6
7      if t = null then
8          t ← steal_task()
9      end
10
11     if t ≠ null then
12         execute_task(t)
13     end
14
15     empty_mpsc_fifo(w)
16 end
```

Algorithm 6: empty_mpsc_fifo(w)

```
1  import ← true
2  while import = true do
3      t ← pop_front(w.mpsc_fifo)
4
5      if t ≠ null then
6          add_task_locally(t, w)
7      else
8          import ← false
9      end
10 end
11
12
13
14
15
16
```

If the check is passed, execution reaches the third part of the procedure, which consists in a call to *node_with_min_cost* with the array holding the per-node values for the data size in line 49. The details of this function are given in Algorithm 4. The return value of the call, which represents the node with the minimal overall access time to the data, is assigned to $node_{min}$.

In the last part, the procedure checks if a transfer to a remote node using the task transfer mechanism involving the MPSC FIFO is needed. This is the case if $node_{min}$ is different from the node of the worker that activated the task (Line 51). As there are generally multiple workers per node, the procedure must first select a worker from the target node. This is done in line 53 by choosing the worker randomly among the workers of the node based on a uniform distribution. Once the target worker is known, the actual transfer is triggered in line 54. If this transfer fails, e.g., due to a full MPSC FIFO of the target worker, the worker activating the task becomes the owner and the procedure returns (line 58). The same happens if the target node is the local node of the activating worker, as this does not require the use of the MPSC FIFO and can be carried out using the worker's own single entry software cache and work-deque.

The algorithm that returns the node with the lowest estimated overall access time, *node_with_min_cost*, is presented in Algorithm 4. Its arguments are the identifier $node_w$ of the node associated to the calling worker and the array *data* with an entry for each node, specifying how much data is located on that node. The function starts with the calculation of an estimation for the access cost for each node in lines 3 to 10. The base for this calculation is the array *data* and a function called *cost*, which indicates the cost of a data transfer between two nodes. In our experiments, this architecture-specific function is modeled according to the distance matrix reported by the NUMACTL tool, which indicates for each pair of nodes the distance and thus an approximation of latency of data transfers the between the nodes. After having calculated the access costs, the node with the minimal cost is selected. The variables $score_{min}$ and $node_{min}$ are initialized with the score and the identifier of the local node, respectively. This ensures that the local node is selected by default if is has the minimal score and there are other nodes with the same score. The array $other_{min}$ stores the nodes that have the same score as the current minimum and num_other_{min} indicates how many of these nodes exist. Their values are updated throughout the execution of the loop in lines 17 to 31. Afterwards, the node with the minimal cost is elected among the nodes from $other_{min}$, by selecting one of them randomly using a uniform distribution (lines 33 to 40). Choosing this node randomly ensures that the node identifier does not have an influence on the election and avoids favoring specific nodes. For example, if $other_{min}$ were traversed sequentially to find the minimal value, nodes with lower identifiers would be more likely to be elected.

Tasks pushed to a remote worker by *last_dep_satisfied* are stored in the MPSC FIFO of the target worker and are thus ineligible by the scheduler as they are neither in the software cache nor in the work-deque. Hence, in order to be executed eventually, they need to be transferred to the software cache and the work-deque. Algorithm 6 illustrates how this simple transfer is done: while the MPSC FIFO is not empty, the front element is removed and added to the work-deque. Using this order has an important side effect. The front of the MPSC FIFO contains the oldest tasks while the back holds the most recent tasks. Thus, during the last iteration of the loop, the most recent task is added to the software cache and becomes the next task to be executed by the worker. Input data of the most recent task has the highest probability to be still present in caches near to the executing core in the memory hierarchy. The last task from the MPSC FIFO is therefore a good candidate for execution. As pushes to a worker can occur at any time, the MPSC FIFO needs to be emptied regularly. This behavior is achieved by adding a call to *empty_mpsc_fifo* to the scheduler loop of each worker, e.g., such as in Algorithm 5, where the function is called each time before the worker selects a new task for execution.

7.3 Topology-aware work-stealing

Work-pushing is triggered when a task becomes ready and transfers the task to an appropriate worker. Work-stealing acts as a complementary technique, which provides workers that run out

of work with tasks ready for execution and hence ensures global computational load balancing. The default random work-stealing strategy neither takes into account the working set of tasks nor the topology of the machine. The property of matching task ownership and data placement from work-pushing could be exploited by the work-stealing algorithm to steal a task whose data is near the node of the stealing worker.

From a core's perspective, the nodes of the machine are reachable at different distances, e.g., at one hop, two hops and so on. Hence, depending on the topology of the machine, the distinction between local and remote nodes can be too coarse when stealing a task. For example, if two workers executing on two different nodes both have tasks available and the first worker is at a distance of a single hop and the other at a distance of two hops, the worker whose node is at a lower distance should be favored for a steal as data transfers from this node and to this node have a lower latency. Similarly, within a node, it might be beneficial to prioritize steals from neighboring workers sharing the same cache as the stealing worker. As data of the task to be stolen may already be present in the shared cache, data accesses during its execution may be faster.

Defining the inter-node and intra-node priorities requires knowledge of the topology of the machine. Before we discuss the topology-aware work-stealing algorithm, we present a lightweight static model for the representation of the memory hierarchy, which is used by the work-stealing algorithm in order to adapt to the topology of the target machine. This model can either be provided by the system administrator or the manufacturer or it could be generated automatically using a tool such as HWLOC [30]. The description can be broken down to the following parts:

- A set $C \subset \mathbb{N}_0$ of identifiers for processing units, e.g. $C = \{0, \ldots, 63\}$.
- An ordered set L containing the levels of the memory hierarchy from the cache nearest to the CPUs down to the different NUMA domains or memory controllers, e.g. $L = \langle L1, L2, L3, RAM \rangle$.
- A function *sibs* : $L \times C \to \mathbb{N}$ describing how many processing units share an instance of a hardware part at a given level. We refer to these processing units as *siblings*. For example, if four cores with identifiers 8, 9, 10 and 11 share a third level cache then $\mathit{sibs}(L3, 8) = \mathit{sibs}(L3, 9) = \mathit{sibs}(L3, 10) = \mathit{sibs}(L3, 11) = 4$. Passing a processing unit as a parameter to *sibs* allows the definition of asymmetric architectures. For example, in the architecture of the SGI test system presented in Section 6.3.2 the number of siblings at a distance of two hops varies between the cores of the different nodes.
- A function *nth_sib* : $L \times C \times \mathbb{N} \to C$ describing which processing unit is the nth sibling of another processing unit in ascending order of CPU identifiers at a given level, e.g. if processing units 0 to 7 share a third level cache, $\mathit{nth\_sib}(L3, 0, 2) = 2$ and $\mathit{nth\_sib}(L3, 3, 2) = 5$.

By using L, *sibs*, *nth_sib* and the order relation on L the neighbors of a core at the different levels of the memory hierarchy can be determined easily for topology-aware work-stealing.

The idea behind our optimized heuristic for work-stealing is that instead of randomly selecting a steal victim, task deques of neighboring workers are favored, which leads to more local memory accesses when the stolen task is executed. However, the work-deques of close workers might not always provide enough tasks to steal. To avoid poor load balancing, other attempts at higher levels in the memory hierarchy must be performed if work-stealing fails on close deques.

Our topology-aware work-stealing technique is shown in Algorithm 7. At each level l beginning with the level nearest to the CPU, a number of steal attempts defined by *attempts*(l) is performed until an attempt is successful or no level is left. In addition to the definitions of the memory hierarchy, the algorithm uses the following notations and functions:

- $\mathit{rand}(n)$ generates a random integer value in $[0; n]$ using a uniform distribution
- *cpu* : $W \to C$ returns the processing unit a worker executes on with W being the set of workers
- *attempts* : $L \to \mathbb{N}_0$ defines the maximal number of steal attempts at a given level of the memory hierarchy
- *steal_task* : $W \to T_\infty \cup \{\mathit{null}\}$ is a function that performs a steal attempt on a target deque and returns the stolen task from the set of tasks T if the attempt is successful or null if the

Algorithm 7: topology_aware_stealing(w)

```
1  cpu_w ← cpu(w)
2
3  for level ∈ L do
4      num_siblings ← sibs(l, cpu_w)
5
6      for attempt ← 1 to attempts(l) do
7          n ← rand(num_siblings − 1)
8          target_cpu ← nth_sib(l, cpu_w, n)
9
10         if target_cpu ≠ cpu_w then
11             target_worker ← cpu^{-1}(target_cpu)
12             t ← steal(target_worker)
13
14             if t ≠ null then
15                 return t
16             end
17         end
18     end
19 end
20 return null
```

attempt fails

7.4 Experimental results

To evaluate the effectiveness of work-pushing and topology-aware work-stealing, we have implemented these techniques in the OpenStream runtime and measured the impact on the locality of data accesses as well as the impact on performance. As topology-aware work-stealing relies on work-pushing, it can only be evaluated together with work-pushing. To differentiate between the configurations, we use the following acronyms: *rnd* for plain random scheduling without work-pushing and without topology-aware work-stealing, *input only* for work-pushing with the *input only* heuristic, *output only* for work-pushing with the *output only* heuristic and *weighted* for work-pushing with the *weighted* heuristic. Combinations of work-pushing and topology-aware work-stealing are indicated by the name of the push heuristic followed by a plus sign and *taws*, i.e., *input only+taws*, *output only+taws* or *weighted+taws*.

7.4.1 Metrics for evaluation

We define multiple metrics to evaluate the locality of data accesses, as well as the performance of the applications for the different configurations. All of them are based on data collected by the run-time during execution of the benchmarks. The main data sources are statistics about data placement obtained from the operating system as described in Section 4.4.1 and samples from hardware performance counters.

Measuring data locality in terms of NUMA

A simple way to measure data locality in terms of NUMA is to configure a set of hardware performance counters for the appropriate events and to count the number of events during the interval of interest. On the Opteron platform, we have configured counters for two northbridge

events, indicating requests to local memory [2] and requests to the memory of a remote node [3], respectively. The northbridge is shared between sets of 8 cores forming a node, such that the events only have to be configured once for each node. Let N^i_{loc} be the number of requests to the local memory controller and let N^i_{rem} be the number of requests to remote memory issued by the cores of node i. We define the *locality of requests to main memory* R_{loc} as the ratio between the number of local requests to the total number of requests for the set of NUMA nodes $\mathcal{N}$ as:

$$R_{\text{loc}} = \frac{\sum_{i \in \mathcal{N}} N^i_{\text{loc}}}{\sum_{i \in \mathcal{N}} N^i_{\text{loc}} + \sum_{i \in \mathcal{N}} N^i_{\text{rem}}} = \frac{\sum_{i \in \mathcal{N}} N^i_{\text{loc}}}{\sum_{i \in \mathcal{N}} \left(N^i_{\text{loc}} + N^i_{\text{rem}} \right)}$$

Due to missing support for the appropriate counters in the kernel version used on the SGI system, we can provide the results for this metric only for the Opteron test system. However, it is possible to approximate R_{loc} based on information about the working set of tasks and page placement available in the OpenStream run-time. Let core_node : $C \rightarrow \mathcal{N}$ be a function that associates each core with the identifier of the core's NUMA node and let node_of : $\mathcal{A} \rightarrow \mathcal{N}$ be the function used earlier for work-pushing that returns the identifier of the node containing the data of the input buffer whose address is specified as the parameter. The function is_local : $W \times A \rightarrow \{0, 1\}$, indicating whether the access to an address from a worker is local, can then be defined as:

$$\text{is\_local}(w, a) = \begin{cases} 1 & \text{if core\_node}(\text{cpu}(w)) = \text{node\_of}(a) \\ 0 & \text{otherwise} \end{cases}$$

In the following definition we assume that each input buffer is entirely placed on a single node. This implies that for each address of the buffer node_of yields the same value. With a function worker_of : $T_\infty \rightarrow W$, which defines by which worker a task was executed, R_{loc} can be approximated as follows:

$$R_{\text{loc}} \approx R^{\text{appr}}_{\text{loc}} = \frac{A_{\text{loc}}}{A_{\text{tot}}} \qquad \text{with} \qquad A_{\text{loc}} = \sum_{t \in T_\infty} \sum_{\substack{(a_s, a_e) \\ \in \\ \text{WS}_C(t)}} (a_e - a_s + 1) \cdot \text{is\_local}(\text{worker\_of}(t), a_s)$$

$$\text{and} \qquad A_{\text{tot}} = \sum_{t \in T_\infty} \sum_{\substack{(a_s, a_e) \\ \in \\ \text{WS}_C(t)}} (a_e - a_s + 1)$$

In this definition, A_{loc} is an approximation for the number of bytes accessed locally and A_{tot} is an approximation for the overall number of bytes accessed throughout execution of all tasks. The resulting approximation for R_{loc} is less precise than using hardware performance counters, since it entirely neglects the cache hierarchy. For example, if a worker executes a task whose input data is present in the local cache, the accesses to the input buffers would be counted by A_{tot}, although the transfer takes place between the core and the cache and not between the core and the memory controller. However, assuming a constant average last level cache miss rate r^{LLC}_m, the definition of the approximation of R_{loc} does not change:

$$R^{\text{appr}}_{\text{loc}} = \frac{r^{LLC}_m \cdot A_{\text{loc}}}{r^{LLC}_m \cdot A_{\text{tot}}} = \frac{A_{\text{loc}}}{A_{\text{tot}}}$$

Last, this metric does not take into account tasks for which the working set cannot be determined or for which the placement of input or output buffers cannot be determined. In particular, these are auxiliary tasks with memory accesses that do not target stream elements and tasks that write to buffers that have not been used before and whose placement thus cannot be determined by the run-time before the access. However, auxiliary tasks only represent a small fraction of all tasks and the occurrence of tasks writing for the first time to a buffer after a refill becomes less likely over time. The accuracy of the approximation is briefly discussed in Section 7.4.2.

2. *CPU_IO_REQUESTS_TO_MEMORY_IO:LOCAL_CPU_TO_LOCAL_MEM*
3. *CPU_IO_REQUESTS_TO_MEMORY_IO:LOCAL_CPU_TO_REMOTE_MEM*

Figure 7.7: *Visual representation of data and task placement*

Measuring performance

The performance is expressed as the speedup over the default OpenStream run-time with random work-stealing and without work-pushing and as the speedup over the parallel baseline of shared memory implementations. The former speedup, s_{rnd}, is defined as the wall clock execution time $t_{\text{wct,rnd}}$ of the default configuration divided by wall clock execution time t_{wct} of the configuration for which the speedup is calculated:

$$s_{\text{rnd}} = \frac{t_{\text{wct,rnd}}}{t_{\text{wct}}}$$

Similarly the latter speedup s_{shm} is defined as the wall clock execution time of the shared memory baseline $t_{\text{wct,shm}}$ divided by t_{wct}:

$$s_{\text{shm}} = \frac{t_{\text{wct,shm}}}{t_{\text{wct}}}$$

Graphical representation of data locality

The explanation of some of the results in the following section requires an analysis of the influence of the task graph on the work-pushing heuristics. For a compact visual representation combining the task graph, the placement of input data and the node on which a task executes, each task hereinafter may be represented by two colored and patterned semi-circles. The color and pattern of the right semi-circle indicate on which node the task's input data is placed, while the color and pattern of the left semi-circle show to which node the executing core belongs (cf. Figure 7.7). A question mark on the left circle indicates that the worker that will execute the task is still to be determined as can be the case for a task that has not been created, a task that is not ready for execution or a task whose dependences have all been satisfied, but which is not executing, yet. A question mark on the right side indicates that the placement of the input buffers is still unknown. This applies to tasks that have not been created and to input buffers that have not been allocated physically. In the examples, we use up to four different NUMA nodes n_a, n_b, n_c and n_d, identified by the following colors and patterns:

- Red / small triangles: node n_a
- Green / small rectangles: node n_b
- Blue / horizontal stripes: node n_c
- Yellow / crosshatch: node n_d

7.4.2 Results for work-pushing

We start the experimental evaluation with a comparison of the different heuristics for work-pushing with default random work-stealing. We first investigate the influence on data locality and then evaluate the impact on performance. Unless mentioned otherwise, each bar in the graphs below represents the mean value for a total of 50 executions. Error bars indicate standard deviation. Due to a huge memory footprint resulting from the default mechanism for broadcasts the *cholesky* benchmark is not evaluated in this chapter.

Locality of accesses to main memory

Figure 7.8 shows the locality of requests to main memory R_{loc} on the Opteron platform for the three work-pushing heuristics and the set of benchmarks using dynamic single assignment. The graph shows that all of the three heuristics improve the fraction of accesses to local memory significantly. When using the *input only* heuristic, approximately half of the requests to memory target local memory for most of the benchmarks. The highest value of 80.6% is achieved for *k-means*

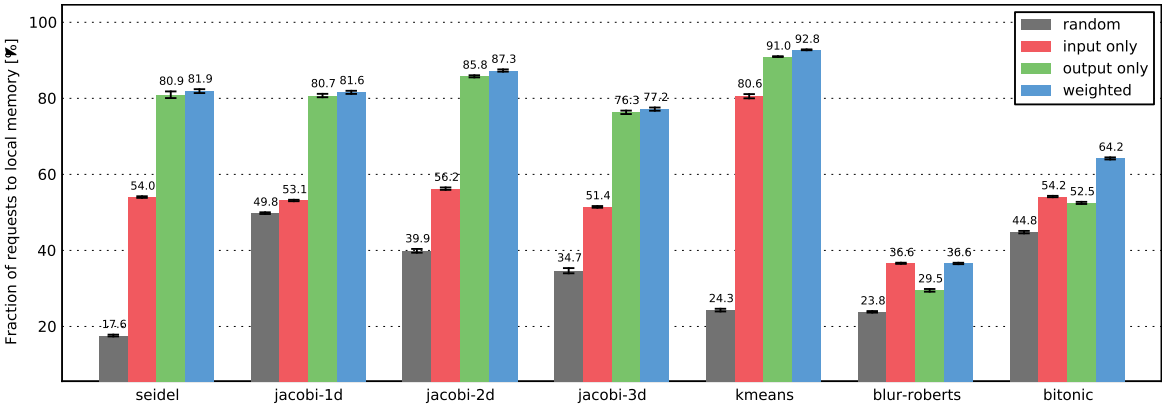
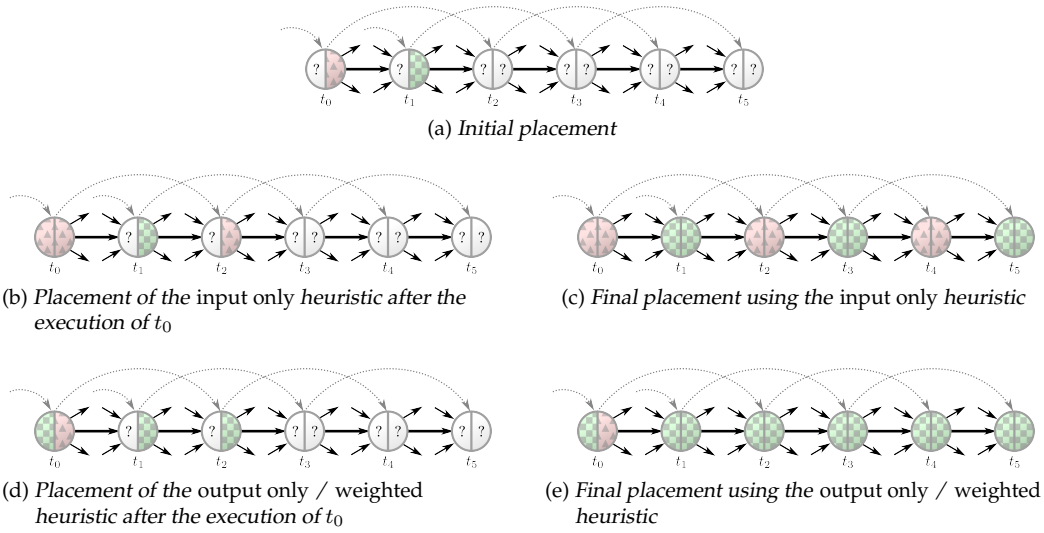


Figure 7.8: *Locality of requests to main memory on the Opteron system for the push heuristics*



(a) *Initial placement*

(b) *Placement of the* input only *heuristic after the execution of* t_0

(c) *Final placement using the* input only *heuristic*

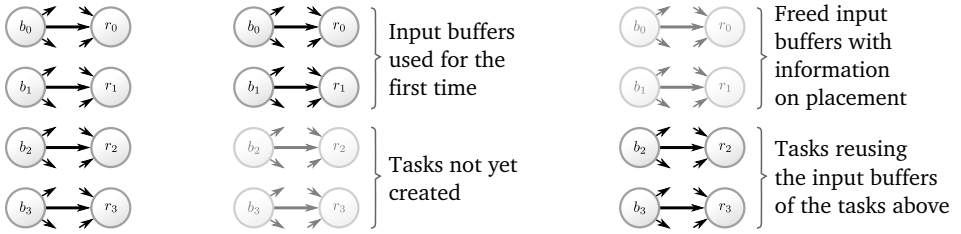
(d) *Placement of the* output only / weighted *heuristic after the execution of* t_0

(e) *Final placement using the* output only / weighted *heuristic*

Figure 7.9: *Influence of the push heuristic on* seidel *and* jacobi

and the lowest value of 36.6% is the result for *blur-roberts*. For *seidel*, *jacobi-1d*, *jacobi-2d*, *jacobi-3d* and *k-means* the *output only* heuristic and the *weighted* heuristic yield approximately the same locality and perform significantly better than the *input only* heuristic with values of about 80% and more than 90% for *k-means*. For the *bitonic* benchmark *input only* and *output only* provide the same data locality of more than 50%, while the *weighted* heuristic increases locality to more than 60%. The locality of *blur-roberts* is similar for *input only* and *weighted* (more than 30%) and a bit lower for the *output only* heuristic (less than 30%).

The reason for the improvement of locality by the *output only* and *weighted* heuristic is a synergistic effect of the structure of the task graph, the control program and the push heuristic. One of the main characteristics of the benchmarks for which the *output only* and *weighted* heuristic yield a higher value for data locality is the presence of highly unbalanced dependences and long paths with heavy dependences, such as in the task graph shown in Figure 7.9. Although this task graph represents only a simplified subset of the actual task graphs and shows only a subset of the light dependences, the behavior of the work-pushing heuristics at execution time is similar. Figure 7.9a shows a possible initial placement of the first two tasks on the path of heavy dependences indicates by the thick arrows. The input buffers of t_0 are placed on the node n_a,



(a) *Task graph* (b) *Beginning of the execution* (c) *After termination of the first tasks*

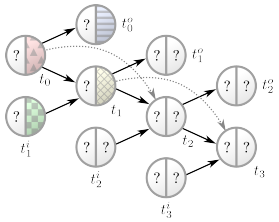
Figure 7.10: *Timing of the determination of data placement in* blur-roberts

while the input buffers of t_1 are placed on n_b. The *input only* heuristic pushes each task to the node that contains the tasks input buffers. As the input buffers of new tasks are allocated on the same node as the worker executing the creating tasks, the transfer of the creating task also conditions the placement of the task following the consumer of the creating task. For example, when t_0 becomes ready, it is pushed to a worker on n_a, which causes the input buffers of t_2 to be allocated on n_a, as shown in Figure 7.9b. The next task on the path, t_1 is pushed to a worker on n_b, which causes the input buffers of t_3 to be allocated on n_b. The resulting placement, shown in Figure 7.9c, is an alternation of nodes n_a and n_b. Due to work-pushing, read accesses are local, but write accesses are always remote. The *output only* and the *weighted* heuristic indirectly break these alternations, as a task is pushed to the worker containing a task's output buffers, which corresponds to a transfer to the node containing the input buffers of the following task on the path. Hence, at the very beginning of the execution, the input buffers of tasks are placed on different nodes due to initial placement, but after the creation of t_2 (Figure 7.9d) all tasks are executed on the same node containing both the tasks' input and output buffers as shown in Figure 7.9e.

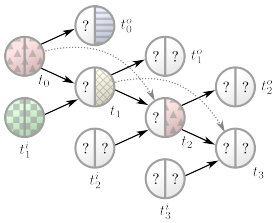
For *blur-roberts* the *output only* heuristic does not perform as well as the *input only* and *weighted* heuristic. This is due to the structure of the task graph, with dependences similar to *seidel* and *jacobi*, but with only two tasks on a dependence path. Figure 7.10a shows a simplified excerpt of the task graph for *blur-roberts*. For a significant part of the tasks, the *output only* fails to obtain information on data placement as the the metadata sections of the output buffers have not been updated due to the fact that they are used for the first time (Figure 7.10b). This information is only added to the metadata section at termination of the first tasks on the path. When these buffers are reused, their placement is known and the *output only* heuristic succeeds (Figure 7.10c). Thus, the *output only* heuristic does not systematically fail for all tasks and is able to improve the locality of data accesses compared to the default task activation strategy. However, the *input only* heuristic is guaranteed to be provided with accurate information just in time as the information on placement for the second tasks on the paths is available when they become ready for execution. Similarly, the *weighted* heuristic can react to the placement of input data. For tasks with missing information on the placement of output buffers, the heuristic simply ignores output dependences and acts exactly like the *input only* heuristic.

For *bitonic*, the improvement of data locality is less pronounced than for the other benchmarks. In addition, the *input only* and *output only* heuristic perform equally well. The *weighted* heuristic, however, yields slightly better results than the other heuristics. This is due to the influence of the task graph on the different heuristics illustrated in Figure 7.11, showing a subset of the task graph of *bitonic*. Assume that initially, the input buffers of t_0, t_0^o, t_1 and t_1^i are placed as shown in Figure 7.11a. The *input only* heuristic causes t_0 to be executed on the node containing its input data, leading to the allocation of the input buffers of t_2 on the same node as t_0 (Figure 7.11b). When t_1 becomes ready, it is scheduled on the node containing its input buffers, causing the input buffers of t_3 to be allocated on the same node as shown in Figure 7.11c. This leads to an alternating placement with limited data locality as seen earlier for the *input only* heuristic applied to *seidel* and *jacobi*.

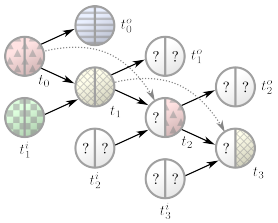
However, in contrast to these benchmarks, the *output only* heuristic does not trigger the same synergistic effect leading to both local read and write accesses. Consider Figure 7.11d, which shows



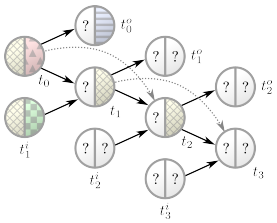
(a) *Initial placement*



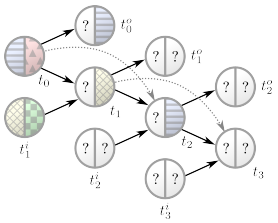
(b) input only *heuristic, first step*



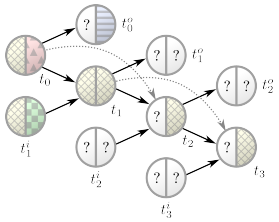
(c) input only *heuristic, second step*



(d) output only *heuristic, first step*

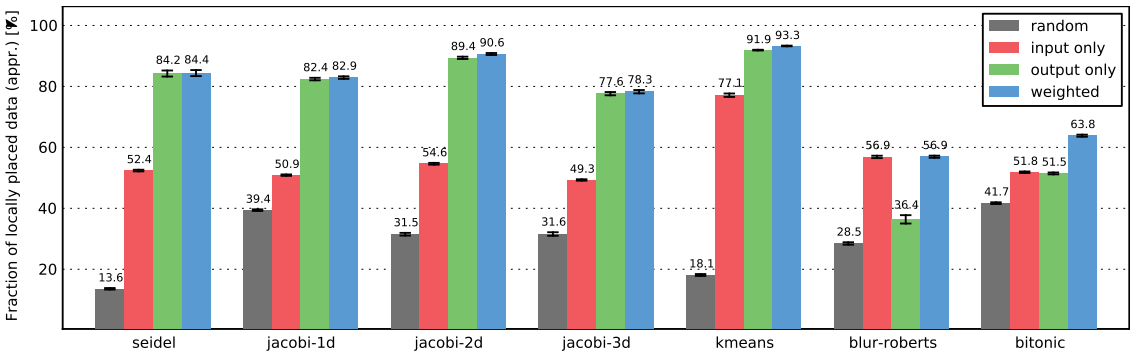


(e) output only *heuristic, alternative first step*

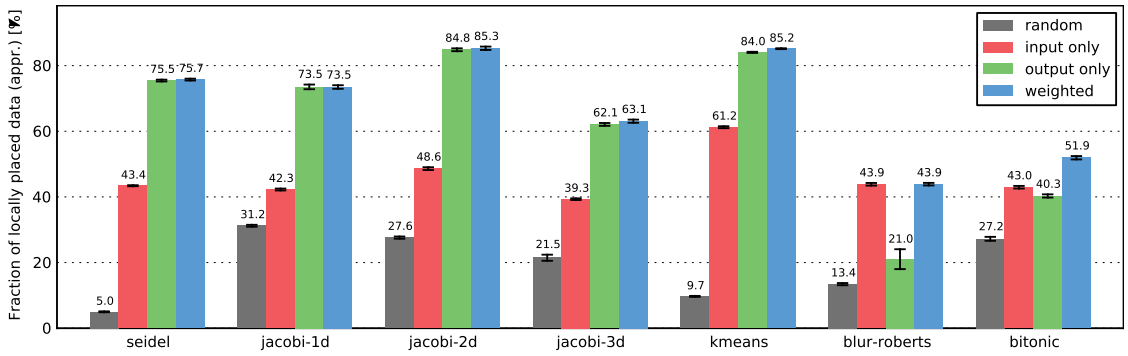


(f) *Possible outcome for the second step using the* weighted *heuristic*

Figure 7.11: *Effect of the push heuristics on* bitonic



(a) *Opteron system*



(b) *SGI system*

Figure 7.12: *Approximation R_{loc}^{appr} of the locality for the push heuristics*

a possible outcome for the task ownership of t_0. Although there is a beginning of a path of tasks with input buffers on the same node including t_1 and t_2, it is not guaranteed that these tasks will be scheduled to nodes that ensure local accesses. For example, where t_1 will be scheduled also depends on the placement of t_0^o, which is created by another task. Hence, the input buffers of t_3 might be allocated on a different node than t_2 and the path of tasks with input data on the same node finishes, leading to a lower locality of memory accesses. In addition, t_0 could have been scheduled differently, as shown in Figure 7.11e, where a worker on the node containing the input buffers of t_0^o is the owner of t_0. In this case, the input buffers of t_2 are allocated on a different node than those of t_1, leading to an alternating pattern for the placement of input buffers with lower data locality.

The *weighted* heuristic initially has the choice between a worker of the node containing the input buffers of t_0, t_1 or t_0^o. If a worker on the same node as t_0^o or t_0 is chosen, the input buffers of t_1 and t_2 will be allocated on different nodes. However, if the same node as the node containing the input buffers of t_1 is chosen, t_1 is guaranteed to execute on the node containing its input buffers (the cost for this node will be minimal among all nodes since it contains the entire input buffers as well as half of the output buffers representing three quarters of the working set of t_1). This causes the input buffers of t_3 to be allocated on the same node as t_1 and t_2 as shown in Figure 7.11f. Afterwards, this pattern repeats for t_2 and t_3 and the series of dependent tasks whose input buffers are placed on the same node is not interrupted as easily as for the *output only* heuristic. The locality of data accesses for the *weighted* heuristic is thus higher than the locality for the other heuristics.

Figure 7.12 shows the approximation $R_{\text{loc}}^{\text{appr}}$ of the locality for the push heuristics on the Opteron and the SGI system. The locality for the Opteron system measured with hardware performance counters of Figure 7.8 is well reflected by the approximation in Figure 7.12a. Figure 7.13 shows the median of the relative error in per cent of the approximation with error bars indicating the

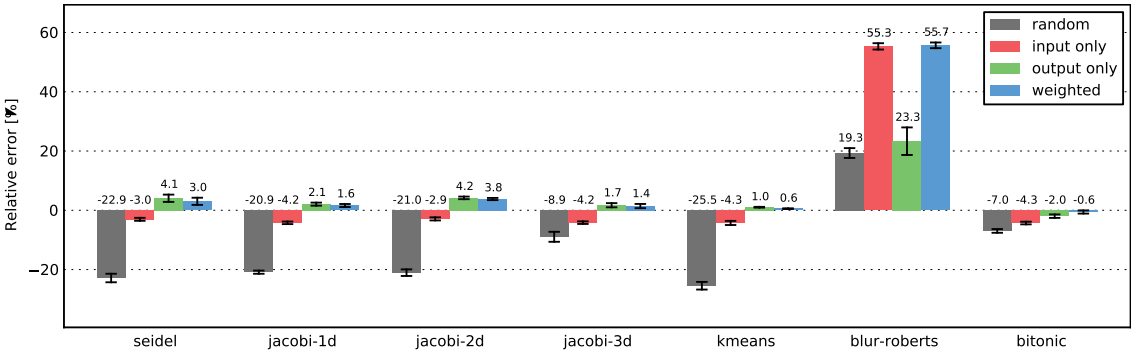


Figure 7.13: *Relative error of R_{loc}^{appr} over the locality measured with hardware performance counters*

standard deviation. Besides an underestimation of the locality for random work-stealing without pushing and an overestimation of the locality of *blur-roberts*, the relative error is equal to or less than five per cent and thus relatively low. Hence, it is reasonable to assume that the approximation for the SGI system shown in Figure 7.12b reflects the actual locality for the push heuristics for *seidel*, *jacobi-1d*, *jacobi-2d*, *jacobi-3d*, *k-means* and *bitonic*. The results shown in this graph are very similar to the results for the Opteron system presented above and the conclusions are the same. Thanks to the use of architecture-independent concepts and the adaptation of the work-pushing heuristics to the respective target platform through the definition of an appropriate function modeling the cost of data transfers, the work-pushing heuristics perform similarly for each benchmark on both machines. The push heuristic that yields the best locality for memory accesses depends on the benchmark and not on the topology of the machine, which emphasizes the portability of both work-pushing as well as the applications.

Impact on performance

Figure 7.14 shows the speedup of the executions with work-pushing enabled over the default randomized work-stealing without work-pushing. The improvements on the locality of accesses to main memory result in a significant increase of the performance for most of the memory-intensive benchmarks with speedups of up to $2.36\times$. The only exception to this rule are the *input only* heuristic for *jacobi-1d* on both systems ($0.87\times$ and $0.75\times$), *jacobi-2d* on the SGI system ($0.92\times$) and *jacobi-2d* on the SGI system ($0.96\times$) as well as the *output only* heuristic for *blur-roberts* on the Opteron system ($0.97\times$). For the *jacobi* benchmarks, this can be explained with the effect shown earlier in Figure 7.9c. Although the locality of read accesses to main memory is significantly higher, resulting in an overall locality that is higher than *rnd*, almost all write accesses target remote memory and performance is lower compared to *rnd*. As far as *blur-roberts* is concerned, the *output only* heuristic fails too often due to missing information on data placement as explained above. In the remaining cases, the heuristic causes a slight load imbalance across memory controllers and thus decreases performance. On the SGI system, this is not the case and the improvement of data locality reduces the execution time in comparison to *rnd*.

The performance of *k-means* is not affected on the Opteron platform and only increases slightly on the SGI system with a maximum speedup of $1.08\times$. This is due to the fact that this benchmark has a very low cache miss rate in contrast to the other benchmarks, which are clearly memory-bound. Hence, the locality of the few memory accesses missing the last level cache only has little influence on performance. Another interesting observation is that the speedup over random work-stealing without work-pushing is generally higher for the SGI system. This is a result of an increased ratio of the average latency of accesses to remote memory over the latency of accesses to local memory compared to the Opteron system. In addition, the higher number of nodes leads to a lower initial locality of random work-stealing on this platform.

Considering the geometric mean of the mean speedups, shown at the right side of the figures, it can be concluded that in most cases the *weighted* heuristic performs best ($1.26\times$ and $1.50\times$),

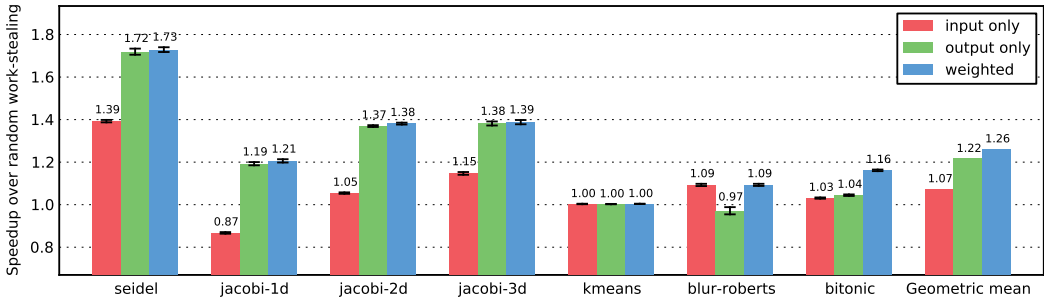
followed by the *output only* heuristic ($1.22\times$ and $1.42\times$) and the *input only* heuristic ($1.07\times$ and $1.04\times$). Hence, using the *weighted* heuristic as the default heuristic for work-pushing might be most beneficial for the average performance of applications whose behavior has not been studied in detail. However, although the *input only* heuristic performs less well than the other heuristics, it can be beneficial combined with NUMA-aware allocation, as shown in the next chapter.

The speedups of random work-stealing without work-pushing and the different push heuristics over the shared memory implementations with interleaved allocation are summarized in Figure 7.15. For the Opteron system, the experiments yield the following results. The *output only* and *weighted* heuristic allow the dynamic single assignment version of *seidel* to compensate the low performance of *rnd* and exceed the performance of the shared memory implementation with a speedup of $1.2\times$. The execution time of work-pushing using the *input only* heuristic for this benchmark is approximately the same as for the shared memory implementation. For *jacobi-1d*, the initial performance of *rnd* is much lower than the shared memory baseline ($0.63\times$). Even the improvements of the *output only* and *weighted* heuristic with speedups of $0.75\times$ and $0.76\times$ cannot increase performance above this level. The *jacobi-2d* benchmark yields similar performance for *rnd* and *input only*. Both of these configurations have a higher execution time than the shared memory implementation with speedups of $0.76\times$ and $0.8\times$. However, the *output only* and *weighted* heuristic increase performance of *jacobi-2d* to the same level as the shared memory implementation ($1.04\times$ and $1.05\times$). The initial performance of *rnd* for *jacobi-3d* already slightly exceeds the performance of the shared memory implementation ($1.02\times$) and each push heuristic widens this gap ($1.17\times$, $1.41\times$ and $1.42\times$). As mentioned above, the *k-means* application is insensitive to data placement on the Opteron system due to its low cache miss rate. The initial performance of this benchmark is worse than shared memory due to the copying of read-only data from one task to another. Hence, the performance of all configurations remains lower than for the shared memory implementation. For *blur-roberts* all dynamic single assignment configurations, with or without work-pushing, outperform the shared memory implementation with a speedup of up to $1.23\times$ for the *input only* and the *weighted* heuristic. The performance increase resulting from work-pushing for the *bitonic* benchmark are not sufficient to meet the performance of the shared memory implementation, the maximum speedup is only $0.60\times$.

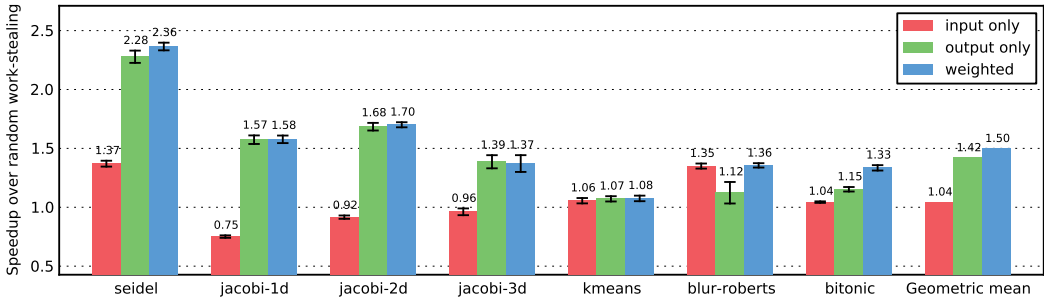
For the SGI platform, the results for the comparison with the shared memory implementations are similar. A notable exception is *jacobi-1d*, which performs significantly better than the shared memory implementation even for default random work-stealing without work-pushing ($1.58\times$). This performance is further improved when using either the *output only* ($2.49\times$) and the *weighted* heuristic ($2.50\times$). Also, the *output only* and the *weighted* heuristic allow the *jacobi-2d* benchmark to exceed the performance of the shared memory implementation more distinctly ($1.17\times$ and $1.18\times$) than on the Opteron platform. The geometric mean of the mean speedups shows that using the *output only* or the *weighted* heuristic, the dynamic single assignment versions outperform the shared memory implementations on average. For the Opteron platform, the geometric mean for these heuristics is still a bit lower than one. The reason for the higher values on the SGI platform is that remote accesses on this platform have a higher latency, such that accesses to data that is distributed over all nodes as in the shared memory benchmarks are slower. Hence, the higher the ratio of the latency of remote memory accesses over the latency of local memory accesses, the more data locality becomes important for performance and the more techniques such as work-pushing are beneficial.

7.4.3 Results for topology-aware work-stealing

The following presentation of the results for topology-aware work-stealing has the same order as the presentations of the results for the push heuristics: we first analyze the impact on the locality of accesses to main memory and demonstrate the resulting performance over random work-stealing without work-pushing. We then show how work-pushing with topology-aware work-stealing performs compared to the shared memory implementations. As topology-aware work-stealing relies on the matching of task and data ownership restored by work-pushing, we do not evaluate topology-aware work-stealing with the default task activation scheme.

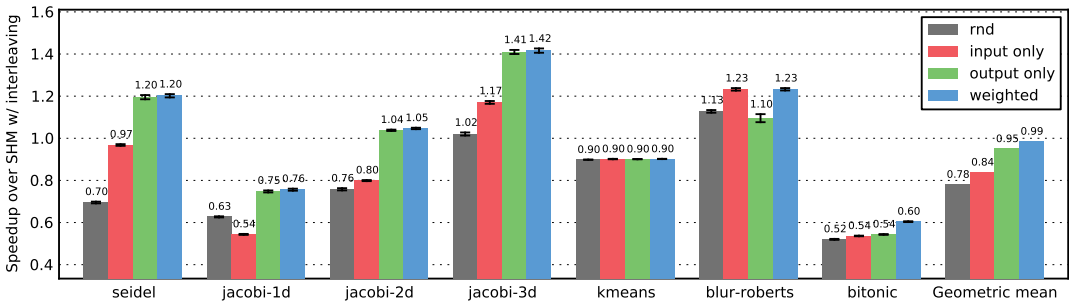


(a) *Opteron system*

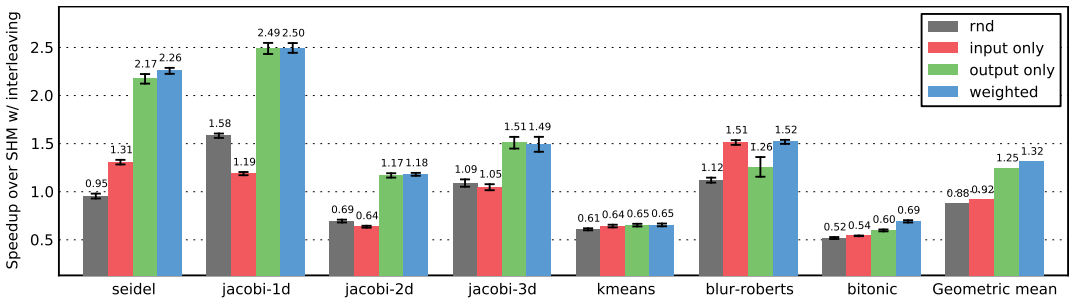


(b) *SGI system*

Figure 7.14: *Speedup of the push heuristics over default random work-stealing without work-pushing*



(a) *Opteron system*



(b) *SGI system*

Figure 7.15: *Speedup of the push heuristics over the shared memory implementations*

Locality of accesses to main memory

Figure 7.16 shows the locality of requests to main memory R_{loc} for the different work-pushing heuristics combined with topology-aware work-stealing on the Opteron platform. Compared to the configurations that use work-pushing only of Figure 7.8, the locality of memory accesses resulting from the combination of work-pushing with topology-aware work-stealing is significantly higher and reaches a value of more than 90% for all applications, except *blur-roberts* and *bitonic*. To highlight these differences, Figure 7.17 shows the relative improvement of the combination of the approaches over work-pushing only. Each bar shows the median of the relative improvement of the locality over the median for the data locality of work-pushing only expressed in per cent. For example, the first value of 22.6% for topology-aware work-stealing and the *input only* heuristic at execution of the *seidel* benchmark indicates that the fraction of accesses to local memory is 22.6% higher compared to work-pushing with the *input only* heuristic with random work-stealing. The figure shows that data locality is improved for all work-pushing heuristics and all benchmarks. The geometric mean ranges from 8.8% for the *weighted* heuristic to 12.2% for the *input only* heuristic. The lower improvements for *output only* and *weighted* are due to the fact that data locality for these heuristics without topology-aware work-stealing is already very high and thus more difficult to improve than for the *input only* heuristic.

Figure 7.18 shows the approximation for the fraction of accesses to local memory for the SGI system. As was the case for the Opteron system, topology-aware work-stealing improves the locality of all benchmarks and all heuristics. The relative improvement over work-pushing without topology-aware work-stealing presented in Figure 7.12b is shown in Figure 7.19. The high variation of *blur-roberts* and the *output only* heuristic results from different timings of buffer reuse and thus different availability of information on buffer placement upon task activation mentioned in Section 7.4.2, such that the results for this benchmark should not be taken into account. We have therefore excluded *blur-roberts* from the calculation of the geometric mean.

In conclusion, topology-aware work-stealing improves the locality of memory accesses on both systems for all benchmarks and almost reaches the maximum locality with values close to or higher than 90% for most of the benchmarks.

Impact on performance

To quantify the impact of the increased data locality of topology-aware work-stealing on performance, Figure 7.20 shows the reduction of execution time of the different push heuristics with respect to work-pushing without topology-aware work-stealing. For *seidel*, *jacobi-2d*, *jacobi-3d* and *bitonic* the execution time can be reduced on both systems and for all work-pushing heuristics.The impact on performance for the *k-means* and *blur-roberts* benchmark is slightly negative on the Opteron platform, while the results for these applications are mixed on the SGI system. The same applies to the JACOBI-1D benchmark on the SGI system only. For *k-means* on the SGI system the median value corresponds to a slight reduction of the execution time, but the variation is high as indicated by the large standard deviation.

Figure 7.21 shows the speedup of the dynamic single assignment versions with work-pushing and topology-aware work-stealing over the shared memory implementations. The characteristics are similar to Figure 7.15. Configurations whose performance already exceeded the performance of the shared memory implementation without topology-aware work-stealing keep this advantage and configuration with initial lower performance still require more time to execute than the baseline.

7.5 Summary and conclusion

In this chapter, we showed that the default task activation scheme, which adds a newly activated task to the single entry software cache of the activating worker, can lead to a mismatch between task and data ownership and thus to poor data locality during execution of the task. In particular, the locality can depend on the timing of the execution of the task's producers or on the structure of the dynamic task graph as well as the order of task creation by the parallel control program. To

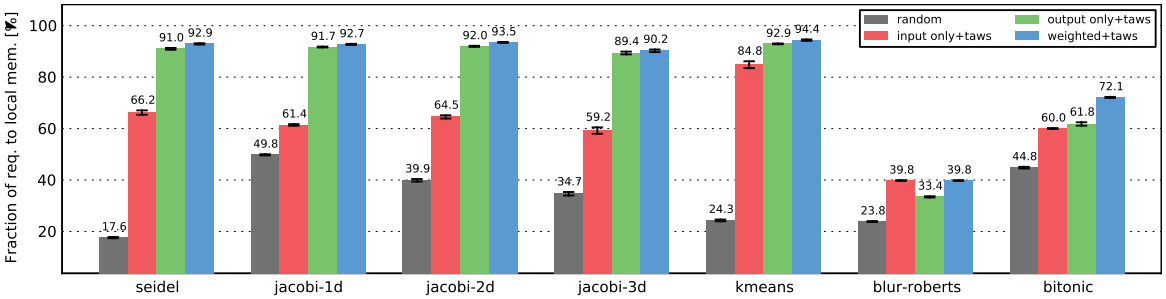


Figure 7.16: *Locality of requests to main memory on the Opteron system for the push heuristics combined with topology-aware work-stealing*

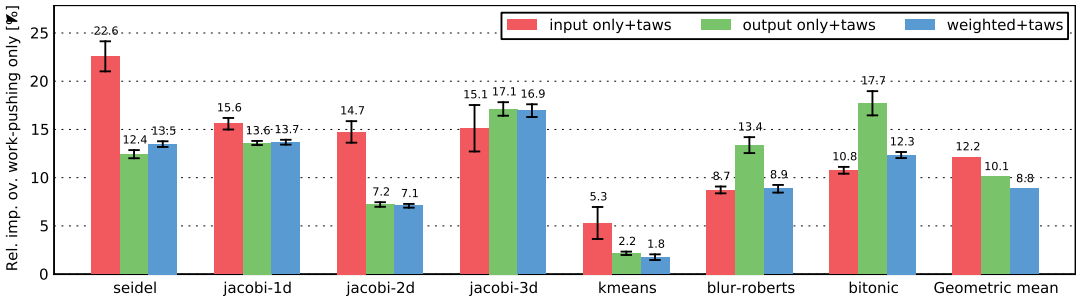


Figure 7.17: *Relative improvement of the locality of requests to main memory on the Opteron system for the push heuristics combined with topology-aware work-stealing*

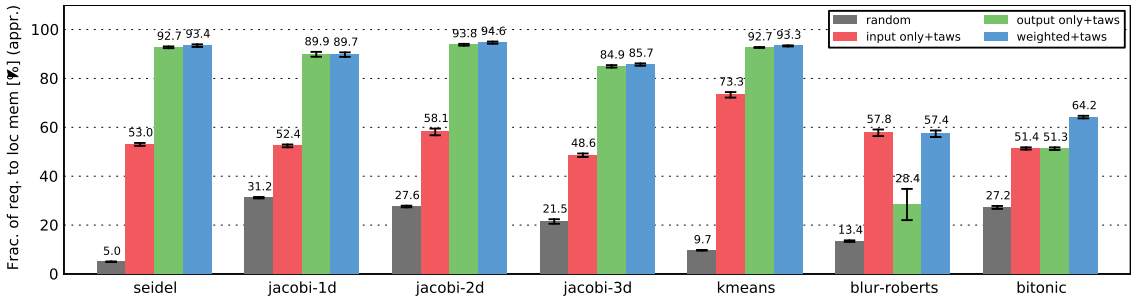


Figure 7.18: *Approximation R_{loc}^{appr} of the locality for the push heuristics combined with topology-aware work-stealing for the SGI system*

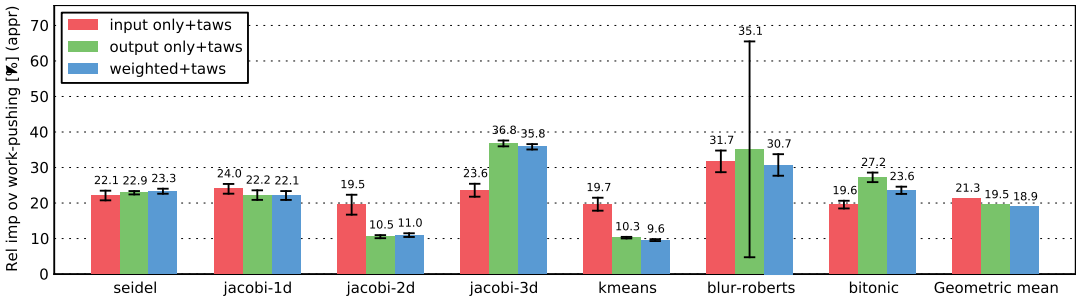
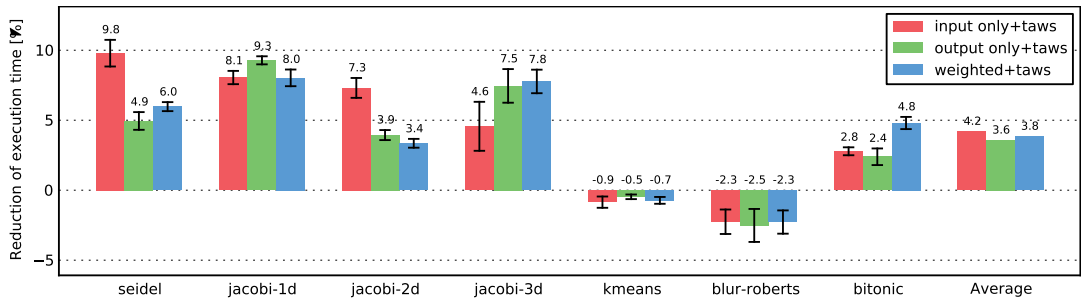
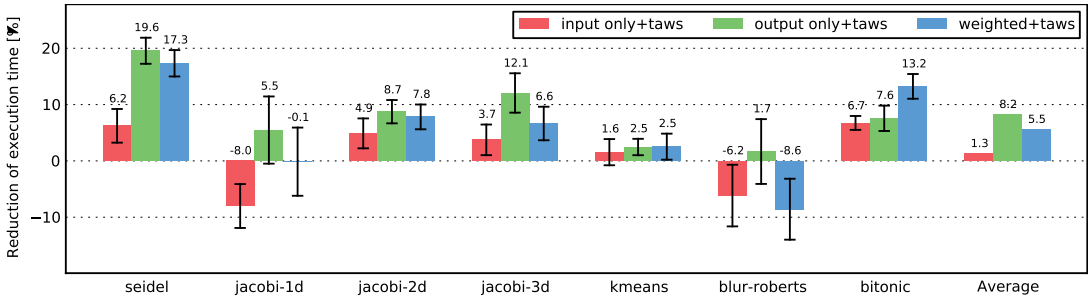


Figure 7.19: *Relative improvement of the approximation R_{loc}^{appr} of the locality of requests to main memory on the SGI system for the push heuristics combined with topology-aware work-stealing*

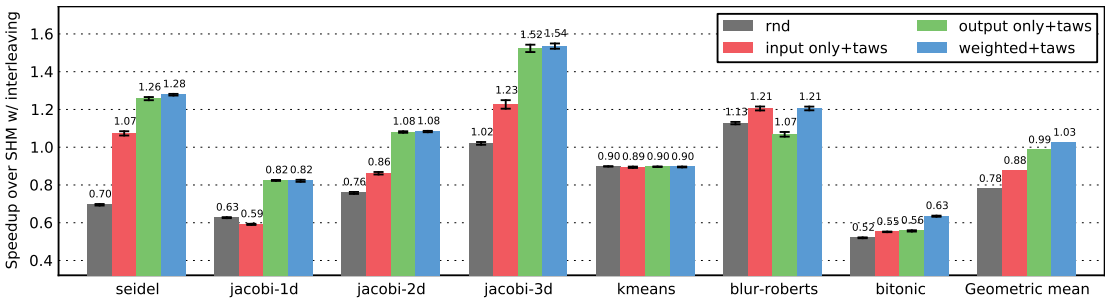


(a) *Opteron system*

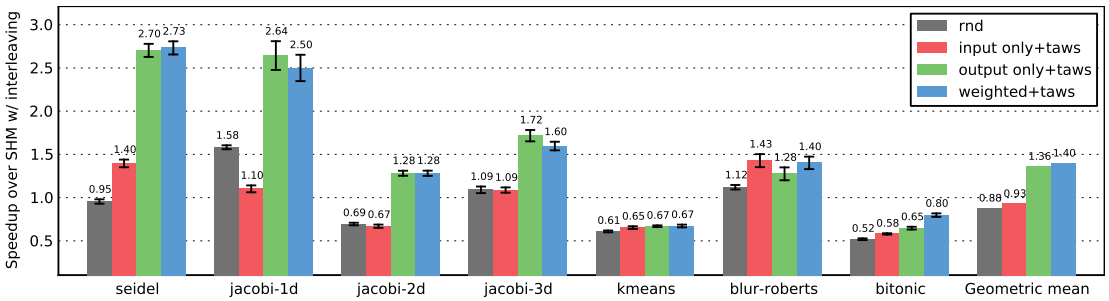


(b) *SGI system*

Figure 7.20: *Improvement of the execution time of the push heuristics combined with topology-aware work-stealing compared to work-pushing only*



(a) *Opteron system*



(b) *SGI system*

Figure 7.21: *Speedup of the push heuristics combined with topology-aware work-stealing over the shared memory implementations*

reduce the mismatch between the ownership of data and tasks and thus to increase the locality of accesses to main memory, we proposed work-pushing, which transfers a task to a worker of an appropriate node upon activation according to a heuristic based on dynamic information on the placement of the task's input and output buffers. We presented three heuristics for the selection of this node, which take into account only input buffers, only output buffers or both kinds of buffers. We also showed that the default random work-stealing mechanism can cause accesses to remote memory even with matching task and data ownership. To mitigate this problem, we introduced topology-aware work-stealing, which prioritizes steals from nearby workers.

The experimental evaluation showed that work-pushing and topology-aware work-stealing can be integrated into the run-time system and improve the data locality significantly on an AMD Opteron system with eight NUMA nodes as well as on an SGI platform with 24 NUMA nodes. On both platforms, the fraction of accesses to local memory using these techniques comes close to the maximal locality with values close to or above 90% for most of the applications. We showed that data placement can be essential for the performance as for memory-intensive applications the improvement of the locality translates into a significant reduction of the wall clock execution time. The maximal speedup over the default task activation scheme with random work-stealing is as high as $1.73\times$ on the Opteron platform and reached $2.36\times$ on the SGI system. In some cases, the improvements on execution time allow the dynamic single assignment implementations to outperform the parallel shared memory baseline (*seidel*, *jacobi-2d* and *jacobi-3d* on the Opteron system and *seidel* as well as *jacobi-2d* on the SGI system) with speedups of up to $1.42\times$ on the Opteron system and $2.50\times$ on the SGI system. For another set of benchmarks, namely *jacobi-1d* and *jacobi-3d* on the SGI system as well as *blur-roberts* on both systems, the performance of the dynamic single assignment versions already exceeds the performance of the shared memory implementations when using the default task activation scheme and random work-stealing and can be improved further using work-pushing and topology-aware work-stealing. Finally, the performance of *k-means* and *bitonic* can be improved in some cases, but remains lower than the performance of the shared memory baseline. We showed that work-pushing and topology-aware work-stealing are portable across the test systems thanks to the use of platform-independent code in conjunction with light-weight descriptions of the hardware topology. Which work-pushing heuristic performs best and whether topology-aware work-stealing should be applied depends on the application. However, in most cases, the *weighted* heuristic yields the best results and topology-aware work-stealing improves performance.

Work-pushing and topology-aware work-stealing are techniques that *react* to the placement of data and only influence data placement *indirectly*. In the next chapter, we focus on run-time mechanisms for memory allocation and introduce techniques that improve the locality of accesses to main memory through *active* data placement.

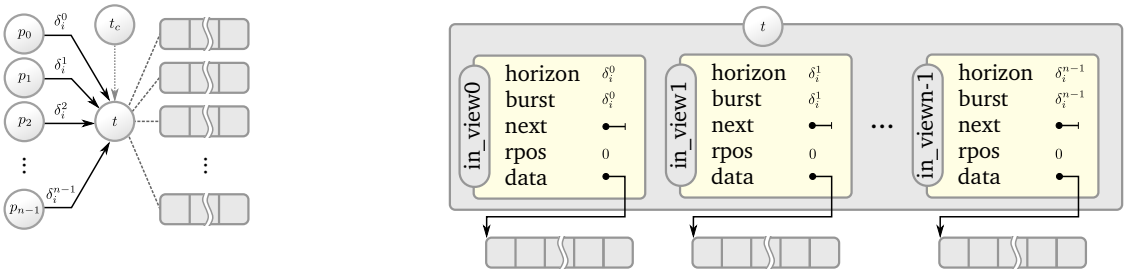
8 Deferred allocation

The work-pushing heuristics and topology-aware work-stealing presented in the previous chapter react to the placement of data and assign tasks to workers whose associated nodes have the lowest latency for accesses to the data. The placement of input buffers is only addressed indirectly through allocations within the memory pools of the workers that execute the creating tasks. The restriction of the OpenStream execution model, which defines that a task cannot create a direct consumer, causes the input buffers of tasks to be placed in advance before the nodes of the workers executing the tasks producers are known. This can lead to data placement that is unfavorable for the locality of accesses using either work-pushing heuristic or topology-aware work-stealing.

In this chapter, we analyze the impact of the default mechanism for the allocation of input buffers on data locality and load balancing across memory controllers. We then present a memory allocation technique which we refer to as *deferred allocation* that aims at mitigating the issues related to early allocation. We show that deferred allocation decouples task creation from data placement and improves both data locality and data distribution. We also illustrate an important side effect of deferred allocation that allows the run-time to decrease an application's memory footprint. The concepts of deferred allocation are evaluated on the same set of benchmarks as in the previous chapter.

8.1 Influence of the allocation mechanism on data locality

In the default scheme for memory allocation of the OpenStream execution model, all input buffers of a task are allocated when the task is created. To distinguish this allocation mechanism from the concepts introduced below, we use the term *immediate allocation* in the remainder of this chapter. Figure 8.1a shows the graph with a task t, which depends on n producers p_0 to p_{n-1} and which is created by t_c. During the execution of t_c, the creation of t is triggered and the input buffers of t are allocated and associated to t as indicated by the buffers connected with dashed lines. A detailed view of the run-time structures involved in this process is given in Figure 8.1b. As a result of the separation of input data from input views, introduced in Section 4.3, only the data structures representing input views of t are embedded in its data-flow frame with `data` fields pointing to the respective input buffers. The placement of these buffers is determined before t becomes ready and depends on the node associated to the worker that executes t_c. The main drawback of this placement strategy is that it does not take into account on which nodes p_0 to p_{n-1} execute. If these tasks are executed by workers on different nodes than the worker that executed t_c, all of the write accesses target remote memory, which leads to a high average latency of memory accesses. In this



(a) *Task graph with n producers providing the input data of a task t*

(b) *Data-flow frame, input views and input buffers after task creation*

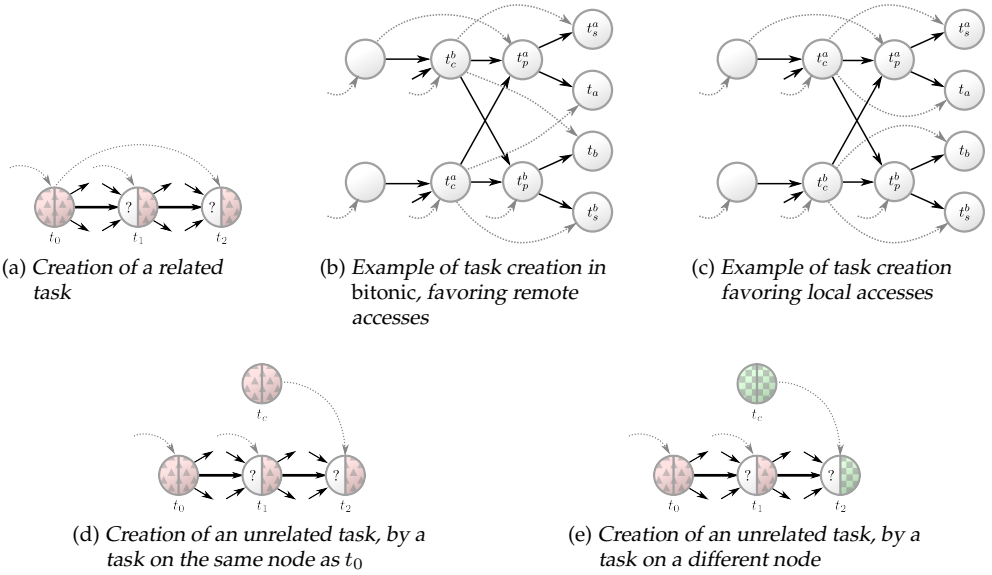
Figure 8.1: *Immediate allocation of input buffers*

section, we examine under which circumstances the producers of a task are executed on workers of different nodes than the creating task. This can be due to the structure of the task graph and the control program or due to work-stealing events. We first analyze the influence of the control program on data locality using immediate allocation, followed by an analysis of the influence of work-stealing. Finally, we show that the creation of initial tasks is an important part of the control program and greatly determines initial data placement when using immediate allocation.

8.1.1 Influence of the control program

A common strategy for the creation of tasks in a parallelized control program is that tasks create their grandchildren in the task graph, as shown in Figure 8.2a. If this pattern of task creation follows a path of heavy dependences and if the input buffers of the task between the creating task and the created task is allocated on the same node as the creator, this results in a chain of tasks with input buffers on the same node and enables execution with a high fraction of accesses to local memory. However, for unbalanced dependences the relationship between the control program and data locality can be more complex and it can be less clear in which order tasks should be created to favor a high fraction of accesses to local memory. Consider the excerpt of the task graph of the *bitonic* benchmark shown in Figure 8.2b, in which two tasks t_a and t_b are created by their respective grandparents t_c^a and t_c^b. The predecessors of t_c^a and t_c^b have common ancestors in the task graph (not shown in the figure), but the path of dependences from the nearest common ancestor to these tasks is long and includes numerous tasks with balanced dependences. It is thus likely that the input buffers of t_a and t_b are placed on different nodes, which in turn results in a placement of the input buffers of t_a and t_b and their respective siblings t_s^a and t_s^b on different nodes. Hence, independently from the choice of the workers for execution, their parent tasks t_p^a and t_p^b each write at least half of their input data to buffers on a remote node. A control program, in which t_a and its sibling t_s^a as well as t_b and its sibling t_s^b are created by the same node, as shown in Figure 8.2c, might be more convenient as t_p^a and t_p^b can be scheduled to workers for which all of the write accesses target local memory.

However, adapting the control program to avoid remote write accesses can be a tedious and complicated task. Some choices for the order of task creation might seem reasonable when analyzing a small excerpt of a task graph, but can have a negative impact on the tasks in other parts of the graph. Hence, the development of a control program resulting in a low number of remote write accesses often requires a global understanding of large parts of the task graph. However, ideally, the control program does not have any influence on the locality of memory accesses and the programmer can concentrate on providing sufficient parallelism without worrying about locality. For example, there should not be any difference in the results for data locality between the creation of t_2 by t_0 in Figure 8.2a and the creation of t_2 by t_c as in Figure 8.2d. This last pattern of task creation represents an extreme case for immediate allocation, since t_0 does not have any common



(a) *Creation of a related task*

(b) *Example of task creation in* bitonic, *favoring remote accesses*

(c) *Example of task creation favoring local accesses*

(d) *Creation of an unrelated task, by a task on the same node as* t_0

(e) *Creation of an unrelated task, by a task on a different node*

Figure 8.2: *Influence of the control program on locality using immediate allocation*

(a) *Initial placement*

(b) *Steal of* t_i *by a remote worker*

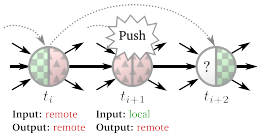
Figure 8.3: *Influence of work-stealing in conjunction with immediate allocation on the locality of write accesses*

ancestor with t_c. Hence, the structure of the task graph does not favor any relationship between the worker executing t_c and the worker executing t_1. The probability that the input buffers of t_2 are placed on the same node as t_1 is low and it is more likely that this pattern for task creation results in a placement similar to Figure 8.2e. Work-pushing can improve the locality in these cases, but it must choose between a high fraction of local accesses for write accesses (as would be the case when using the *output only* or *weighted* heuristic) and a high fraction of local read accesses of t_1 (such as for the *input only* heuristic).

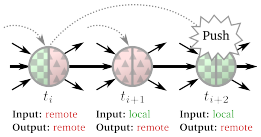
8.1.2 Influence of work-stealing

The early placement due to immediate allocation can also cause remote accesses upon work-stealing by a remote worker. Consider the initial placement in Figure 8.3a, where the input buffers of a task t_i and its successor t_{i+1} in a chain of heavy dependences are both placed on the same node. If t_i is stolen by a remote worker, as shown in Figure 8.3b, not only read accesses, but also write accesses target remote memory. In addition, the input buffers of t_{i+2} are allocated on the node of the worker that executes t_i, such that remote accesses become inevitable during execution of t_{i+1}. Thus, when using immediate allocation, it is possible that a steal does not only have a negative impact on the locality of accesses during execution of the stolen task, but also on tasks that depend on it.

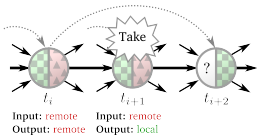
Figure 8.4 shows how the different work-pushing heuristics react to the data placement after the steal. The *input only* heuristic transfers t_{i+1} to the node that contains the input buffers of this task, which results in local accesses to input data, but which causes remote write accesses due to



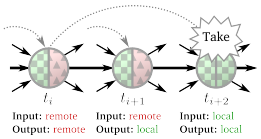
(a) *Activation of t_{i+1} Work-pushing using the* input only *heuristic*



(b) *Activation of t_{i+2} Work-pushing using the* input only *heuristic*



(c) *Activation of t_{i+1} Work-pushing using the* output only / weighted *heuristic*



(d) *Activation of t_{i+2} Work-pushing using the* output only / weighted *heuristic*

Figure 8.4: *Work-pushing after a steal using immediate allocation*

the placement of the input buffers of t_{i+2} (Figure 8.4a). Similarly, t_{i+2} must be transferred upon activation, resulting in local accesses to input data but also in remote write accesses to output buffers as shown in Figure 8.4b. The locality of accesses is higher when using the *output only* or *weighted* heuristic as illustrated by Figure 8.4c and 8.4d. An important point is that these heuristics keep the tasks t_{i+1} and t_{i+2} on the worker that initially stole t_i if t_i is the producer of t_{i+1} that finishes last and if t_{i+1} is the last producer of t_{i+2}. This ensures that the worker can obtain work simply by removing a task from the software cache (indicated by the label *take* in the figures) and thus does not have to steal another task right after the execution of t_i. However, if these tasks do not finish last among all producers, t_{i+1} and t_{i+2} are each transferred to a worker on the same node and the amount of locally accessed data remains the same.

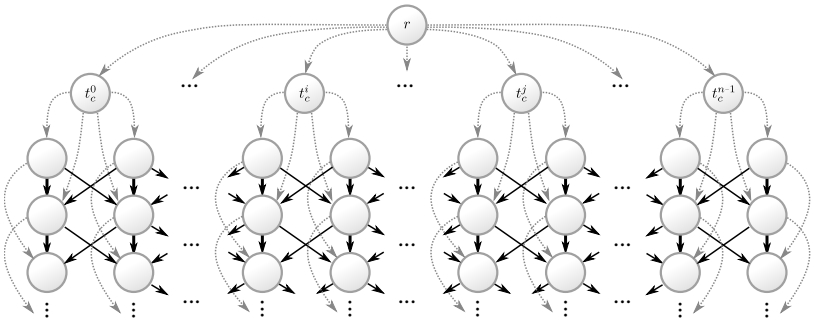
8.1.3 Influence of the creation of initial tasks

Although the creation of initial tasks is part of the control program, whose influence has been discussed above, this phase deserves particular attention as it largely influences data distribution at the beginning of the execution. In order to provide sufficient parallelism at the beginning of the execution, the creation of initial tasks is parallelized in the benchmarks presented in Section 6.1. Instead of creating all initial tasks during execution of the root task, the root task creates a set of auxiliary tasks whose only purpose is to create the initial tasks of the application. At the beginning of the execution these tasks are stolen by the remaining workers and thus execute in parallel, resulting in parallel creation of the initial tasks.

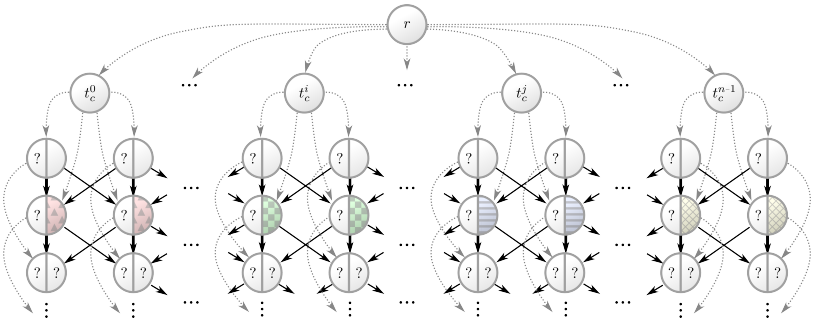
Figure 8.5a illustrates this principle on the *jacobi-1d* benchmark. The root task r creates n auxiliary tasks t_c^0 to t_c^{n-1}, which in turn create the first two tasks on each chain of heavy dependences associated to each block of data [1]. A large fraction of the tasks t_c^0 to t_c^{n-1} is stolen by remote workers, which causes the input buffers of the first tasks on the chains to be spread across the machine's memory controllers. Hence, a side effect of the parallelization of the creation of initial tasks is a distribution of data that avoids the placement of an excessively large fraction of input data is placed on the node of the worker executing the root task. This is shown in Figure 8.5b, where the input buffers of the initial tasks are spread in a uniform manner across the nodes. Note that the right semi-circle of the first tasks of the chains of heavy dependences below the auxiliary tasks for task creation do not indicate any data placement, since these tasks do not have input dependences and thus do not have input buffers.

In practice, some nodes might be privileged for steals of the t_c^0 to t_c^{n-1} from the worker executing the root task due to faster access time to the memory of the node on which the worker executes.

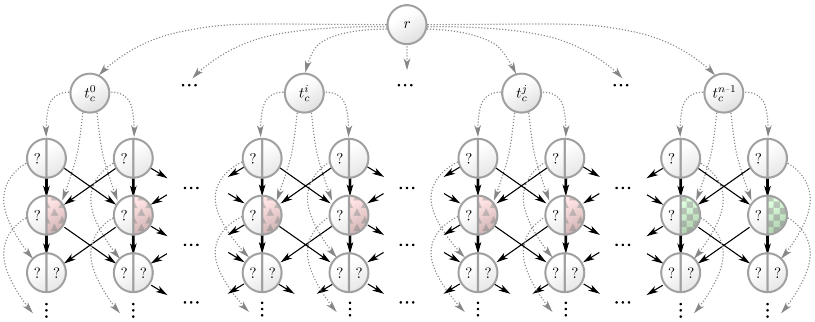
1. In the actual implementation the number of tasks created is four, but to keep the task graph simpler, we have shown the creation of two tasks per auxiliary task.



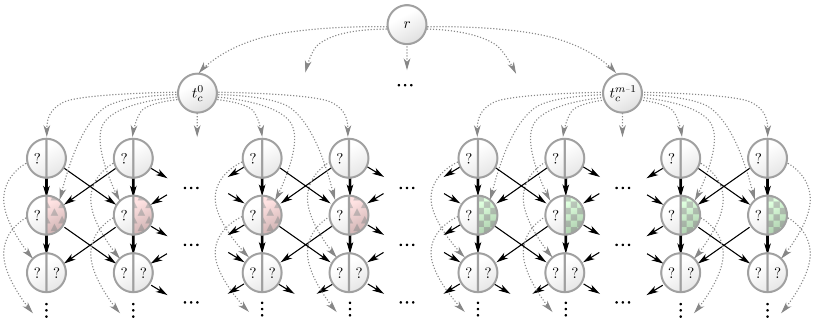
(a) *Structure of the task graph*



(b) *Distribution of initial data on multiple nodes*



(c) *Imbalance due to uneven distribution of auxiliary tasks*



(d) *High number of tasks created by each auxiliary task for task creation, leading to an imbalanced distribution of data*

Figure 8.5: *Influence of the creation of initial tasks on data locality*

As a result, it is possible that these workers steal more auxiliary tasks for task creation than others, resulting in a situation such as the one illustrated in Figure 8.5b, where the amount of data varies between nodes. This leads to imbalance at the beginning of the execution and can cause contention on the nodes containing more input buffers than others.

Different delays for work-stealing are not the only possible cause for data distributions favoring node contention. Such an imbalance can also arise from task creation itself if the number of tasks created by an auxiliary task is not well chosen. This is shown in Figure 8.5d, in which only $m < n$ auxiliary tasks t_c^0 to t_c^{m-1} each create significantly more initial tasks than in the previous figures. While the number of tasks and the actual set of tasks created by each auxiliary task might be easy to determine for regularly-structured task graphs with similar progress on task execution for each part of the graph, it might be more difficult to develop an appropriate strategy for more complex graphs. For example, although the task graph of the one-dimensional version of the *seidel* benchmark shown in Figure 8.6a is similar to the task graph of *jacobi-1d* it is less obvious to find a pattern for the creation of initial tasks that does not introduce node contention. The main difference with *jacobi-1d* are the dependences between tasks processing data blocks of the same iteration represented by the horizontal arrows. The presence of these dependences is the reason that tasks at the left side of the graph are activated earlier than tasks on the right side. The directions in which the application progresses within the task graph are illustrated by Figure 8.6b. If the auxiliary tasks for task creation create the tasks at the beginning of the paths with heavy dependences in groups of horizontal neighbors the nodes containing the input buffers of tasks at the left side of the graph are targeted more frequently at the beginning of the execution than other nodes.

In all of the cases presented above, immediate allocation plays a key role for initial data distribution, since it determines the placement of the buffers of initial tasks at task creation. Data distribution can depend on the topology of the hardware, on the number of tasks created by each auxiliary task for the creation of initial tasks and on the position of the tasks in the task graph.

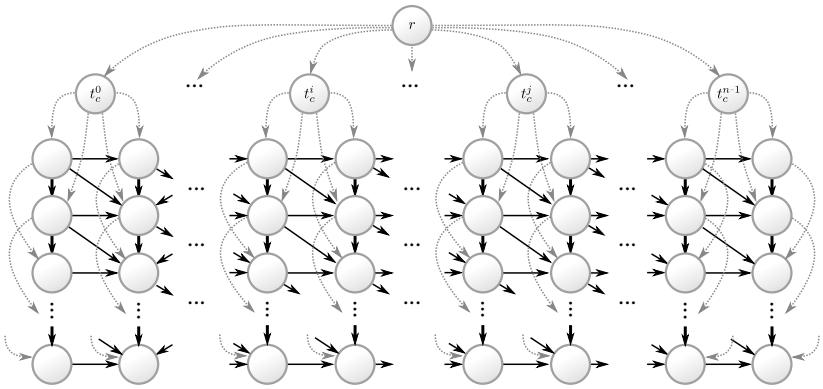
In the following section, we introduce *deferred allocation*, which aims at mitigating the negative effects on data locality and load balancing described above. We first present the principles of this technique and show how we have integrated this mechanism into the OpenStream run-time and the compiler. Afterwards, we illustrate how deferred allocation mitigates the negative effects on data locality and load balancing described above.

8.2 Deferred allocation

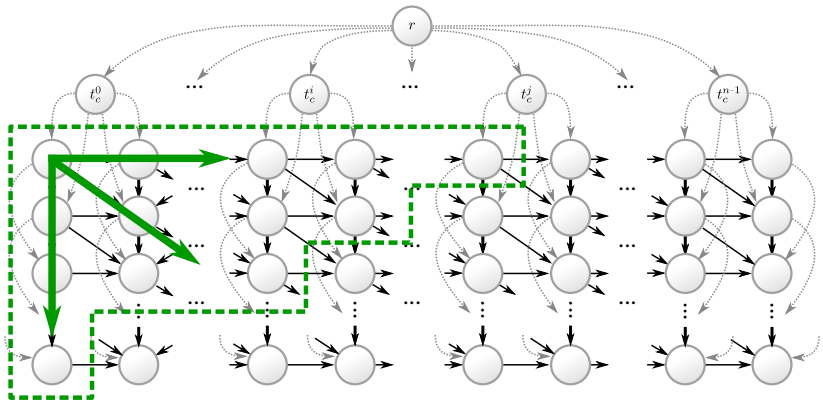
The main drawback of immediate allocation is that decisions for placement of data are made before the location of execution of the task writing the data is known. Work-pushing and topology-aware work-stealing can improve data locality and load balancing, but can only react to a given placement that has been determined in advance. However, even with these techniques enabled, data locality can still depend on the control program in general and the creation of initial tasks in particular.

8.2.1 Principles of deferred allocation

To mitigate these issues, we introduce a strategy for memory allocation that we refer to as *deferred allocation*. The key idea of deferred allocation is to delay the allocation and thus the placement of each input buffer as long as possible, until the node executing the producer that writes to the buffer is known. Hence, instead of allocating input buffers of a task upon its creation during execution of the creating task, each input buffer is allocated by the task that writes to it. If multiple producers write to the same input buffer, the buffer is allocated during the call to `resolve_dependences` that matches the first output view with the input view, which avoids

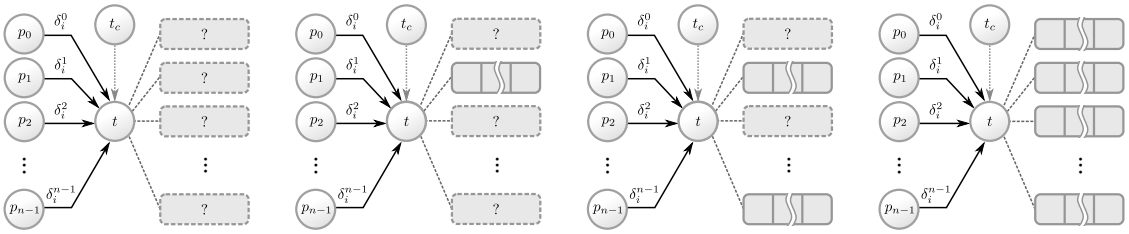


(a) *Task graph of the one-dimensional version of* seidel



(b) *Progress within the task graph*

Figure 8.6: *Example of a task graph that requires a less obvious scheme for the creation of initial tasks to avoid contention*



(a) *Task creation* (b) *Activation of* p_1 (c) *Activation of* p_{n-1} (d) *After execution of all producer tasks*

Figure 8.7: *Example of deferred allocation of the input buffers of a task* t *with* n *producers*

concurrent allocations or synchronization during execution of the producers of a task [2]. Hence, deferred allocation is used only for the input buffers of input views that are matched with a single output view, which is the case for a large majority of views in our benchmarks.

Figure 8.7 illustrates deferred allocation on the task graph of Figure 8.1a. At the creation of t only its data-flow frame is allocated, but none of its input buffers, as indicated by the question marks in Figure 8.7a. Let p_1 be the first of the producers of t that is activated. Before the instructions of the task body can be executed, the input buffer of t receiving the output data of p_1 must be allocated. The state after the allocation of this input buffer is shown in Figure 8.7b. The same procedure is repeated for each of the producers when these become ready for execution. As there is either a one-to-one mapping of the producers output views to the input views or an input buffer was already allocated upon the call to `resolve_dependences`, the deferred allocations can be carried out in parallel without any overhead for locking or synchronization. When t becomes ready for execution, all producers have terminated and all input buffers of t are guaranteed to be allocated as shown in Figure 8.7d.

8.2.2 Modification of the run-time

The implementation of deferred allocation requires changing the data structures and code of the run-time. When using immediate allocation, input buffers are managed by the creating task as well as the task that is created: the creating task allocates the input buffers and the newly created task frees these buffers when it terminates. During dependence resolution, the data pointers of the output views of a producer are set to addresses within the address ranges of the input buffers of the consumers. Hence, neither the consumer task, nor the consumer view are known by an output view. When using deferred allocation, however, allocation of input buffers is under the responsibility of the producer. When the producer allocates the input buffer for a consumer, the data pointer of the input view of the consumer must be set as well as the data pointer of the output view. Thus, the producer must be aware of the address of the structure representing the input view.

We have therefore modified the data structure representing a view in order to store the address of the input view with which an output view has been matched during the call to `resolve_dependences`. When deferred allocation is triggered, a new input buffer is allocated and its address is made available to the producer view and the consumer view. The function that carries out these steps is *prepare_data* with the pseudo-code of Algorithm 8. The function first determines whether the data pointer of the output view v_o already points to the input buffer of the consumer, which is the case for output views providing access to the elements of an input view with multiple producers. If the input buffer has not been allocated before, the function obtains a new buffer from

2. Restriction 4.1 in Section 4.3.1 defines a one-to-one matching of input and output views. However, in a few cases a pattern where multiple producers write to a single input view is still useful and does not have significant impact on performance. For example, the benchmarks studied in this thesis use a data-flow style barrier at the end of the execution, in which a task receives integer tokens from all tasks performing the last iteration on a block of data and the root task synchronizes with this task using a taskwait construct.

Algorithm 8: prepare_data(v_o)

```
1  if v_o.data ≠ null then
2      w ← this_worker()
3      node_w ← local_node_of_worker(w)
4      pool ← memory_pool_of(node_w)
5      v_i ← v_o.consview
6      v_i.data ← alloc(pool, v_i.horizon)
7      v_o.data ← v_i.data
8  end
```

Algorithm 9: prepare_data_vec(v_v, num)

```
1  v_a ← view_array_base(v_v)
2
3  for i ∈ {0, ..., num − 1} do
4      v_o ← v_a[i]
5      prepare_data(v_o)
6  end
7
8
9
```

the memory pool of the node on which the calling worker executes. This requires several steps. In Line 2, the function first determines the identifier of the calling worker. This identifier is passed to *local_node_of_worker* in order to obtain the identifier of the worker's NUMA node (Line 3). The node identifier is needed to obtain a reference to the worker's memory pool (Line 4). The actual allocation takes place in Line 6, in which the return value of a call to *alloc* is assigned to the data pointer of the input view of the consumer. The same value is assigned to the data pointer of the output view afterwards.

Deferred allocation for multi-dimensional views are treated by another function called *prepare_data_vec*, shown in Algorithm 9. The implementation of this function is straightforward: for each individual view *prepare_data_vec* simply issues a call to *prepare_data*.

Figure 8.8 shows the effects of deferred allocation on the data-flow frames of t and p_1 of Figure 8.7. After the creation of t in Figure 8.8a, all of the data pointers of the input views of t are initialized to `NULL`. When p_1 is created, the run-time calls `resolve_dependences` for its output view, which causes the pointer `consview` to be set to the address of the respective input view of t (Figure 8.8b). The data pointers for both the output view of p_1 and the input view of t are still set to `NULL`. Figure 8.8c shows the state of the data structures at the beginning of the execution of p_1. The address of the input buffer allocated during the call to `prepare_data` has been assigned to the data pointer of the output view of p_1 and to the data pointer of the respective input view of t.

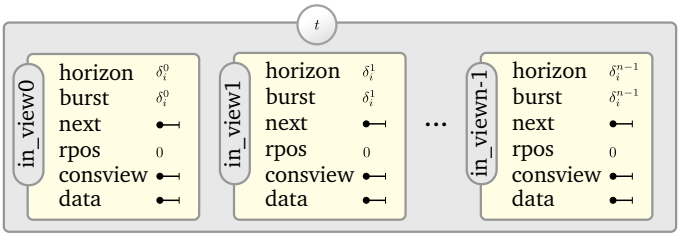
8.2.3 Modification of the compiler

To guarantee that a producer task executes correctly, the calls to `prepare_data` must be issued at the very beginning of the execution of each task that has at least one output view. To this end, we have modified the compiler, such that it generates a call to `prepare_data` for each output view and a call to `prepare_data_vec` for each multi-dimensional view during processing of input and output clauses.

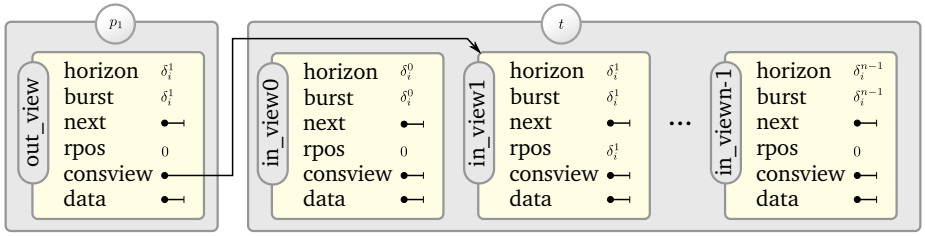
The following listing is used as an example to illustrate the result of the translation with the modified OpenStream compiler supporting deferred allocation.

Listing 8.1: Task with output dependences causing the modified compiler to add calls to prepare_data to the task body

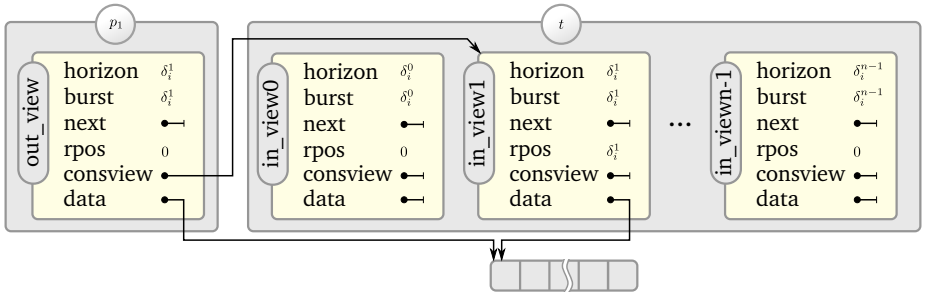
```
1  void stream_function(void)
2  {
3    ...
4
5    int horizon = 10;
6
7    float out_view_sq[horizon];
8    float out_view_sqrt[horizon];
9    float in_view[horizon];
10
11   #pragma omp task input(fstream >> in_view[horizon]) \
12                    output(sq_stream << out_view_sq[horizon], \
13                    sqrt_stream << out_view_sqrt[horizon])
14   {
15     for(int i = 0; i < horizon; i++) {
16       out_view_sq[i] = in_view[i]*in_view[i];
17       out_view_sqrt[i] = sqrtf(in_view]);
```



(a) *After creation of* t



(b) *Matching of* t *with* p_1



(c) *Allocation of the input buffer upon activation of* p_1

Figure 8.8: *Immediate allocation of input buffers*

```
18        }
19      }
20
21      ...
22    }
```

For each of the task's output views a call to `prepare_data` must be added at the beginning of the output views as shown in the resulting outlined task body in the listing below.

Listing 8.2: General lines of the code with deferred allocation generated by the compiler

```
1   void work_function_1(struct frame_1* fp)
2   {
3     prepare_data(fp->out_view_sq);
4     prepare_data(fp->out_view_sqrt);
5
6     for(int i = 0; i < fp->horizon; i++) {
7       ((float*)fp->out_view_sq.data)[i] = ((float*)fp->in_view.data)[i] *
8                                           ((float*)fp->in_view.data)[i];
9       ((float*)fp->out_view_sqrt.data)[i] = sqrtf(((float*)fp->in_view.data)[i]);
10    }
11
12    tdecrease(fp->out_view_sq.owner, fp->out_view_sq.horizon);
13    tdecrease(fp->out_view_sqrt.owner, fp->out_view_sqrt.horizon);
14    tend(fp);
15  }
```

8.2.4 Deferred allocation and work-pushing

Deferred allocation delays the allocation of output buffers to the moment when the core executing a producer task is known. This improves the locality of write accesses when a producer task is executed as shown below, but it does not address the locality of read accesses. Hence, data locality might still need to be improved through work-pushing. When using immediate allocation, the addresses of input buffers *and* output buffers of a task are known when the task becomes ready for execution and the work-pushing heuristics presented in Section 7.2 can use both information on the placement of input buffers and information on the placement of output buffers to decide which worker should become the owner of the task. When using deferred allocation, this is not necessarily the case, since the location of a subset or all of the output buffers of a task might be determined only at the beginning of the execution of the task and can thus be unknown at the moment the task becomes ready. The entire set of addresses of output views of a a task is known when the task becomes ready only if all output views provide access to input views with multiple writers, which occurs only very rarely. Hence, in most cases, the *output only* heuristic and the *weighted* heuristic could only operate with incomplete information on the placement of output buffers. In this case, the *output only* heuristic would behave like the default task activation mechanism in which the worker satisfying the last dependence of a task becomes its owner and the *weighted* heuristic would behave as the *input only* heuristic.

However, due to deferred allocation, the run-time can assume that the output buffers will be placed on the node of the worker executing the task and it is sufficient for data locality to base the transfer decision on the placement of the input buffers of a task. Thus, we have only implemented and evaluated the *input only* heuristic for work-pushing in conjunction with deferred allocation.

8.3 Influence of deferred allocation on data locality

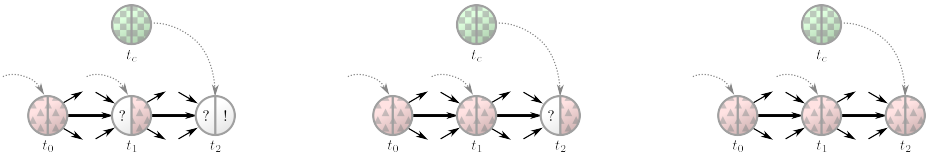
After the definition of the principles and the integration of deferred allocation into the OpenStream run-time, this section shows how these modifications impact the locality of accesses compared to immediate allocation.

8.3.1 Influence of the control program

Figure 8.9 shows the placement of input buffers in a chain of tasks with heavy dependences and the creation of a task in the chain by an unrelated task which is not a predecessor in the task graph, as already illustrated for immediate allocation in Figure 8.2d and Figure 8.2e. Input buffers that have not been allocated, but whose associated tasks have already been created, are labeled with an exclamation mark. This is the case for t_2 in Figure 8.9a, which has been created by t_c, but whose input buffer holding the majority of its input data has not yet been allocated. The allocation takes place when t_1 becomes active, as shown in Figure 8.9b. In contrast to immediate allocation, the majority of the memory accesses are local, although t_2 has been created by a task executed by a remote worker. The only accesses that can target remote memory are read accesses to the input buffers associated to light dependences if the corresponding producers have been executed by remote workers. However, these buffers only hold a small fraction of input data, such that read and write accesses can be considered as entirely local. For unbalanced dependences, deferred allocation thus effectively decouples data locality from task creation by the control program.

The decoupling also applies to the task graph of Figure 8.2b with balanced and unbalanced dependences. Figure 8.10 shows the placement of input buffers at execution of the tasks t_c^a and t_c^b, which create t_a, t_s^a, t_b and t_s^b. Half of the input buffers of t_p^a and t_p^b are placed on one node and the other half is stored on another node, while the input buffers of t_a and t_b and their siblings have not been allocated yet. The nodes to which the owners of t_p^a and t_p^b are associated determine where the input buffers of t_a, t_s^a, t_b and t_s^b are allocated. The figures 8.10b, 8.10c, 8.10d and 8.10e show the possible outcomes for the placement.

In Figure 8.10b, the worker executing t_c^b has become the owner of t_p^a and the worker executing

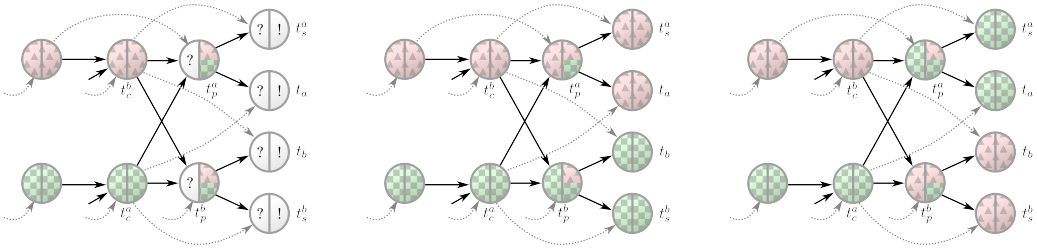


(a) *Creation of t_2*

(b) *Execution of t_1, allocation of the major input buffer of t_2*

(c) *Execution of t_2*

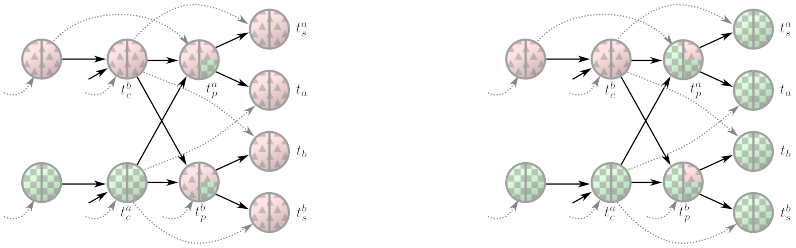
Figure 8.9: *Decoupled control program and buffer allocation on a path of heavy dependences*



(a) *Execution of t_c^a and t_c^b*

(b) *Execution of t_a, t_s^a, t_b and t_s^b*

(c) *Execution of t_a, t_s^a, t_b and t_s^b with swapped owners for t_p^a and t_p^b*



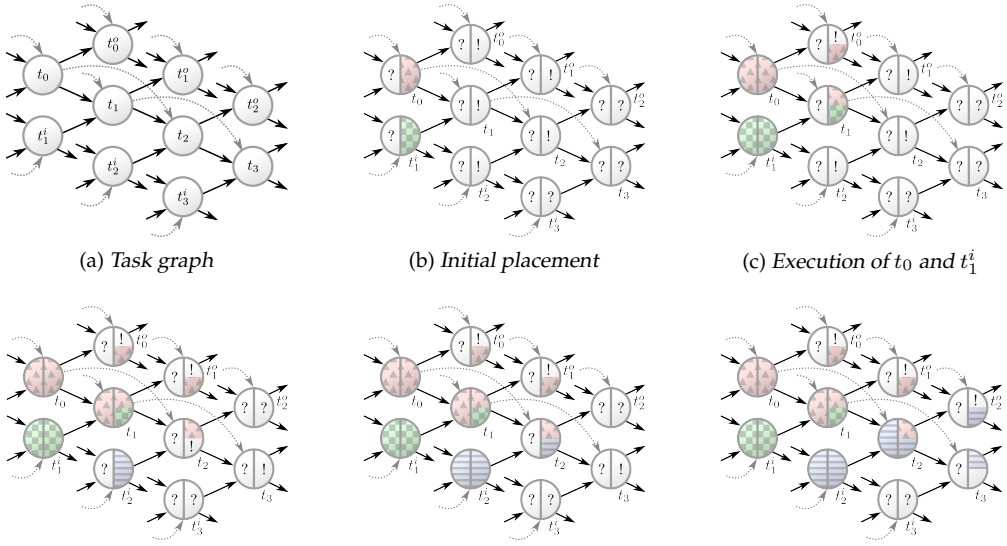
(d) *Execution of t_a, t_s^a, t_b and t_s^b with identical owners for t_p^a and t_p^b*

(e) *Execution of t_a, t_s^a, t_b and t_s^b with identical owners for t_p^a and t_p^b (alternative)*

Figure 8.10: *Decoupled control program and buffer allocation on a path of heavy dependences*

t_c^a has become the owner of t_p^b. This leads to the placement of the input buffers of t_a and t_s^a on the same node as the input buffers of t_c^b and the placement of the input buffers of t_b and t_s^b on the same node as the input buffers of t_c^b. A similar situation is shown in Figure 8.10c, where the owners of t_p^a and t_p^b are swapped. In both results for placement, half of the input buffers of the tasks on the right are placed on one node and the other half is placed on another node, with high locality of memory accesses and well balanced load. However, it is also possible that the same worker becomes the owner of t_p^a and t_p^b as shown in Figure 8.10c and Figure 8.10d. While this has an influence on load balancing across memory controllers, the amount of locally accessed data is the same as in the previous scenarios for data placement and thus remains high.

The previous task graph contains both balanced and unbalanced dependences, with a high amount of the data placed on the node of the worker executing a task. However, the task graphs of applications can contain large sub-graphs that are entirely composed of tasks with unbalanced dependences, such as the *bitonic* benchmark. An excerpt of the task graph of this benchmark is shown in Figure 8.11a. In the following discussion, we concentrate on the chain formed by the tasks t_0 to t_3.



(a) *Task graph* (b) *Initial placement* (c) *Execution of t_0 and t_1^i*

(d) *Execution of t_1* (e) *Execution of t_2^i* (f) *Execution of t_2*

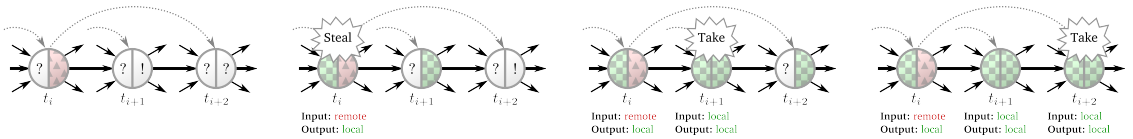
Figure 8.11: *Deferred allocation on a task-graph with balanced dependences*

Figure 8.11b shows a possible initial placement for the input buffers of t_0 and t_1^i. To keep the graphs at the following steps of the execution simple, we assume that the two input buffers of tasks that do not depend on tasks on the chain are both placed on the same node (e.g., t_1^i in Figure 8.11b and t_2^i in Figure 8.11d). The execution of t_0 and t_1^i leads to the allocation of the input buffers of t_1 as shown in Figure 8.11c. As implied by the balanced dependences, half of the input buffer is placed on the node of the worker executing t_0 and the remaining input data is placed on the node executing t_1^i. During execution of t_1 in Figure 8.11d, the input buffers of t_1^o and t_2, receiving the output of t_1, are allocated locally on the node of the worker executing t_1. Three quarters of the data involved in the execution of t_1 are thus accessible locally. At execution of t_2^i in Figure 8.11e, the remaining input buffer of t_2 is allocated on the node of the worker executing t_2^i. In Figure 8.11f, the amount of locally placed data at execution of t_2 is identical to the amount of locally placed data during execution of t_1: three quarters of the data involved in the execution are accessible on the node executing the task. In contrast to immediate allocation, deferred allocation is thus able to provide high data locality even for tasks with balanced dependences.

8.3.2 Influence of work-stealing

Another important effect of deferred allocation is the improvement of data locality in case of work-stealing events by remote workers. An example for such a situation is provided in Figure 8.12. The initial placement for the input buffers of t_i in Figure 8.12a is identical to Figure 8.3a of Section 8.1.2, used for the illustration of work-stealing in conjunction with immediate allocation. As t_i is not yet ready for execution, the input buffers of t_{i+1} have not been allocated. The allocation is triggered after the steal and execution of t_i by a remote worker, as shown in Figure 8.12b. The deferred allocation on the local node of the stealing worker causes only read accesses to target remote memory and all write accesses are local. If t_i satisfies the last dependence of t_{i+1} and if t_{i+1} is the producer of t_{i+2} that finishes last, t_{i+1} and t_{i+2} are also executed by the worker that initially stole t_i as shown in Figure 8.12c and Figure 8.12d. However, even if this is not the case, data locality can be restored using *any* of the work-pushing heuristics, as these all transfer t_{i+1} and t_{i+2} to a node for which all of the accesses to memory target the node's local memory.

Deferred allocation generally improves the locality of accesses to main memory of tasks stolen by remote workers. As the output buffers of tasks are guaranteed to be allocated locally, the



(a) *Initial placement* (b) *Steal of t_i by a remote worker* (c) *Execution of t_{i+1}* (d) *Execution of t_{i+2}*

Figure 8.12: *Work-stealing in conjunction with deferred allocation*

fraction of data located on the same node as the worker executing the task is at least as high as the fraction of output data over the sum of the size of all data, i.e. output *and* input data. Let t be a task with n input dependences of size δ_i^0 to δ_i^{n-1} and m output dependences of size δ_o^0 to δ_o^{m-1}. Using deferred allocation, the fraction of locally placed data f_{loc} is:

$$f_{\text{loc}} \geq \frac{\sum_{j=0}^{m-1} d_o^j}{\sum_{k=0}^{n-1} d_i^k + \sum_{j=0}^{m-1} d_o^j}$$

Of course this definition is only valid if each node is provided with sufficient memory and if all input buffers allocated from the memory pool associated to a node are placed on the node.

8.3.3 Creation of initial tasks

The decoupling of the execution location of a creating task from the placement of the input buffers of the created task also has a positive impact on load balancing across memory controllers upon creation of initial tasks. Neither the number of tasks created by each auxiliary task, nor faster steals of auxiliary tasks by workers on nodes with faster accesses to the work-deque of the worker executing the root task can lead to an imbalanced initial placement of input buffers as seen for immediate allocation in Section 8.1.3.

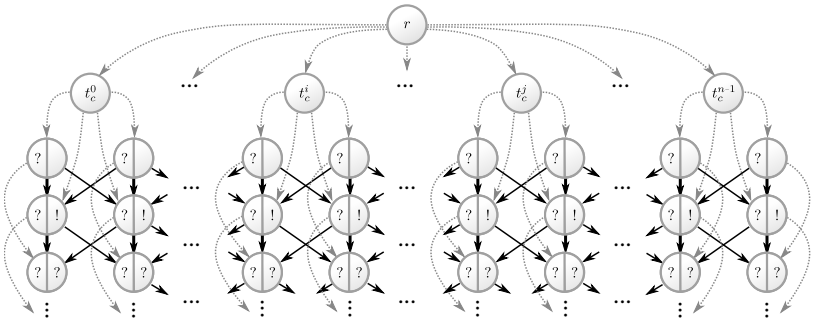
Consider Figure 8.13a and Figure 8.13b with different number of auxiliary tasks for the creation of initial tasks. As the placement of input buffers of the initial tasks is only determined upon execution of the predecessors, the outcome is the same for both task graphs and only depends on the timing of task execution. As in the illustration for immediate allocation, the right semi-circles of tasks at the beginning of the chains with heavy dependences are empty due to the absence of input buffers for these tasks. A possible scenario for the placement is shown in Figure 8.13c. Due to the decoupling between the locations of execution of creating tasks and the placement of the input buffers of the created tasks, as well as the improvement of the data locality upon work-stealing by remote workers, the input buffers of initial tasks that have been created by the same auxiliary task are not necessarily placed on the same node. Figure 8.13d illustrates the same data placement as in Figure 8.13c but for the task graph of Figure 8.13b, in which more than two tasks are created by each auxiliary task for task creation.

As a result of deferred allocation, load balancing across memory controllers is not predetermined and only depends on computational load balancing [3]. Workers that run out of work steal additional auxiliary tasks and thus cause more data to be placed on the associated node.

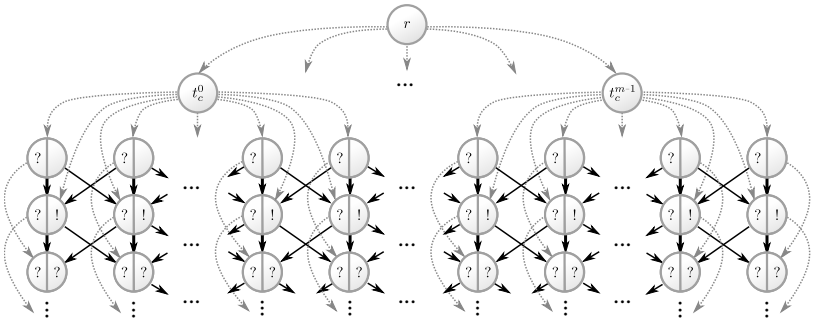
8.3.4 Reduction of the memory foot print

Delayed allocation of input buffers when using deferred allocation does not only increase data locality of write accesses, but also has an important side effect on the memory footprint of the application. When using immediate allocation on a path of dependent tasks $t_0, \ldots, t_n$, where t_i creates t_{i+2}, at least three buffers are in use at any time between the execution of t_1 and the execution of t_{n-1}. Figure 8.14a, showing the execution of a task t_i with $0 < i < n-1$, illustrates

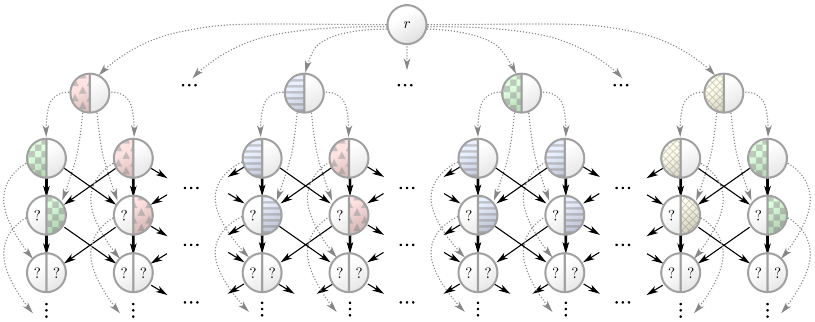
3. However, the placement of the metadata stored in data-flow frames is still determined by the task creation, but the size of the metadata is small compared to the actual data stored in input buffers.



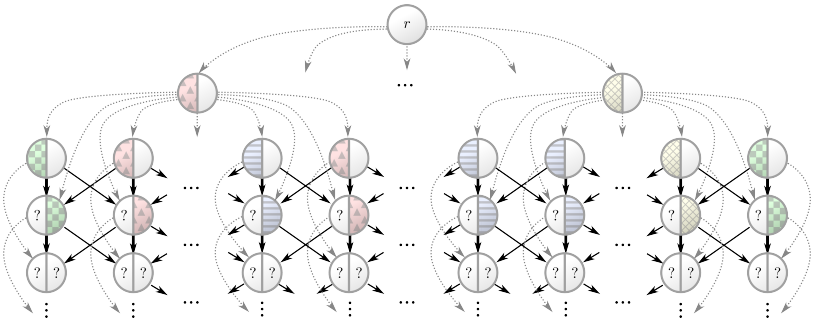
(a) *Deferred allocation with two tasks created by each auxiliary task for task creation*



(b) *Deferred allocation with more than two tasks created by each auxiliary task for task creation*

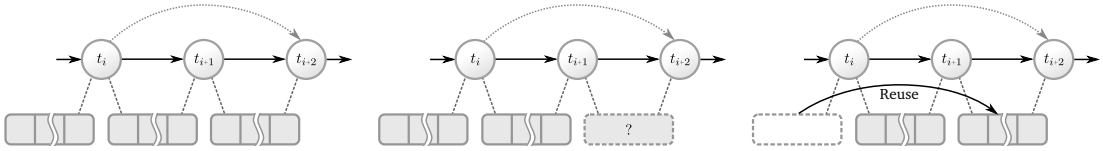


(c) *Possible outcome for the placement of input buffers of initial tasks when creating two tasks with each auxiliary task for task creation*



(d) *Another possible outcome for the placement of input buffers of initial tasks when creating more than two tasks with each auxiliary task for task creation*

Figure 8.13: *Improved data locality and load balancing resulting from the creation of initial tasks using deferred allocation*



(a) *Immediate allocation* (b) *Deferred allocation* (c) *Deferred allocation, reuse of the input buffer of* t_i

Figure 8.14: *Deferred allocation compared to immediate allocation*

this property. The first buffer, which contains the input data of t_i cannot be freed until t_i terminates. The second buffer, which is the output buffer of t_i and which corresponds to the input buffer of t_{i+1} can be freed earliest at termination of t_{i+1}, which happens after termination of t_i. As t_i creates t_{i+2}, the input buffer of t_{i+2} is allocated during execution of t_i and thus before termination of t_i.

Figure 8.14b illustrates the same setting using deferred allocation. A first difference to the previous scenario is that the allocation of the input buffer of t_{i+2} is delayed until the beginning of the execution of t_{i+1}. When t_{i+1} becomes ready, t_i has terminated, since termination of t_i is a prerequisite for activation of t_{i+1}. The input buffer of t_i is not referenced any more and can be reused for t_{i+2} as shown in Figure 8.14c. Hence, deferred allocation reduces the number of buffers that have to exist simultaneously on a chain of dependent tasks by one.

Figure 8.15 on page 174 provides a more detailed view of the events related to memory management when using deferred allocation. Figure 8.15a shows the initial state of the memory pools of two workers w_r, executing the auxiliary task that creates t_0 and t_1, and w_e, executing the chain of dependent tasks. For simplicity, the task graph only shows the heavy dependences between the tasks on the chain. We assume that w_r and w_e are associated to different nodes and thus use different memory pools. Each pool has a list of free data-flow frames, containing only the metadata of a task, and a list of free input buffers. Let s_f be the size of objects corresponding to frames and let s_v be the size of objects in the pool corresponding to the size of the input buffers used by the tasks on the chain. Initially, all free lists are empty, as shown in Figure 8.15a. When execution of the auxiliary task for task creation starts, t_0 is created and w_r refills the free list for frames in its local memory pool (Figure 8.15b). A refill for the list of input buffers is not necessary as t_0 does not have any input dependences. Next, the data-flow frame of t_0 is initialized, but the task remains blocked due to the missing consumer t_1 (Figure 8.15c). When t_1 is created, t_0 becomes ready (Figure 8.15d) and can be stolen by w_e [4] (Figure 8.15e). At the beginning of the execution of the task body of t_0, the input buffer of t_1 is allocated. This is shown in Figure 8.15f, where w_e carries out a refill operation on the list of free input views. The input view is taken from the free list (Figure 8.15g) and t_2 is created, which causes a refill of the list of free frames of w_e, followed by a remove of the first free object from that list in Figure 8.15h. The task t_0 then finishes execution and activates t_1, which allows w_e to free the frame of t_0 (Figure 8.15i). Upon execution of t_1 deferred allocation triggers and removes an objects from the list of free views of w_e (Figure 8.15j). At this point, no further refill operations are necessary, since three frames and two views can be used at the same time. Hence, upon creation of t_3, the frame of t_0 can be reused (Figure 8.15k) and the input buffer of t_1 can be reused as the input buffer of t_3 upon deferred allocation by t_2 (Figure 8.15l).

The total amount of memory that can be saved due to the reduction of the number of co-existing input buffers during execution of tasks on a chain directly depends on the size and structure of the task graph. For a high number of chains with inter-chain dependences that prevent chains from executing to the end before executing tasks from neighboring chains, such as in *seidel*, the *jacobi* benchmarks or the *bitonic* sorting network, the memory that can be saved directly depends on the number of chains present in the task graph. Using deferred allocation, the number of refill

4. In reality t_0 stays in the software cache of w_r and cannot be stolen at this point. However, each auxiliary for task creation task usually creates and unblocks more than a single task, such that t_0 is quickly transferred to the work-deque of w_r and gets exposed to steals.

operations on memory pools can be reduced, which results in a lower memory footprint of the application and which reduces overhead for the initialization of buffers upon physical allocation.

8.4 Experimental results

In this section, we evaluate the impact of deferred allocation using the benchmarks of Section 6.1. We first analyze the locality of memory accesses as well as the reduction of the applications' memory footprints before showing the impact on performance. The following abbreviations are used to identify the different configurations:

- *rnd* refers to default random work-staling, immediate allocation and without work-pushing or topology-aware work-stealing,
- *dfa* refers to deferred allocation only without work-pushing or topology-aware work-stealing,
- *dfa+input only* is used for deferred allocation and work-pushing with the *input only* heuristic and,
- *dfa+input only+taws* refers to deferred allocation with work-pushing and topology-aware work-stealing.

Due to the excessive memory footprint of the *cholesky* benchmark without broadcast-specific optimizations, the results for this benchmark will be presented in Chapter 9.

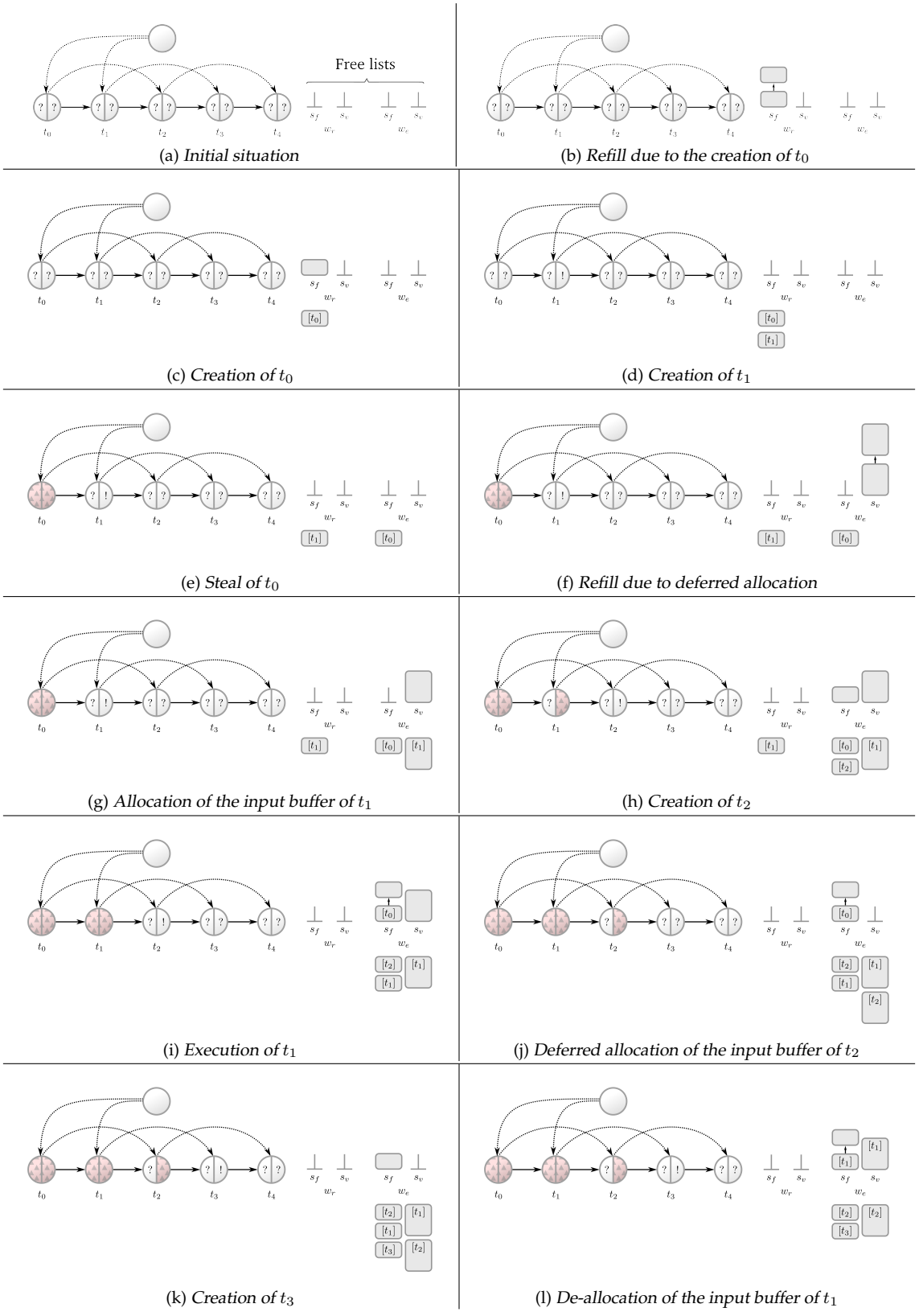
Data locality

Figure 8.16 shows the fraction of requests to local main memory over the total number of requests to main memory for the execution of our benchmarks on the Opteron platform. Thanks to the high locality of write accesses, deferred allocation only already improves data locality significantly for all of the benchmarks. For *seidel*, *jacobi-1d*, *jacobi-2d*, *jacobi-3d*, *k-means* and *bitonic* about two thirds of all requests target local memory. The low locality of memory accesses in the *blur-roberts* benchmark can be explained with the fact that its task graph only contains very short paths of heavy dependences. Thus, accesses to the input matrix and the output matrix in shared memory with interleaved allocation, as explained in Section 5.3.4, have a large impact on data locality.

Work-pushing using the *input only* heuristic can further improve data locality of all benchmarks with the highest improvements for *seidel*, *jacobi-1d*, *jacobi-2d*, *jacobi-3d*, and *k-means*. More than 85% of the requests to main memory of these benchmarks target local memory. For *blur-roberts* the improvement is much lower due to the influence of the matrix in shared memory explained above. For the *bitonic* benchmark, work-pushing only yields a minimal increase of data locality. This is due to the fact that the activating workers of the vast majority of tasks are workers that provided half of the tasks' input data, such that in most cases work-pushing does not initiate a transfer to a remote worker. The difference between the default task activation mechanism and work-pushing is thus minimal.

Except for *blur-roberts*, topology-aware work-stealing increases the locality of accesses to main memory for all benchmarks. For *seidel*, *jacobi-1d*, *jacobi-2d*, *jacobi-3d* and *k-means* deferred allocation combined with work-pushing and topology-aware work-stealing leads to more than 90% of the requests to main memory that target local memory and thus yields almost maximum locality. For *bitonic*, this value is lower due to the balanced dependence pattern, but still reaches a value that is greater than 75%.

As in the analysis of the results for data locality for the different work-pushing heuristics and topology-aware work-stealing in the previous chapter, we cannot provide results for the locality of accesses to main memory for the SGI platform gathered using hardware performance counters due to the lack of support by the operating system. In addition, deferred allocation makes it impossible to determine the location of output buffers *before* execution of a task. Hence, the approximation of the fraction of memory accesses to local memory $R_{\text{loc}}^{\text{appr}}$ as defined in Section 7.4.1 cannot be calculated based on placement information gathered before task execution. Therefore,



(a) *Initial situation*

(b) *Refill due to the creation of t_0*

(c) *Creation of t_0*

(d) *Creation of t_1*

(e) *Steal of t_0*

(f) *Refill due to deferred allocation*

(g) *Allocation of the input buffer of t_1*

(h) *Creation of t_2*

(i) *Execution of t_1*

(j) *Deferred allocation of the input buffer of t_2*

(k) *Creation of t_3*

(l) *De-allocation of the input buffer of t_1*

Figure 8.15: *Illustration of the reduced memory footprint due to deferred allocation*

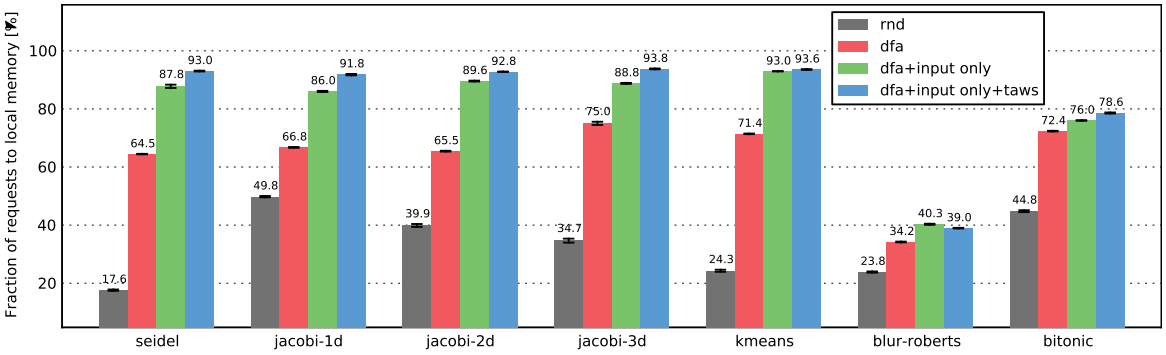


Figure 8.16: *Locality of requests to main memory on the Opteron system for deferred allocation*

we introduce another approximation of the fraction of accesses to local memory called $R_{\text{loc}}^{\text{wloc}}$ whose definition is similar to $R_{\text{loc}}^{\text{appr}}$, but which assumes that all write accesses target local memory.

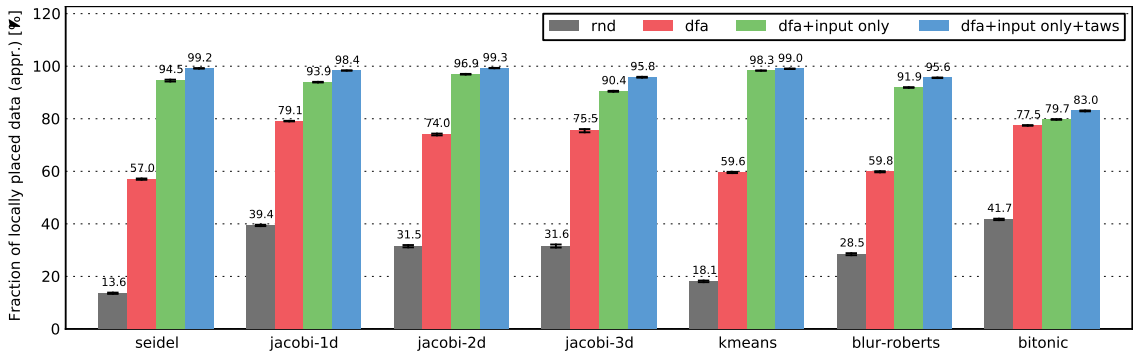
Figure 8.17 shows the results for the locality based on the approximation $R_{\text{loc}}^{\text{wloc}}$ for the Opteron system and the SGI platform. The value indicated for *rnd* corresponds to the estimation using $R_{\text{loc}}^{\text{appr}}$, as deferred allocation is not enabled for this configuration and write accesses thus cannot be assumed to be local. The relative error of the approximation over the locality measured with hardware performance counters for the Opteron system is given in Figure 8.18. As in the previous chapter, the error for *blur-roberts* is extremely high and the values for the approximation for this benchmark cannot be taken into account. The error for the other benchmarks is much lower and ranges between -25.5% and 18.5%. For deferred allocation with work-pushing and deferred allocation with work-pushing and topology-aware work-stealing, the relative error is below 10%.

The results for the SGI system are similar to the Opteron system. Deferred allocation only already improves the locality of memory accesses significantly for all of the benchmarks. Work-pushing increases the locality further for all benchmarks except *bitonic*. The increase of topology-aware work-stealing is lower, but never decreases locality. Except *bitonic*, all benchmarks reach a value for locality based on the approximation $R_{\text{loc}}^{\text{wloc}}$ which is close to the maximum of 100%. For the *bitonic* benchmark more than three quarters of the data are accesses locally. As mentioned above, the values of $R_{\text{loc}}^{\text{wloc}}$ for *blur-roberts* should not be taken into account. Hence, we do not provide any conclusion based on this data for *blur-roberts*.

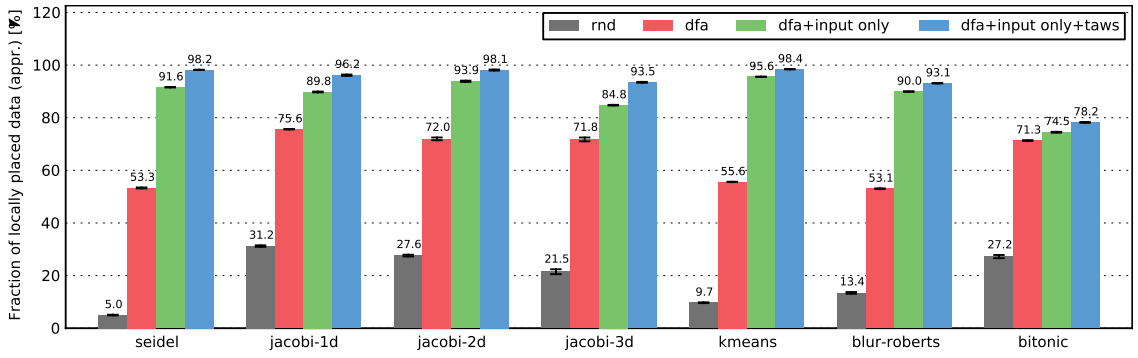
To relate the results for the locality of memory accesses for deferred allocation with the results of the previous chapter on work-pushing and topology-aware work-stealing, Figure 8.19 shows the fraction of requests to local main memory on the Opteron platform for the work-pushing heuristics in conjunction with topology-aware work-stealing and deferred allocation with work-pushing and topology-aware work-stealing. A notable result is that deferred allocation with work-pushing and topology-aware work-stealing yields approximately the same locality as the work-pushing heuristic with the highest locality for all benchmarks for most of the benchmarks. For the *bitonic* benchmark, deferred allocation even yields a significantly higher locality than all of the work-pushing heuristics with topology-aware work-stealing. Hence, in all cases, independently from the structure of the task graph, it is beneficial for data locality to use deferred allocation in conjunction with work-pushing and topology-aware work-stealing.

8.4.1 Memory footprint

As discussed in Section 8.3.4, deferred allocation can reduce the memory footprint of an application considerably. Figure 8.20 shows the maximum resident size of the dynamic single assignment versions with default random work-stealing and different combinations of deferred allocation, work-pushing and topology-aware work-stealing as well as for the shared memory implementations of the benchmarks. The huge difference between the memory footprint of *blur-*



(a) *Opteron system*



(b) *SGI system*

Figure 8.17: *Approximation R_{loc}^{wloc} (and R_{loc}^{appr} for* rnd*) of the locality for deferred allocation*

roberts on the Opteron system and the SGI system is due to the different size of the input and output matrix on both systems. Also, the shared memory version of *blur-roberts* has a footprint that is substantially higher than the dynamic single assignment versions. This is due to the use of an auxiliary matrix for intermediate results generated by the blur filter, which ensures that elements in the overlapping areas of blocks are not overwritten before all calculations depending on this data have terminated.

A rule of thumb for all other benchmarks is that the dynamic single assignment implementations either have a significantly larger footprint than the shared memory versions of the benchmarks without deferred allocation (*seidel*, *jacobi-2d*, *jacobi-3d*, *k-means* and *bitonic*) or require approximately the same amount of memory (*jacobi-1d*). In all cases except for *jacobi-1d* and *blur-roberts*, deferred allocation considerably reduces the amount of memory required for execution. The short dependence paths in *blur-roberts* and the structure of the task graph of *jacobi-1d* with few dependences lead to less tasks that are *in-flight* at the same time. Hence, less input buffers are needed simultaneously and the size of the input matrix dominates the memory footprint.

The improvement over *rnd* is shown in Figure 8.21. For *seidel*, *jacobi-2d*, *jacobi-3d*, *k-means* and *bitonic* the reduction is close to or even exceeds 30%. The only negative impact was measured for *jacobi-1d* on the Opteron system, for which deferred allocation can increase the footprint by up to 21.9%. We did not investigate this issue in detail, but we believe that this behavior is related to improved load balancing, causing refill operations on a higher number of memory pools. For *blur-roberts* the improvement is less than 5% on both platforms.

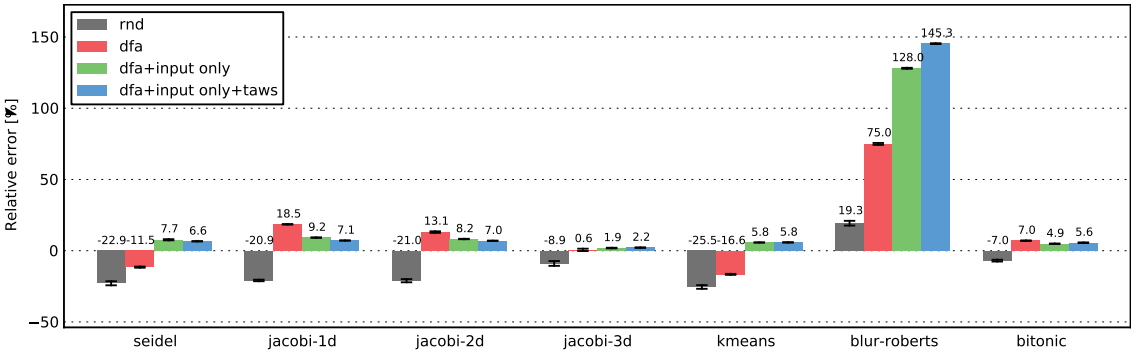


Figure 8.18: *Relative error of R_{loc}^{wloc} (and R_{loc}^{appr} for* rnd*) over the locality measured with hardware performance counters for the Opteron system*

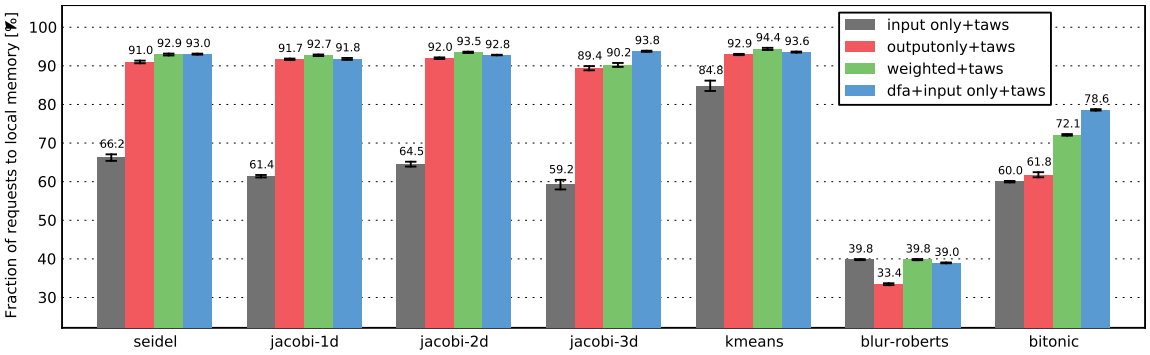
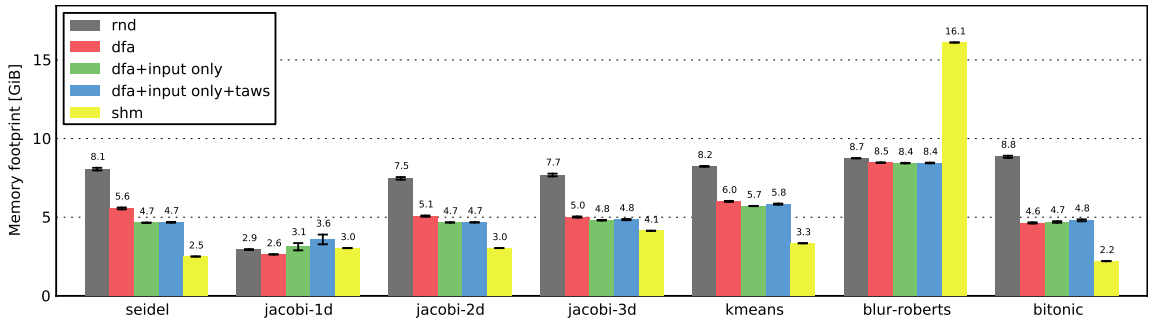


Figure 8.19: *Comparison of the locality of requests to main memory on the Opteron system for work-pushing and deferred allocation*

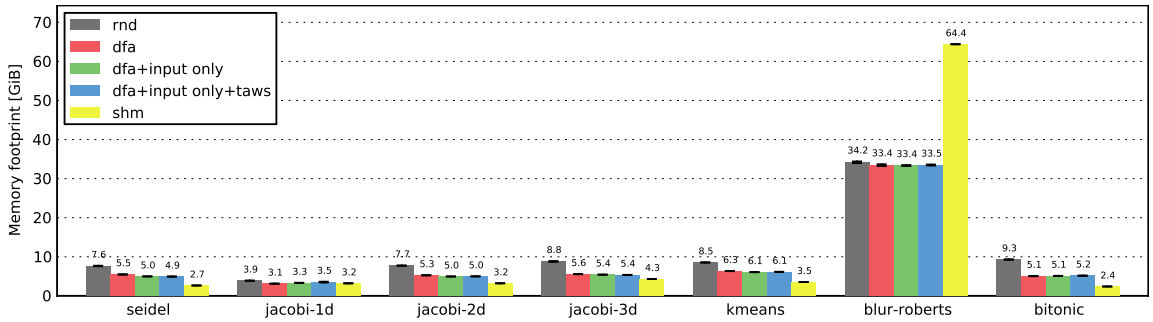
8.4.2 Performance

Figure 8.22 shows the speedup of the different configurations using deferred allocation over random work-stealing. Deferred allocation improves performance of all benchmarks without any exception. The highest speedups are achieved for *seidel* with up to $3.57\times$ on the SGI system and $2.71\times$ on the Opteron system. The smallest speedup is the speedup of the *k-means* benchmark with only $1.01\times$ on the Opteron system and $1.29\times$ on the SGI system. However, the speedup for the best configurations of the other benchmarks is at least $1.38\times$ on the Opteron system and $1.68\times$ on the SGI system. Hence, the improvements on data locality and the reduction of the memory footprint by deferred allocation translate into large performance gains for memory-intensive applications and moderate gains for cache-bound applications.

Figure 8.23 relates the performance of deferred allocation to the shared memory implementations. Except for *k-means* on both platforms and *bitonic* on the Opteron system, the dynamic single assignment implementations with deferred allocation outperform the shared memory implementations at least by a factor of 1.15 for the best configuration. In the best case, i.e., for *jacobi-1d* using deferred allocation and work-pushing on the SGI system, execution can be sped up by a factor of 4.17. A notable difference with the results for work-pushing with topology-aware work-stealing of the previous chapter is that *jacobi-1d* on the Opteron system and *bitonic* on the SGI system now perform better than the shared memory implementations.

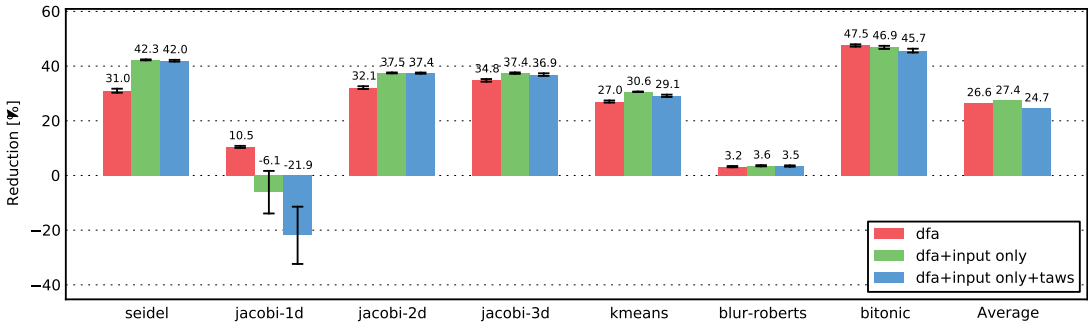


(a) *Opteron system*

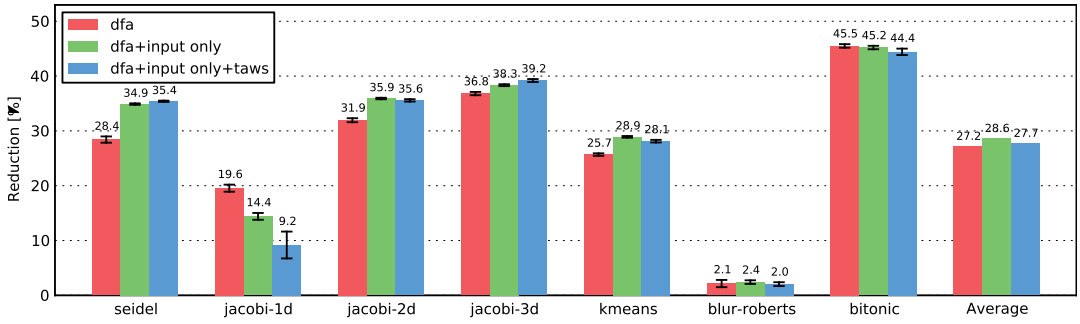


(b) *SGI system*

Figure 8.20: *Maximum resident size for dynamic single assignment implementations with and without deferred allocation and the shared memory implementations*

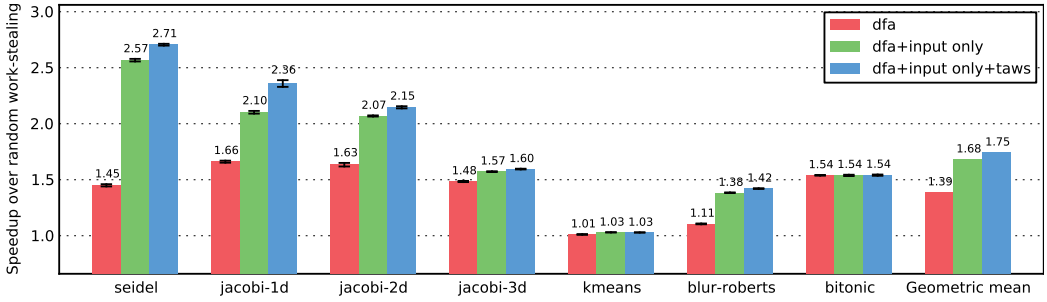


(a) *Opteron system*

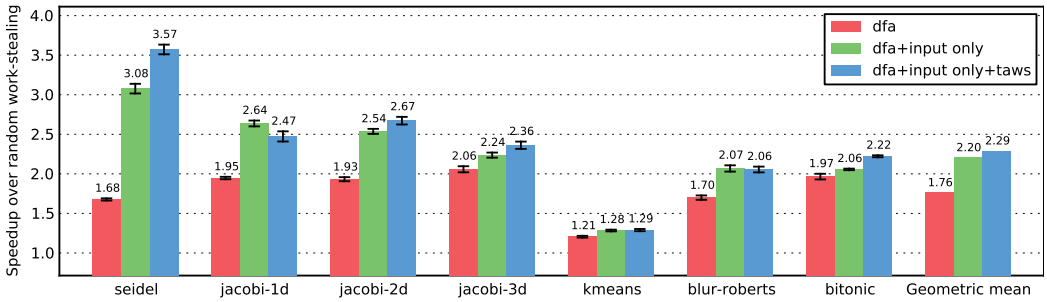


(b) *SGI system*

Figure 8.21: *Reduction of the maximum resident size by deferred allocation compared to* rnd

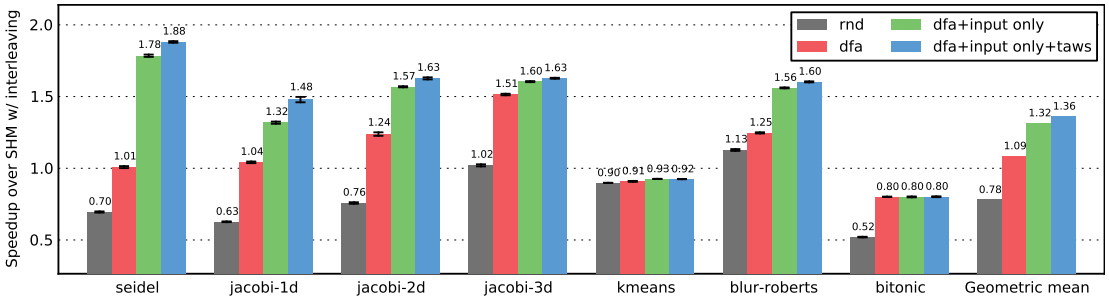


(a) *Opteron system*

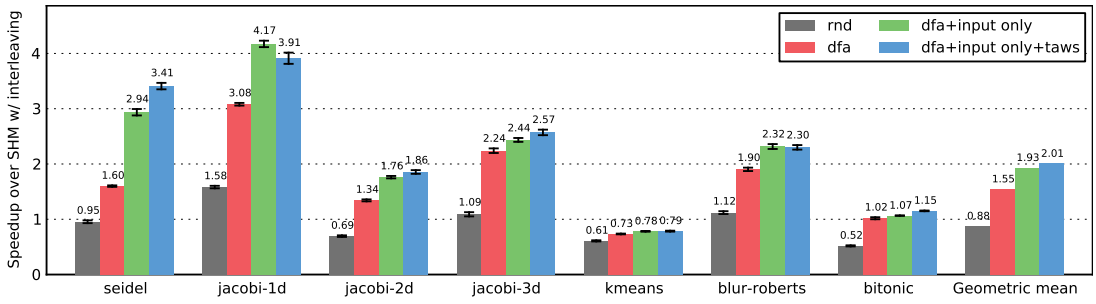


(b) *SGI system*

Figure 8.22: *Speedup of deferred allocation over default random work-stealing without work-pushing*



(a) *Opteron system*



(b) *SGI system*

Figure 8.23: *Speedup of deferred allocation over the shared memory implementations*

8.5 Ongoing work: reduction of the memory footprint with the inout_reuse clause

A major drawback of dynamic single assignment is the possibly larger working set of each task compared to shared memory solutions with in-place updates even with deferred allocation. This has two consequences. The first consequence is a possibly negative impact on the memory footprint of the application, as shown in the previous section. Second, as data of both input and output buffers occupies caches, tasks using dynamic single assignment have a higher footprint in the hierarchy of caches, resulting in a higher number of cache misses. In this section, we present a new clause that we have implemented for OpenStream, called inout_reuse. This clause allows tasks that read data from an input buffer, process the data and write the results to an output buffer of the same size to be rewritten such that they only use a single buffer with in-place updates. As this is ongoing work, we only present the principles of this solution as a perspective, but do not provide conclusive results. The syntax of the inout_reuse clause is the following:

```
1 inout_reuse(stream_expr >> view_expr >> stream_expr)
```

The first stream expression defines from which stream the elements are read and the second stream expression defines to which stream the elements are written after the in-place updates in the task body. To illustrate this concept, consider the following task that reads data from a stream `s1`, calculates the square root of each element and writes the result to another stream `s2`. Using dynamic single assignment with an input and an output clause, this task would be specified as follows:

Listing 8.3: Example of a task with equal-sized input and output views

```
1 float in_view[horizon];
2 float out_view[horizon];
3
4 #pragma omp task input(s1 >> in_view[horizon]) \
5                 output(s2 << out_view[horizon])
6 {
7   for(int i = 0; i < horizon; i++)
8     out_view[i] = sqrtf(in_view[i]);
9 }
```

As the size of the input and output view is the same and as the data between these two views is directly related, the task can be rewritten using the inout_reuse clause as:

Listing 8.4: Example of a task using the inout_reuse clause

```
1 float rview[horizon];
2
3 #pragma omp task input(s1 >> rview[horizon] >> s2)
4 {
5   for(int i = 0; i < horizon; i++)
6     rview[i] = sqrtf(rview[i]);
7 }
```

The memory footprint of the task is half of the memory footprint of the implementation using one input and one output clause. As an inout_reuse clause always references two streams, a minimal task graph of an application using this clause must have at least three tasks, as shown in Figure 8.24a. The first task writes the initial data to the stream of the first stream reference in the inout_reuse clause, the second task uses the inout_reuse and the third task reads from the stream corresponding to the second reference in the inout_reuse clause. A task using the inout_reuse clause can be the producer or the consumer of another task using this clause. The only restriction that applies to this construct is that the views of the producer and consumer must have exactly the same size. This allows the run-time system to use a single input buffer that only needs to be handed from the producer to the consumer.

Figure 8.24b shows a task graph with two consecutive tasks using the inout_reuse clause. Between t_1 and t_2 the same input buffer can be reused. Figure 8.25a to Figure 8.25l illustrate the steps at the execution of the application with this task graph. Before we explain each of these

(a) *Minimal example* (b) *Two subsequent tasks using the inout_reuse clause*

Figure 8.24: *Examples of a task graphs with tasks using the inout_reuse clause*

steps in detail, we first provide an overview of additional data structures and additional fields representing views and frames. For simplicity, the figures show only new and existing fields relevant for the illustration of the inout_reuse clause.

To keep the changes for the code carrying out dependence resolution in `resolve_dependences` as little as possible, we have chosen to implement inout_reuse views using two views, one input view and one output view. This allows the run-time to match input and output views with inout_reuse views the same way as it matches input views with output views. We refer to the input view of an inout_reuse view as the *reuse input view* and to the output view as the *reuse output view*. Among the new fields representing a view is a field called `reuse_view`, pointing to the reuse input view of the producer view if this is an inout_reuse view and `refctr`, indicating for a reuse input view by how many views its input buffer is referenced, including the view itself. Frames are also provided with a reference counter, as views embedded in the frame can be referenced beyond the lifetime of the task associated to the frame.

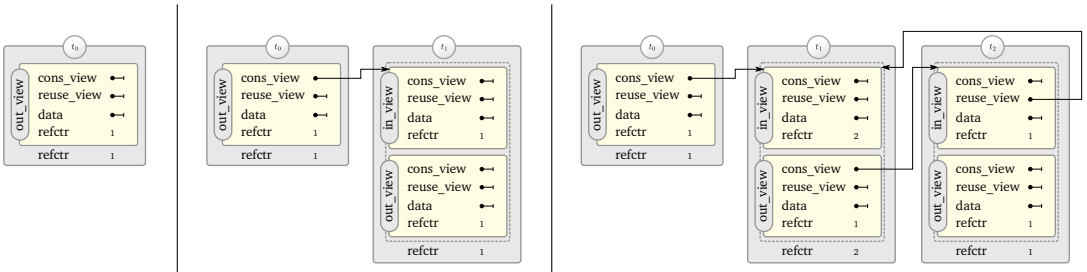
In the following illustration of the steps at execution time, we assume that deferred allocation is enabled. Figure 8.25a shows the creation of the task t_0. As the output view has not been matched with an input view yet, `cons_view` and `data` are initialized with `NULL` and since the task does not have any predecessors, `reuse_view` also receives `NULL`. The reference counters for the view and the frame are set to one.

The creation of t_1 in Figure 8.25b causes the output view of t_0 to be matched with the reuse input view of the inout_reuse view of t_1. No special treatment is required and the `cons_view` pointer receives the address of the reuse input view, just as if the reuse input view were an ordinary input view not associated to an inout_reuse clause. As t_0 does not have any inout_reuse views and as the consumers of t_1 have not yet been created, all pointers of the views of t_1 are initialized with `NULL` and the reference counters are set to one.

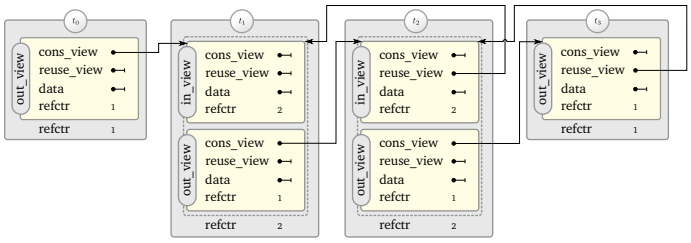
Figure 8.25c shows the matching of the two inout_reuse views of t_1 and t_2 after the creation of t_2. The reuse input view of t_2 references the reuse input view of t_1 by assigning the address of the reuse input view of t_1 to `reuse_view`. As the reuse input view of t_1 is now referenced by t_1 and t_2, the reference counter is increased to the value two. The same applies to the reference counter of the frame of t_1.

The creation of t_3 and the matching of its output view is shown in Figure 8.25d. Similar to t_2, the field `reuse_view` receives the address of the reuse input view of the predecessor t_2 and the reference counters of t_2 are updated accordingly. At the beginning of the execution of t_1 in Figure 8.25e, deferred allocation is triggered and the input buffer of the reuse input view of t_1 is allocated, which results in an update of the data pointers of the output view of t_0 and the reuse input view of t_1. Upon termination of t_0 in Figure 8.25f, the reference counter of its frame is decremented and reaches zero, indicating that the frame is no longer being referenced. Note that the reference counter of the output view is not updated. This is due to the convention that input buffers can only be owned by input views. Reference counters of output views are thus never used and do not need to be updated. The frame associated to t_0 is freed in Figure 8.25g.

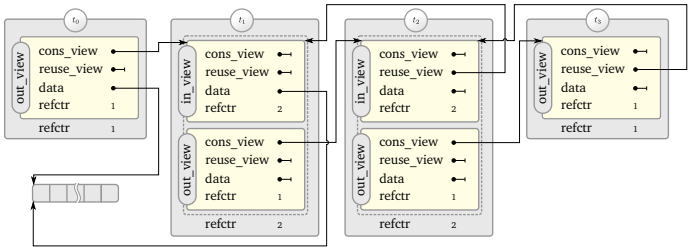
Figure 8.25h and Figure 8.25i show what happens at termination of t_1. In a first step, the `data` pointer of the reuse input view of t_2 receives the address of the input buffer used by t_1 (Figure 8.25h). Afterwards, the `data` pointer of the reuse input view of t_1 is set to `NULL` and its reference counter as well as the reference counter of the frame are decremented (Figure 8.25i). This transfers the ownership for the input buffer from t_1 to t_2



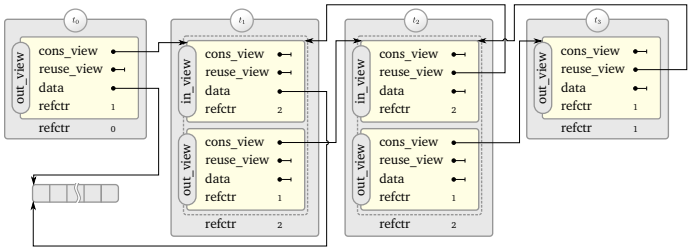
(a) *Creation of* t_0 (b) *Creation and matching of* t_1 (c) *Creation and matching of* t_2



(d) *Creation and matching of* t_3

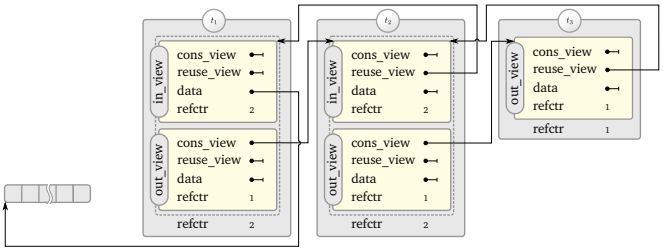


(e) *Start of the execution of* t_0*, allocation of the input buffer*

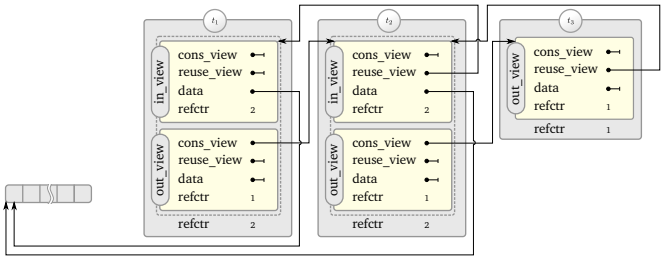


(f) *Termination of* t_0

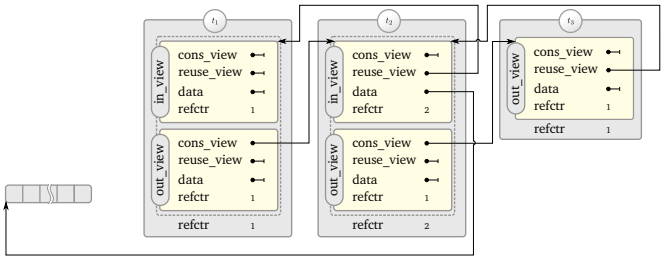
Figure 8.25: *Steps during execution of an application using the inout_reuse clause*



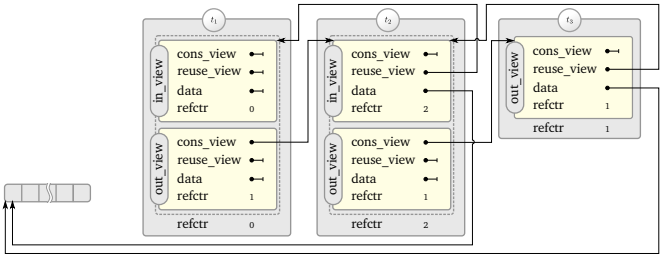
(g) *De-allocation of* t_0



(h) *Termination of* t_1

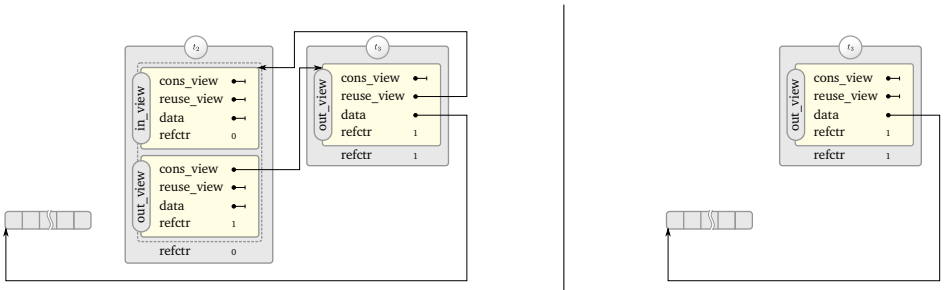


(i) *Start of the execution of* t_2



(j) *Termination of* t_2

Figure 8.25: *Steps during execution of an application using the inout_reuse clause (continued)*



(k) *De-allocation of t_1 and update of reference counters*

(l) *De-allocation of t_2*

Figure 8.25: *Steps during execution of an application using the inout_reuse clause (continued)*

(a) *Transfer from t_i to t_{i+1}*

(b) *Transfer from t_{i+1} to t_{i+2}*

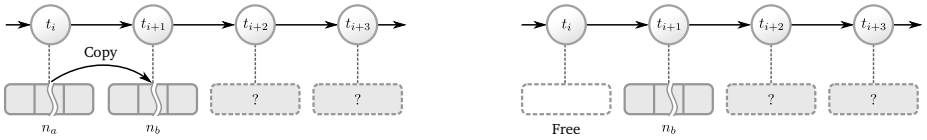
Figure 8.26: *Transfer of ownership resulting in a minimal memory footprint of dependent tasks*

Upon termination of t_2 in Figure 8.25j, the reference counters of t_2 are decremented again and the frame of t_2 becomes ready for de-allocation freed [5]. To pass the contents of the input buffer to t_3 the `data` pointer of t_3 is set to the corresponding address. Figure 8.25k shows the last part of the termination of t_2. Similar to the end of t_1 the data pointer of the reuse input view of t_2 is set to `NULL`, transferring the ownership of the input buffer to t_3. As neither the reuse input view nor the frame of t_2 are used anymore, the respective reference counters are decremented.

In the last step, shown in Figure 8.25l, the frame of t_2 is freed and t_3 becomes ready for execution. Nothing in the data structures of t_3 indicates that the input buffer has been transferred using inout_reuse clauses. When the reference counters of the input view of t_3 is decremented, it reaches zero and as the data pointer has a value different from `NULL`, the input buffer will be freed.

As can be seen in the example, only a single input buffer is allocated for all of the tasks. Hence, the minimal number of input buffers for a chain of dependent tasks is reduced from two buffers for deferred allocation without the inout_reuse clause (as discussed in Section 8.3.4) to a single input buffer. Figure 8.26 illustrates this property on an example with four tasks t_i, t_{i+1}, t_{i+2} and t_{i+3}. In Figure 8.26a, the ownership for the input buffer allocated at the very beginning of the chain is transferred from t_i to t_{i+1}. During its execution, t_{i+1} reads the data written to the input buffer by t_i and overwrites the contents with the results of its own calculation. In the next step, shown in Figure 8.26b, this data is passed to t_{i+2}, again by transferring the ownership of the input buffer. This pattern continues for all remaining tasks on the chain until the destruction of the input buffer at the end of the chain after the execution of a task with an ordinary input view. However, a drawback of this approach is that all tasks reusing a buffer target the same memory controller during their execution. For example, if t_i and t_{i+1} execute on different NUMA nodes, one of them accesses remote memory. However, in contrast to a static placement, the run-time can decide to migrate the data of a buffer from one node to another when the ownership for data is transferred between two views. For example, when t_i finishes, the data pointer of the reuse input view of t_{i+1} could be initialized with the address of a new buffer on an appropriate node. The contents of the buffer used by t_{i+1} would simply need to be copied from the previous buffer to the new buffer and the reference counters of t_{i+1} would need to be updated, such that the old buffer would be

5. Technically this could already happen at the beginning of the execution of t_2. However, in earlier implementations the transfer of ownership was handled differently, which required the de-allocation to be delayed.



(a) *Detection that a local copy is needed* (b) *De-allocation of the source buffer*

Figure 8.27: *Copying the contents of an inout_reuse view when changing nodes*

destroyed properly. This is shown in Figure 8.27, where t_i is executed by a core associated to the node n_a and t_{i+1} is executed by a core associated to n_b. Before t_i is executed, the run-time allocates a new input buffer on n_b, copies the contents of the old input buffer to the new one, sets the data pointer of the reuse input view of t_{i+1} accordingly and frees the old input buffer.

The inout_reuse clause can also be used to pass read-only data or data that is modified infrequently from one task to another. For example, the read-only point data passed between tasks that calculate the distance to cluster centers in the *k-means* benchmark does not have to be copied from an input view to an output view by using the inout_reuse clause.

We have integrated the support for inout_reuse clauses into the OpenStream compiler and runtime as well as different schemes for the allocation of buffers during transfer of ownership between two tasks on different nodes. Furthermore, the inout_reuse clause is an incremental extension and applications using the clause can take advantage of all of the optimizations presented in previous chapters. However, the analysis of the data locality and performance of these solutions is still in an early phase and will be continued in future research following this thesis.

8.6 Summary

In this chapter, we introduced deferred allocation, which delays the allocation and thus the placement of input buffers to the latest possible moment. We showed that this technique mitigates the negative effects on data locality and load balancing of immediate allocation related to the structure of the control program, work-stealing events by remote workers and the creation of initial tasks. As a side effect, deferred allocation can reduce the number of input buffers that are used simultaneously, which results in a reduction of the memory footprint of the application. We showed that deferred allocation can be combined with the *input only* heuristic for work-pushing as well as the mechanism for topology-aware work-stealing presented in the previous chapter.

The experimental evaluation shows that deferred allocation improves the locality of accesses to main memory significantly compared to default random work-stealing with immediate allocation. For a large set of benchmarks, deferred allocation in conjunction with work-pushing and topology-aware work-stealing almost reaches maximum locality. The fraction of local accesses for a benchmark with balanced dependences can be improved to more than 75% of local accesses. The memory footprint is decreased by as much as 47.5% and is only increased for a single benchmark on the Opteron platform. Compared to the default random work-stealing technique and immediate allocation, deferred allocation improves the performance of all benchmarks and yields speedups of up to $3.57\times$ when used in conjunction with the other optimizations. In addition, deferred allocation allows the majority of the dynamic single assignment implementations to outperform the shared memory implementation significantly.

All of the optimizations presented in this chapter and the previous chapter are implemented at the run-time level. They rely on high-level programming information naturally available in task-parallel programs with point-to-point data dependences at execution time and do not require program or input set profiling. Once an application has been implemented using dynamic single assignment on stream elements, the run-time is able to automatically improve its data locality and thus increases performance in many cases, independently from the actual topology of the hardware. Especially on larger systems, dynamic single assignments versions can be significantly faster than shared memory implementations. However, the shared memory implementations in

the comparison use interleaved allocation with excellent load balancing across memory controllers, but with poor average data locality. For an entirely fair comparison, it would be necessary to use shared memory versions with explicit, manual data placement adapted to the target platform. Although these versions might yield higher performance, they require substantial engineering efforts and become dependent on the target machine.

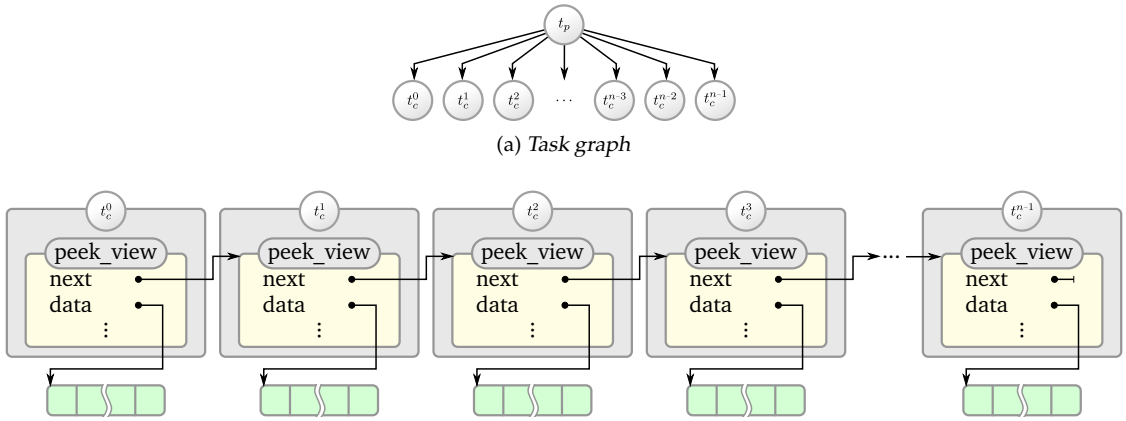
The improvements achieved through data-aware scheduling and deferred allocation of the previous chapters focused on the data locality of applications with point-to-point dependences, where data produced by a task is read by a single consumer. In the next chapter, we focus on the performance of broadcasts, where the output data of a single producer needs to be transferred to multiple consumers.

9 Optimizing broadcasts

The previous chapters focused on point-to-point communication, where elements of a stream are written and read by exactly one task. We showed that it is possible to improve the placement of tasks and data dynamically at execution time to increase the locality of memory accesses by exploiting these dependences, leading to higher performance of memory-intensive applications. In this chapter, we focus on applications with broadcasts, in which the results of a producer are read by more than a single reader. We show that deferred allocation, presented in the previous chapter, improves performance of such applications, but that the benefits are limited due to the high memory footprint caused by the broadcast mechanism presented in Section 3.3. We then introduce broadcast tables, an optimization for broadcasts that reduces the memory footprint and increases performance. The experimental evaluation is conducted on the *cholesky* benchmark, which extensively uses broadcasts throughout its entire execution. We show that using broadcast tables, OpenStream is able to match the performance of state-of-the-art implementations of Cholesky Factorization for many-core systems. Data locality and load balancing across memory controllers are addressed at the end of the chapter, generalizing the concept of broadcast tables for future research.

9.1 Memory footprint and execution time of broadcasts

To motivate the performance issues related to broadcasts, recall the principles of broadcasts in the execution model of OpenStream. In this model, data that is read by multiple readers through peek views is broadcast by the producer of the data at the end of its execution by copying the data to each of the readers' input buffers. The order of the copies depends on the order of the matching of the readers' peek views on the stream used for the broadcast. Figure 9.1a illustrates these concepts on a broadcast with one producer t_p and n readers t_c^0 to t_c^{n-1}. Assume that the order of task creation of the readers corresponds to their indexes, i.e., the first reader that is created is t_c^0, followed by t_c^1 and so on. According to the matching mechanism described in Section 3.4.3, all readers are added to a list of siblings reading the same stream elements upon their respective calls to `resolve_dependences`, which immediately follows task creation. The tick operation on the stream that is used for the broadcast must take place after the creation of all readers and after the creation of the producer. This causes the producer t_p to be matched with the first reader t_c^0 in the list of siblings. The synchronization counter of the producer is decreased and the producer is activated, unless it has other unmet dependences. During the execution of t_p, the task writes its output data to the input buffer of t_c^0, just as if there was a simple point-to-point dependence



(a) *Task graph*

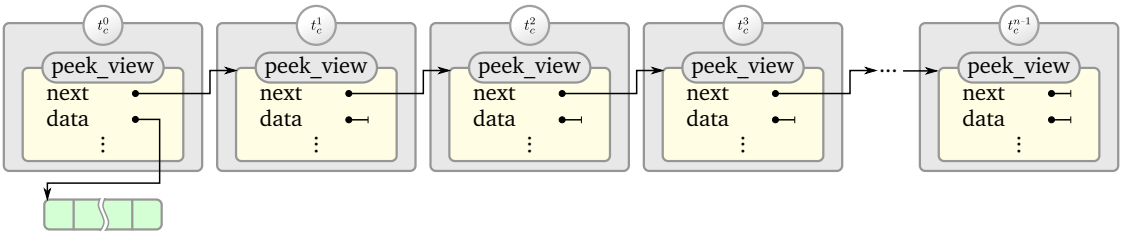
(b) *Multiple copies at the end of the broadcast*

Figure 9.1: *Broadcast to n readers with multiple copies*

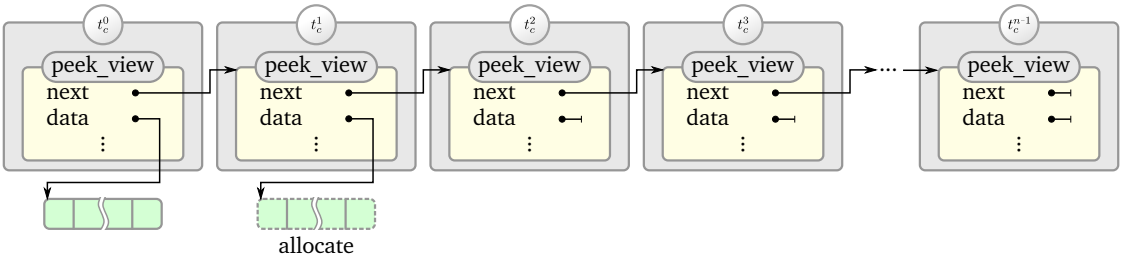
between t_p and t_c^0. The actual broadcast takes place at the end of the execution of t_p and consists in copying the input data of t_c^0 to the input buffers of the remaining tasks t_c^1 to t_c^{n-1}. In the worst case with respect to the memory footprint, there are n copies stored in n buffers at completion of the broadcast, as shown in Figure 9.1b.

Deferred allocation, presented in the previous chapter, can help to reduce the footprint, but does not guarantee a lower memory footprint under all circumstances. Figure 9.2 illustrates the events leading to the minimal footprint for a broadcast. The situation at the beginning of the broadcast is given in Figure 9.2a, where the output data of the producer is only available in the input buffer of the first reader t_c^0. Due to the use of deferred allocation, neither of the input buffers of the remaining readers t_c^1 to t_c^{n-1} has been allocated at task creation. Each of these buffers will be allocated during the traversal of the chain of readers, when the data of the first reader needs to be copied to the private input buffer of the corresponding reader. The first allocation is shown in Figure 9.2b, where the data pointer of the peek view of t_c^1 is initialized with the address of the newly allocated buffer. In the next step, shown in Figure 9.2c, the data of the broadcast is copied from the input buffer of t_c^0 to the input buffer of t_c^1. Note that after this operation, t_c^1 is ready for execution as all of its input data is available and unlike t_c^0 no other task depends on the existence of its buffer. However, due to the use of the single entry software cache, presented in Chapter 3.4.1, the task is protected from steals and remains only accessible to the worker that activated the task, which is the worker executing the producer of the broadcast. Figure 9.2d and 9.2e show the allocation and the initialization of the input buffer of the next task t_c^2. The activation of t_c^2 after these events causes t_c^1 to be transferred from the single entry software cache to the work-queue of the worker executing the producer and thus exposes t_c^1 to steals. For a minimal memory footprint, this task must be stolen and be executed by another worker and its input buffer must be freed before the producer reaches t_c^3 in the chain of readers. Only in this case the buffer of t_c^1 can be reused for t_c^3, as shown in Figure 9.2i. A minimal footprint also requires that this pattern of copying, stealing, freeing and reusing continues for the remaining readers until the end of the broadcast.

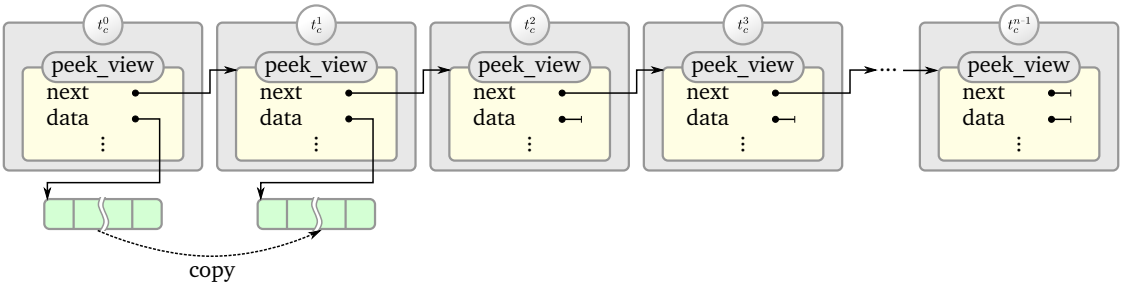
As shown above, in the best case for the memory footprint only three input buffers are needed: the input buffer of t_c^0 from which data is copied and the input buffer of two of the remaining readers (the first of the remaining readers is the one in the single entry software cache and the second one is the reader whose input buffer is being written). However, this requires that each reader t_c^i terminates before the input buffer of t_c^{i+2} is allocated, which in turn requires that another worker steals and executes t_c^i in between. It is highly unlikely that this timing of events occurs during execution and it is more likely that the input buffers of at least a few of the readers are allocated before the first reuse of an input buffer can take place. How many input buffers co-exist also depends on the amount of parallelism of the application and load balancing across workers.



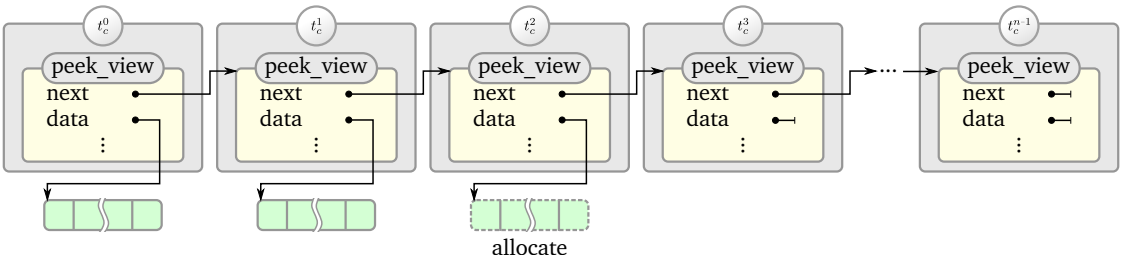
(a) *After the execution of the task body of* t_p



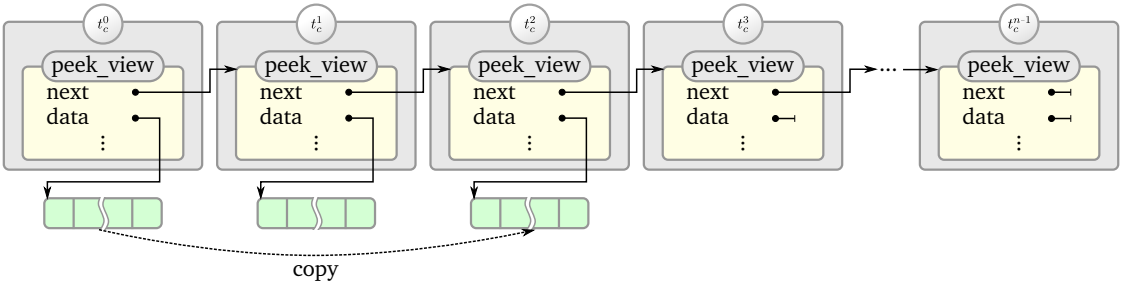
(b) *Allocation of the input buffer of* t_c^1



(c) *Copy by* t_p *to the input buffer of* t_c^1

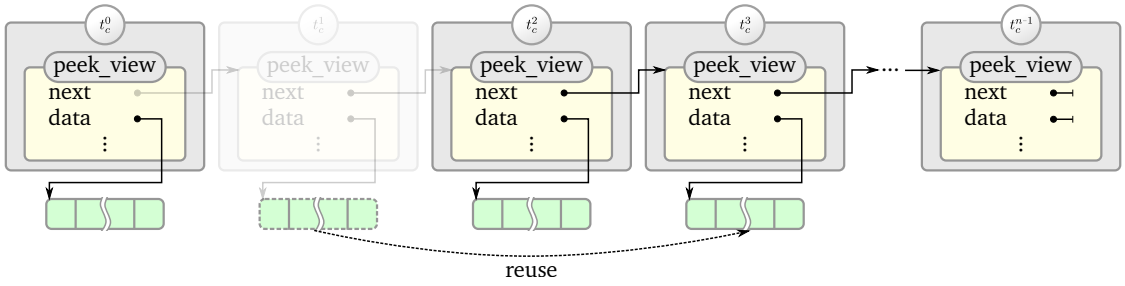


(d) *Allocation of the input buffer of* t_c^2



(e) *Copy by* t_p *to the input buffer of* t_c^2

Figure 9.2: *Broadcast with deferred allocation*



(i) *Allocation of the input buffer of t_c^3, reusing the buffer of t_c^2*

Figure 9.2: *Broadcast with deferred allocation (continued)*

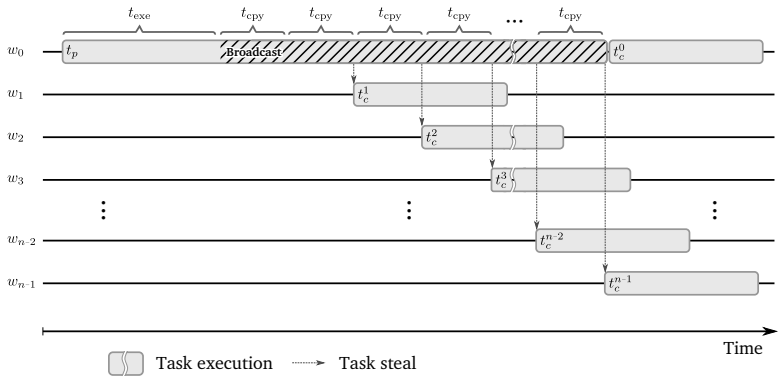


Figure 9.3: *Timing related to copies during a broadcast*

If many workers are idle during the broadcast, readers that are ready for execution are likely to be stolen quickly and the time to the first reuse of a buffer is low. On the contrary, if all workers are busy executing other tasks, the worker executing the producer of the broadcast accumulates readers that are ready for execution in its work queue. As these tasks are not executed before the worker finishes the broadcast, their resources cannot be freed and the number of co-existing buffers is high. As we will show in the experimental evaluation of this section, the footprint using deferred allocation ranges between the minimal footprint and the footprint of the default implementation for broadcasts that allocates all buffers of the readers at their creation.

Another issue arising from the default mechanism for broadcasts is related to the timing of the copies shown in Figure 9.3. Let t_{exe} be the time that is necessary for the producer to carry out all instructions of its task body, including the writes to the input buffer of the first reader, and let t_{cpy} be the time that is needed to copy the contents of the input buffer of t_c^0 to the input buffer of one of the remaining readers [1]. As the input buffer of the first reader acts as the source for all copy operations, t_c^0 cannot become ready until all copies are done. Hence, t_c^0 becomes ready only upon termination of t_p. The other readers t_c^i are unblocked earliest at $t_{\text{exe}} + \sum_{j=1}^{i} t_{\text{cpy}} = t_{\text{exe}} + i \cdot t_{\text{cpy}}$ after the start of t_p. Note that the second reader t_c^1 becomes ready after the first copy, but cannot be stolen by other workers as it is first transferred to the single entry software cache. Therefore, the arrow indicating the steal of t_c^1 in Figure 9.3 is located at $2 \cdot t_{\text{cpy}}$ after the beginning of the broadcast. The average waiting time $t_{w,\text{avg}}$ of a reader to become ready for execution can be calculated as

1. In reality this time is not constant and depends on many factors related to the hardware topology (e.g., the distance between the writer and the targeted buffer) and on dynamic effects, such as the contention on the memory controllers involved in the transfer. However, for simplicity we assume that t_{cpy} is constant for all copies of the broadcast.

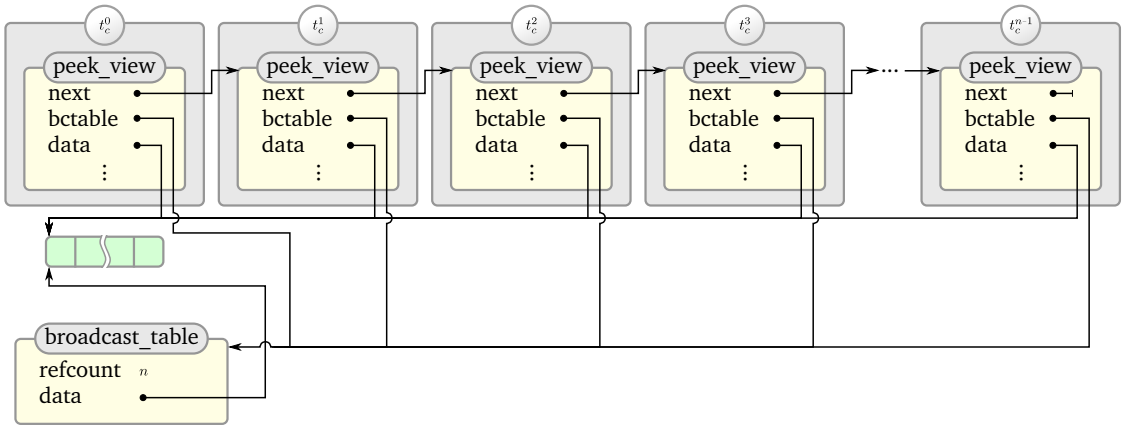


Figure 9.4: *Sharing of a single input buffer in a broadcast using a broadcast table*

follows:

$$
\begin{aligned}
t_{w,\text{avg}} &= \frac{1}{n}\left(\underbrace{(n-1)\cdot t_{\text{cpy}}}_{\text{First reader}} + \underbrace{\sum_{i=1}^{n-1} i \cdot t_{\text{cpy}}}_{\text{Remaining readers}}\right) \\
&= \frac{t_{\text{cpy}}}{n}\left((n-1) + \sum_{i=1}^{n-1} i\right) \\
&= \frac{t_{\text{cpy}}}{n}\left((n-1) + \frac{n\cdot(n-1)}{2}\right) \\
&= \frac{t_{\text{cpy}}}{n}\cdot\frac{(n+2)(n-1)}{2} \\
&= \frac{t_{\text{cpy}}}{n}\cdot\frac{n^2+n-2}{2} \\
&= \frac{1}{2}t_{\text{cpy}}\cdot\left(n+1-\frac{2}{n}\right)
\end{aligned}
$$

Hence, the average waiting time grows linearly with the number of readers involved in a broadcast and the more readers are involved in the broadcast the longer it takes for each task on average to become ready for execution. In the next section, we will show how the memory footprint as well as the average waiting time can be reduced considerably.

9.2 Reducing the memory footprint and execution time

As peek views are a special form of input views with a burst of zero, they do not allow the data that is available through the view to be modified by the task owning the view. Hence, all values that are broadcast from a producer to its readers are guaranteed to remain constant after termination of the producer. An individual copy of the data for each reader is thus not required to preserve the semantics of an OpenStream program. In addition, sharing a single buffer among all readers does not only reduce the memory footprint considerably, but also decreases the average time from the termination of the producer to the activation of a reader by orders of magnitude, as shown below. However, as described in Section 4.3, an input buffer is used only by a single task and freed upon its termination. Sharing a buffer among multiple tasks thus requires a protection mechanism that prevents the buffer from being freed until all of the readers of the broadcast have terminated.

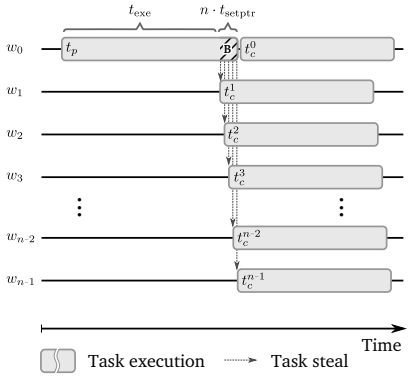


Figure 9.5: *Timing of a broadcast when using a broadcast table*

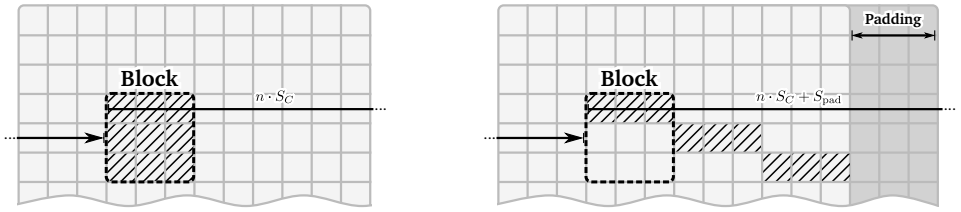
Figure 9.4 shows the data structures involved in a broadcast with a single input buffer shared by all readers. The central data structure is called `broadcast table` and allows the run-time to keep track of references to the shared input buffer. The broadcast table is allocated at the beginning of the broadcast upon termination of the producer and its data pointer is initialized with the address of the input buffer of the first reader t_c^0. In addition to the data pointer, each input view is also provided with a pointer to the broadcast table. During the broadcast, these pointers are set to the address of the broadcast table, which allows each reader to locate the shared input buffer and to update its data pointer when it becomes ready for execution. After the broadcast and when all readers are ready for execution, all data pointers of the readers' input views point to a unique input buffer, as illustrated by the figure [2]. The reference counter `refcount` holds the number of tasks currently using the buffer. At the beginning of the broadcast, this counter is initialized with the number of readers and is successively decreased with each reader that terminates afterwards. Similar to the synchronization counter of tasks, it is updated using an atomic decrement each time a reader finishes execution. When the counter reaches zero, the input buffer referenced by the broadcast table as well as the broadcast table itself are freed. To be able to carry out the decrement on the correct broadcast table, the address of the broadcast table must be known when a task terminates. Therefore, each of the peek views is provided with an additional pointer named `bctable`, which points to the broadcast table used by the view.

Obviously, the memory footprint using broadcast tables does not grow linearly anymore as in the default implementation, since the amount of memory per broadcast is constant for any number of readers. The number of buffers that can be saved compared to the default broadcast mechanism depends on the timing of allocations, executions and reuse of buffers with this mechanism, as shown in the analysis for the minimal number of buffers in the last section. The value ranges from a reduction by two buffers if the default broadcast mechanism yielded the minimum size of buffers, to $n - 1$ in the worst case scenario for the default broadcast mechanism.

As far as the average waiting time is concerned, it still depends on the number of readers, but is orders of magnitude smaller than before. Figure 9.5 illustrates the timing of a broadcast based on a broadcast table. At the end of the execution of the producer, the pointer to the broadcast table of all readers must be initialized. If the time that is required to initialize a pointer is t_{setptr}, the entire broadcast can thus be carried out in $n \cdot t_{\text{setptr}}$. Depending on the size of the data to be broadcast, setting a pointer is usually orders of magnitude faster than copying the data, such that the entire broadcast is orders of magnitude faster than without broadcast tables.

Reducing the waiting time has also a positive effect on the critical path of an applicaiton. In the worst case without broadcast tables, a task on the critical path is the last reader in the chain of siblings and becomes ready after $(n-1) \cdot t_{\text{cpy}}$. By using broadcast tables, this time is reduced to $n \cdot t_{\text{setptr}}$ and the task on the critical path becomes ready almost instantaneously. In summary,

2. As a reader t_c^i might execute before another reader t_c^j with $j > i$ becomes ready, it is possible that there are less than n references to the broadcast table at the end of the broadcast.



(a) *Without padding* (b) *With padding*

Figure 9.6: *Cholesky: improved layout of data in shared memory*

broadcast tables increase performance by:

- reducing the time spent on copying data from the first reader of the broadcast to the individual input buffers of the remaining reader. This reduces the average and worst-case waiting time, unblocking tasks including those on the critical path almost instantaneously.
- reducing the number of input buffers that is necessary to store the input data of all reader, resulting in a lower memory footprint and less time spent on logical and physical allocation of buffers.

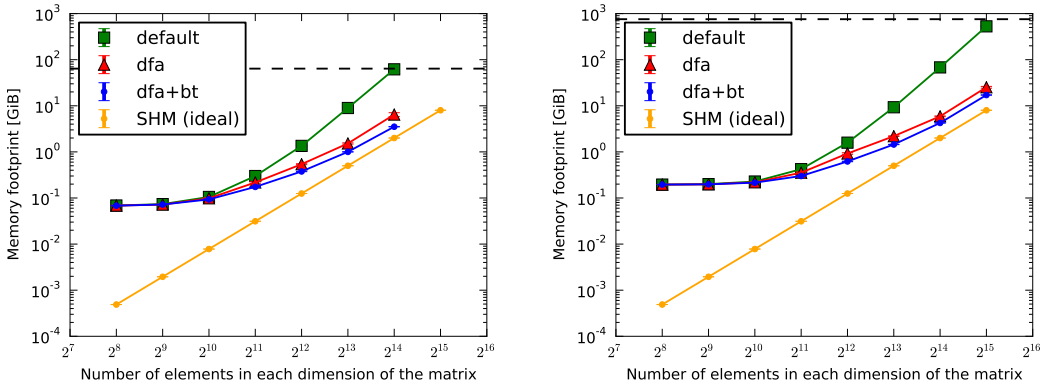
In the next section, we compare the performance and the memory footprint of the default broadcast mechanism with broadcast tables on the *cholesky* benchmark, a broadcast-intensive linear algebra kernel carrying out Cholesky Factorization.

9.3 Experimental evaluation

To evaluate the impact of broadcast tables on the memory footprint and on performance, we measured the execution time and the footprint of the *cholesky* benchmark, described in Section 6.1.5. One of the key characteristics of this benchmark is that it makes extensive use of broadcasts to a large number of readers. In a second evaluation, we compare the OpenStream implementation to two state-of-the-art implementations of the Cholesky Factorization, PLASMA [60] and an OpenMP implementation provided by Intel's MATH KERNEL LIBRARY [2]. Before we discuss the experimental results, we first describe the changes that we have applied to the layout of the matrix to be factorized in main memory, in order to improve the fraction of requests to caches that result in cache hits.

9.3.1 Changes of the data layout improving cache hit rates

The elements of a two-dimensional $N \times N$ matrix in shared memory are stored linearly at consecutive addresses ranging from a base address a_0 to a_{N^2-1} with $a_{i+1} = a_i + S_e$, where S_e is the size in bytes of a single element of the matrix (e.g., 8 bytes for a double precision floating point value). To improve the cache hit rate, the matrix is processed in blocks of $S_B \times S_B$ elements that can be held in the cache simultaneously. The elements of each row of such a block are stored at consecutive addresses, but the distance between two consecutive rows is greater than zero if the matrix is composed of more than one block, i.e., if $S_B \neq N$. Figure 9.6a illustrates the distance between the first element of the first row of a block to the first element of the second row. If this distance is a multiple of the cache size S_C, the elements of each row are mapped to the same index in the cache, as indicated by the striped pattern in the figure. Depending on the associativity of the cache, this effectively limits the space that is available for the elements of a block and can result in a high miss rate. For example, if the associativity is four, then only four rows of a block can be held in the cache simultaneously, although the capacity of the cache might be orders of magnitude higher. As the elements of a block are read multiple times, this leads to an unnecessarily high number of cache misses.



(a) *Opteron platform* (b) *SGI system*

Figure 9.7: *Memory footprint of* cholesky *with and without broadcast tables. The configurations are the default broadcast without deferred allocation (default), with deferred allocation (dfa) and with deferred allocation in conjunction with broadcast tables (dfa+bt).*

A simple way to deal with this situation is to add padding elements to each row of the matrix, which increases the distance between the rows of a block. This is shown in Figure 9.6b, in which each row of the matrix is padded with S_{pad} bytes, resulting in a distance of $n \cdot S_C + S_{\text{pad}}$ between two rows of a matrix. Ideally, S_{pad} is equal to the size of a row of the block, such that two consecutive rows of a block are mapped to consecutive indexes of the cache. In all experiments discussed below, we used a padding of 2048 bytes, which corresponds to the size of a row of a block of $S_B = 256$ double precision floating point elements, independently from the size of the matrix, as defined in Section 6.4.

As the padding is beneficial to all operations on matrices in shared memory, we have applied the same padding to the input matrix of the *cholesky* benchmark using dynamic single assignment. Most of the time, this benchmark operates on blocks stored in streams, representing contiguous regions of memory that do not induce the problems described above. However, initial data is read from the input matrix in shared memory, such that the padding is beneficial to tasks processing input data and writing the results to streams.

9.3.2 Impact on the memory footprint and performance

To evaluate the impact of deferred allocation and broadcasts tables on the memory footprint and on the execution time, we have executed the *cholesky* benchmark using three configurations of the run-time. The first configuration, named *default*, uses neither deferred allocation nor broadcast tables, but applies work-pushing using the *weighted* heuristic and topology-aware work-stealing, as described in Section 7.2 and 7.3. The second configuration, labeled *dfa*, applies work-pushing with the *input only* heuristic, topology-aware work-stealing and deferred allocation. The last configuration, labeled *dfa+bt*, finally applies all optimizations at the same time, i.e., work-pushing using the *input only* heuristic, deferred allocation, topology-aware work-stealing and broadcast tables.

Figure 9.7 shows the memory footprint in GiB of the benchmark executing on the Opteron system and on the SGI platform, presented in Section 6.3, using the three configurations above for a matrix size ranging from $2^8 \times 2^8$ to $2^{14} \times 2^{14}$ and $2^{15} \times 2^{15}$ double precision floating point elements, respectively. Each point on the curves represents the median value for a total of 50 runs using all 64 cores on the Opteron platform and all of the 192 cores of the SGI system. Error bars indicate the standard deviation. The lower bound for the memory footprint of an algorithm that performs Cholesky Factorization is achieved if a single global matrix used in shared memory with in-place updates, in which case the memory footprint is equal to the size of the matrix. To illustrate the overhead of the above configurations compared to the lower bound, the figure also

plots the size of the matrix, indicated by the line labeled *SHM (ideal)*. The dashed lines on the graphs indicate the total size of the main memory for each system. To highlight the differences for both small and large matrices, the axes of the graph use a logarithmic scale. The reason for the smaller maximum size of the matrix on the Opteron system is that the unoptimized version swaps out memory pages to the hard disk for matrices bigger than $2^{14} \times 2^{14}$. Swapping may also occur for matrices of $2^{14} \times 2^{14}$ on this system, but only affects a small subset of the data.

For small matrices of up to $2^{10} \times 2^{10}$ elements, the three configurations yield approximately the same memory footprint. This is primarily due to the fact that the number of blocks is small, such that the number of readers per broadcast is also small. Figure 9.8 illustrates this property by plotting the number of broadcasts and the total number of peek views of the entire application. As can be seen in the graph, both the number of broadcasts and the number of peek views or readers increase with the size of the matrix. As the number of tasks that rely on a block increases with the number of blocks of the matrix, the gap between the two curves is small for smaller matrices and widens for bigger matrices. In other words, for larger matrices there are more broadcasts and each broadcast has more readers. In addition, Figure 9.7 shows that the difference between the actual footprint of the application and the ideal footprint decreases for larger matrices. The reason for this behavior is that the footprints in the graph represent the overall footprints of the application, including the input matrix in shared memory, all auxiliary data structures, such as data structures that represent workers, streams, frames containing only metadata as well as the stack segments for each worker. As most of this overhead is constant, the relative overhead thus becomes smaller for bigger matrices.

On both systems, deferred allocation and broadcast tables reduce the memory footprint of large matrices significantly. For the largest matrix on the Opteron system, these optimizations reduce the memory footprint by a factor of approximately 15, from more than 62 GiB to approximately 6 GiB (*dfa*) and less than 4.2 GiB (*dfa+bt*). For the largest matrix on the SGI platform, the reduction is even higher: the initial footprint for the default broadcast algorithm without deferred allocation of more than 520 GiB is reduced to less than 26 GiB (*dfa*) and about 17 GiB (dfa+bt), which corresponds to a factor of 20 and more than 30. Hence, both the use of deferred allocation and broadcast tables result in huge improvements on the memory footprint of the application. The largest difference represents more than one order of magnitude. Gains for bigger matrices are expected to be even higher.

Figure 9.9 shows the total number of allocations of 512 KiB-blocks, corresponding to the size of a block of the matrix, from memory pools during the execution of the benchmark. The number of allocations for the default broadcast mechanism and the one for deferred allocation are identical. This is due to the fact that deferred allocation favors the reuse of buffers, but still allocates one buffer for each peek view. In contrast to this, broadcast tables reduce the number of allocations significantly. Figure 9.10 takes into account the reuse of buffers and presents the total number of refill operations for 512 KiB blocks. While the default mechanism yields the least reuse, both deferred allocation and broadcast tables reduce the number of refills significantly.

A similar conclusion as for the memory footprint can be drawn from Figure 9.11, showing the wall clock execution time for the three configurations, i.e., the default broadcast mechanism, deferred allocation and deferred allocation in conjunction with broadcast tables. However, the gap between the configurations appears only for matrices with $2^{14} \times 2^{14}$ elements and more. For a $2^{15} \times 2^{15}$ matrix processed on the SGI platform, execution with the default configuration terminates after about 80 s. Deferred allocation reduces the execution time to about 10 s and the configuration using broadcast tables finishes in less than 7 s on average. For the largest matrix on the Opteron system, the execution time for the default broadcast mechanism varies between about 200 s and more than 500 s. This time can be reduced to less than 6 s by deferred allocation and to less than 5 s by using broadcast tables.

9.3.3 Comparison with state-of-the-art implementations of Cholesky Factorization

The results above show that deferred allocation and broadcast tables can improve both the memory footprint as well as the execution time significantly. In this section, we show that the

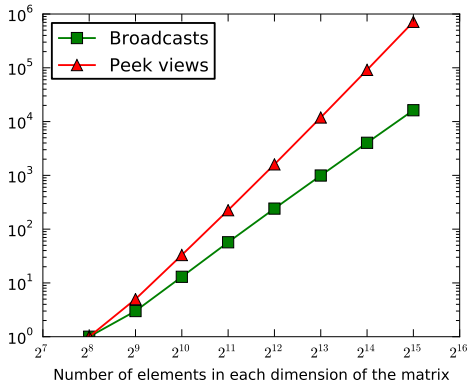
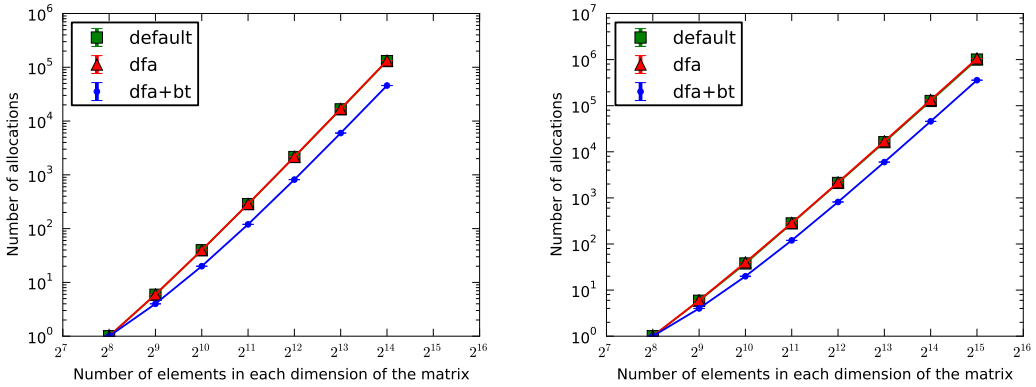


Figure 9.8: *The number of broadcasts and readers in* cholesky *as a function of the size of the matrix*



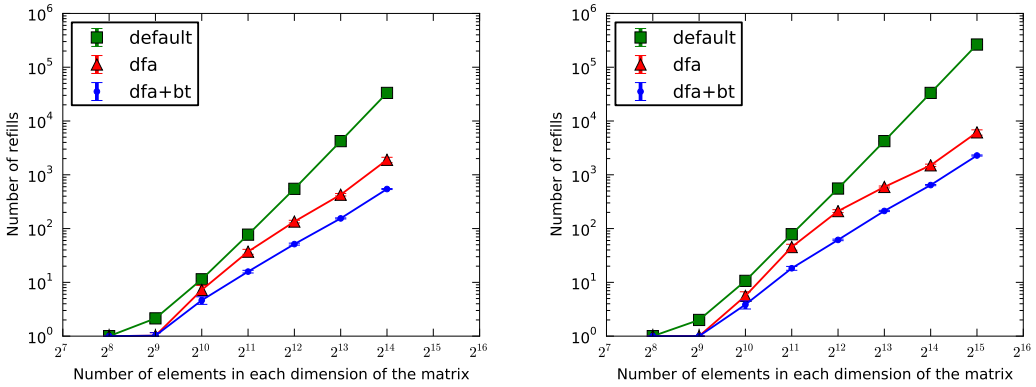
(a) *Opteron platform* (b) *SGI system*

Figure 9.9: *Number of allocations of* 512 KiB*-blocks from memory pools during execution of* cholesky*. The configurations are the same as in Figure 9.7.*

resulting performance is comparable to state-of-the-art implementations of Cholesky Factorization for many-core systems. To this end, we compare our implementation based on OpenStream to parallel Cholesky Factorization provided by PLASMA [60] and an OpenMP implementation by Intel's MATH KERNEL LIBRARY [2], presented below.

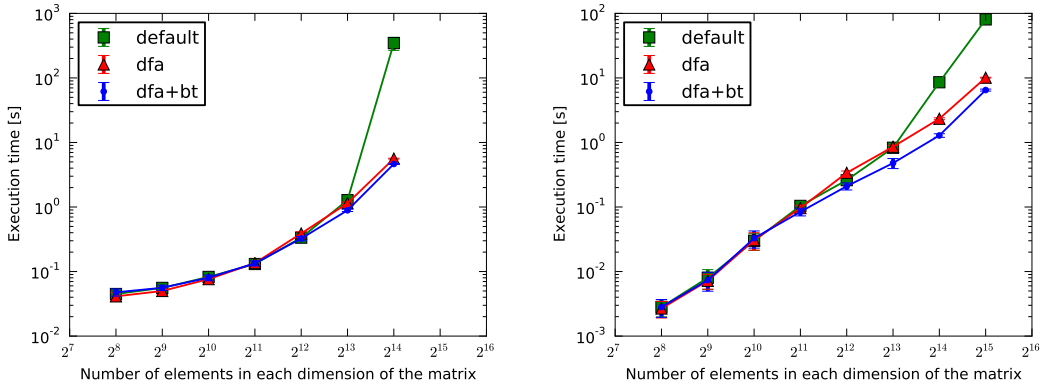
PLASMA and QUARK

PLASMA [60] (PARALLEL LINEAR ALGEBRA SOFTWARE FOR MULTI-CORE ARCHITECTURES) is a library providing a set of ready-to-use functions for dense linear algebra operations optimized for the execution on multi-socket multi-core systems. The aim of PLASMA is to provide efficient implementations of LAPACK [12] routines for modern parallel hardware architectures that can be used as a drop-in replacement of existing implementations. Internally, PLASMA breaks operations down into multiple computations with data dependences and passes the resulting DAG of dependent tasks to the QUARK [85] scheduler. As a computation may result in a large number of task executions, PLASMA creates tasks on-the-fly and passes them to QUARK as soon as possible. Each QUARK task is defined by a pointer to a function with the code to be executed by the task, a set of parameters and a specification of the modes in which parameters are accessed (read, write or both) as well as the size of the data. The order of task execution is derived automatically from the parameters of the task by the QUARK scheduler. In addition to data dependences, the scheduler also takes into account information about the topology of the machine based on information obtained from the HWLOC [30] library. High sequential performance of each task is achieved through calls to an optimized BLAS library. For our experiments, we have



(a) *Opteron platform* (b) *SGI system*

Figure 9.10: *Number of refills during execution of* cholesky *for blocks of* 512 KiB. *The configurations are the same as in Figure 9.7.*



(a) *Opteron platform* (b) *SGI system*

Figure 9.11: *Execution time of* cholesky *with and without broadcast tables. The configurations are the same as in Figure 9.7.*

configured the latest available version of PLASMA (2.6.0) to use the sequential kernels of the MKL for each task.

Parallel Cholesky Factorization using the MKL

Besides the sequential implementations of linear algebra routines, the INTEL MATH KERNEL LIBRARY [2] also offers parallel implementations of BLAS [20] and LAPACK [12] operations, including the `dpotrf` function for Cholesky Factorization of double precision floating point matrices. These implementations are based on OpenMP and automatically adapt the tile size to the target architecture [8]. For our experiments, we used the parallel implementation of the latest release of the MKL, which was the INTEL MATH KERNEL LIBRARY 11.1 UPDATE 3 FOR LINUX at the time the experiments were run.

Results

Figure 9.12 shows the execution time in seconds for the OpenStream implementation, the parallel implementation by the MKL and for PLASMA, operating on matrices whose size ranges from $2^8 \times 2^8$ to $2^{15} \times 2^{15}$ double precision floating point elements. As in the previous graphs, each point represents the median value for 50 runs using all 64 cores on the Opteron platform and all of the 192 cores of the SGI system and error bars indicate the standard deviation. On the Opteron platform, the parallel implementations by the MKL and OpenStream perform equally well, while PLASMA is slightly slower for matrices with more than $2^{13} \times 2^{13}$ elements. The fastest implementations on the SGI system are OpenStream and PLASMA with the same execution time. In contrast to the Opteron system, the parallel implementation by the MKL is by far the slowest implementation on the SGI system. In addition, the execution time of the parallel version of the MKL routine has huge variations for large matrices. As the source code of the MKL is not publicly available, we could not investigate the cause for the low performance and the variations, but we suspect that this is related to the synchronization scheme used by the OpenMP code of the MKL.

The absolute performance in GFLOPS is shown in Figure 9.13[3]. As the absolute performance is proportional to the inverse of the execution time shown above in Figure 9.12, OpenStream and the MKL reach higher values than PLASMA on the Opteron system for larger matrices. Similarly, OpenStream and PLASMA outperform the MKL on the SGI system. For a fixed block size for all experiments as in the graphs, parallelism in *cholesky* increases with the size of the matrix. Hence, for small matrices parallelism is limited and might even be below the number of cores of the system and for large matrices all cores can effectively contribute to the factorization of the matrix. As a result, the graphs show that the absolute performance increases with the size of the matrix.

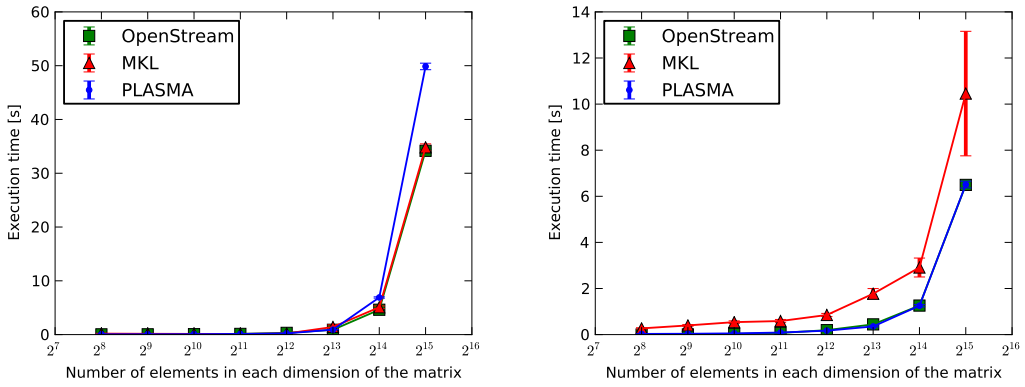
The highest median performance of about 1.8 TFLOPS is achieved by both PLASMA and OpenStream for matrices with $2^{15} \times 2^{15}$ elements on the SGI system, representing about 50% of the theoretical peak performance of the machine of $24 \cdot 153.6$ GFLOPS ≈ 3.67 TFLOPS [3]. Given the fact that the peak performance can only be achieved for embarrassingly parallel algorithms operating on the register bank, the exploitation of the machine for Cholesky Factorization can be considered as high. However, the performance increase between matrices of size $2^{14} \times 2^{14}$ and matrices of $2^{15} \times 2^{15}$ elements gives reason that performance has not leveled out. We could not evaluate the performance of all implementations with bigger matrices due to a technical restriction of PLASMA to matrices with less than 2^{32} elements.

The memory footprint of the three implementations is illustrated in Figure 9.14. For large matrices, OpenStream and the MKL require the least amount of memory on both systems. The memory footprints of PLASMA and the MKL are smaller than the footprint of OpenStream only on the SGI system for small matrices.

9.3.4 Conclusion

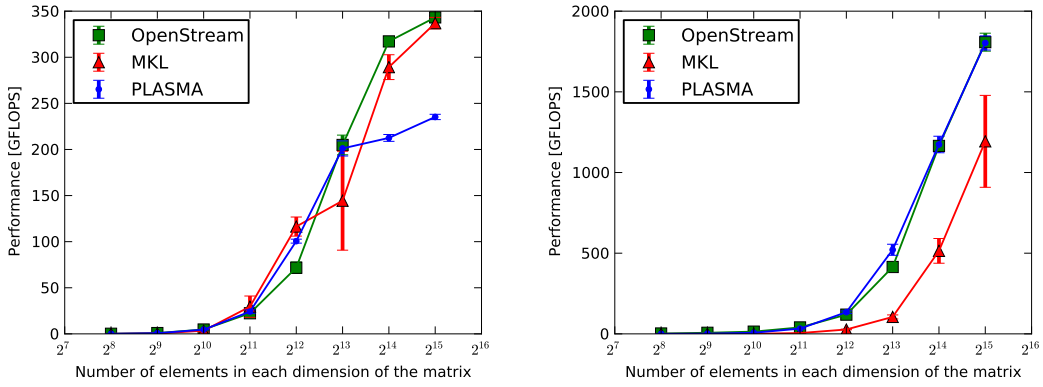
The comparison of different versions of the OpenStream run-time using the default immediate allocation scheme, using deferred allocation and using both deferred allocation and broadcast ta-

3. These values are based on the number of floating point operations for a Cholesky Factorization of a matrix of $2^N \times 2^N$ of $\frac{1}{3}2^{3N} + \frac{1}{2}2^{2N} + \frac{1}{6}2^N$ as reported in [21].



(a) *Opteron platform* (b) *SGI system*

Figure 9.12: *Execution time of* cholesky *compared to state-of-the-art implementations for many-core systems*



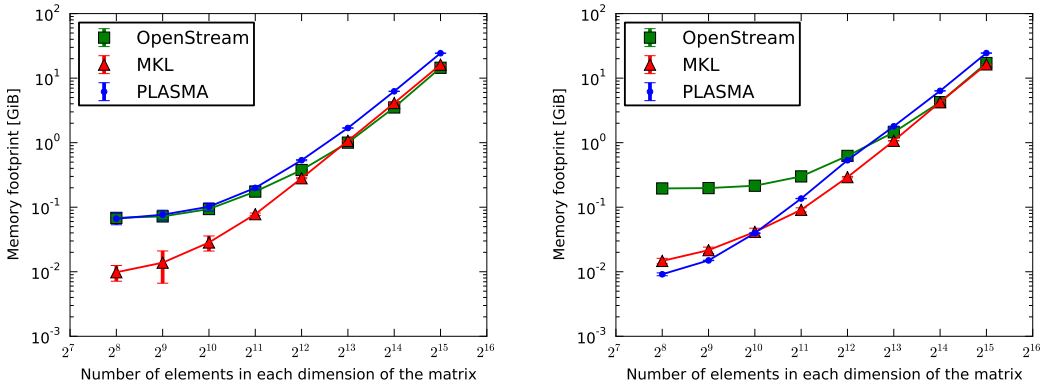
(a) *Opteron platform* (b) *SGI system*

Figure 9.13: *Performance of* cholesky *compared to state-of-the-art implementations for many-core systems*

bles shows that broadcast tables yield the best performance and the smallest memory footprint for *cholesky*, a benchmark that uses the broadcast mechanism of OpenStream extensively. The results of the comparison with PLASMA and the MKL show that the dynamic single assignment implementation in OpenStream using broadcast tables is able to match the performance of PLASMA operating on a shared memory matrix with interleaved allocation for load balancing across the machine's memory controllers. All benchmarks have been tuned for the target architecture with respect to the block size and conflict misses in the cache using appropriate values for the padding of the shared memory matrices. Hence, the overhead on the memory footprint due to the use of dynamic single assignment as well as the overhead on execution time of the OpenStream run-time can be considered as sufficiently small and OpenStream can be considered as a state-of-the-art run-time system that enables the implementation of high performance applications for many-core systems.

9.4 NUMA-aware broadcasts with on-demand copies

As shown in the previous section, broadcast tables can reduce the memory footprint of an application with frequent broadcasts significantly. However, in contrast to the default scheme for broadcasts, in which every reader has its own input buffer, all readers of a broadcast read from



(a) *Opteron platform* (b) *SGI system*

Figure 9.14: *Footprint of* cholesky *compared to state-of-the-art implementations for many-core systems*

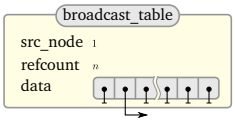


Figure 9.15: *Broadcast table with support for multiple copies*

the same input buffer. As this buffer is located on a single NUMA node, this potentially creates contention on a single memory controller and may lead to poor data locality during execution of the readers.

Deferred allocation for broadcasts suffers from a similar problem for data locality. Compared to the default mechanism, the memory footprint is reduced due to the reuse of buffers, but it is not minimal. Each reader still reads from a private buffer and due to the fact that all buffers of a broadcast are allocated by the worker executing the producer of the broadcast, all buffers are placed on the same node. As with broadcast tables, this potentially leads to high contention and poor data locality. In addition, multiple buffers with the same data might co-exist, which represents a redundancy that might result in a waste of cache space.

In this section, we address the locality of memory accesses and contention of broadcasts by generalizing the concept of broadcast tables. Instead of referencing a single input buffer located on a single node, the generalized form of broadcast tables, presented below, is able to provide copies of the data on multiple nodes, which allows readers executed by workers associated to the same NUMA node to benefit from locally available data.

9.4.1 Broadcasts with on-demand copies

Figure 9.15 gives an example of the data structure for generalized broadcast tables. A first modification compared to broadcast tables with a single copy of the data consists in the replacement of the data pointer in the data structure representing a broadcast table with an array of pointers to input buffers with one entry per NUMA node. After the termination of the producer of a broadcast, this table only contains a single valid entry, pointing to the input buffer of the first reader. To find this initial entry quickly without having to seek through the entire array, the broadcast table contains an additional field called `source_node`, which indicates on which NUMA node the original input buffer is placed. In the example of Figure 9.15, the input buffer of the first reader is located on node one. The pointer to the input buffer of the first reader can be determined in constant time by indexing the `data` array with `src_node`. All data pointers of the peeking views of the remaining readers are initially set to `NULL`, indicating that it has not yet been determined which input buffer will be accessed by a reader. As with broadcast tables using a single copy, the

update of the data pointer is delayed to the beginning of the execution of a reader. The generalized form of broadcast table can take advantage of this mechanism and may create a local copy of the broadcast data as the node on which the reader executes is known at that moment. The steps that are necessary for this update are carried out by the procedure `prepare_peek_data`, which is shown in Algorithm 10.

The first lines of the algorithm determine which worker is executing the reader whose peek view has been passed to the function, its NUMA node and the broadcast table of the view. The test in Line 6 checks whether a copy of the data is already available on the local node. If this is the case, the procedure simply sets the data pointer of the view to the appropriate entry of the array of the broadcast table and returns (Lines 7 and 8).

The first check for a local copy might fail in two situations. As `prepare_peek_data` can be called concurrently for many readers at the same time, it might be that a local copy is currently being created and will become available shortly after the check, indicated by the value *updating* in the array of pointers to local copies. In this case, the reader can either wait for the copy to be available by going back to the first check, as in Line 13 (busy waiting), or the reader can decide to use a remote copy instead, as in Line 15. Whether the reader waits or not can be set through a configuration option of the run-time at compile time, referred to as `busy_wait` in the algorithm.

If no other worker is currently creating a local copy, the reader allocates a copy itself as shown in Lines 18 to 27. The reader needs to update the according entry in the array of copies atomically in Line 19 to indicate to concurrent readers that a copy is being created. This update might fail if another worker has decided to do likewise between the last check and the attempt of the atomic update. If this is the case, the worker simply needs to check the status of the copy again by going back to the label `retry`. If the atomic update succeeds, the reader is responsible for the allocation and initialization of the local copy and determines the local memory pool (Line 20), allocates an input buffer (Line 21), copies the contents of the original buffer (Line 22) and updates the broadcast table (Line 23).

Figure 9.16 illustrates the execution of Algorithm 10 for the peek view of the nth reader t_c^{n-1} of a broadcast. The initial situation is shown in Figure 9.16a, in which already two copies of the input data exist on nodes one and two and where t_c^{n-1} is executed by a worker on node $N-2$, with N standing for the total number of NUMA nodes of the system. As the entry of the data array of the broadcast table is initially `NULL`, the reader allocates its own buffer and sets the corresponding entry to *updating* indicated by the letter U in Figure 9.16b. Next, it copies the contents of the original buffer to the newly allocated local buffer as shown in Figure 9.16c. As the broadcast table indicates that the source node is the second NUMA node with an identifier of one, the pointer to the original buffer is retrieved from the second entry of the `data` array. Note that during the copying, the entry for the $(N-1)$th node remains *updating*, since setting it to the address of the newly allocated buffer could result in read accesses to incomplete data by concurrent readers of the same node. When all data has been copied to the new buffer, the corresponding entry in the data array of the broadcast table can be set to the buffer's address (Figure 9.16d). The reader t_c^{n-1} becomes ready for execution and reads its input data from the local node.

As there is exactly one single entry per NUMA node in the data array of a broadcast table, the maximum number of copies per broadcast is limited to the number of NUMA nodes. For a high number of readers in a broadcast, the memory footprint is thus lower than in the default broadcast mechanism with per-reader copies, but higher than for broadcast tables with a single copy. Similarly, the overhead for a copy is only generated for the first reader on a node, resulting in less time spent on copying than by the default mechanism and more time compared to broadcast tables with a single copy. However, all subsequent readers executing on the same node can simply reuse the appropriate entry of the data array without any overhead, resulting in accesses to local memory during execution and less contention on the node containing the input buffer of the first reader of the broadcast [4].

4. If busy waiting is disabled, memory accesses might still be remote if subsequent readers on the same node become ready before the copy is ready.

Algorithm 10: prepare_peek_data(v_p)

```
1  w ← this_worker()
2  node_w ← local_node_of_worker(w)
3  bt ← v_p.bctable
4
5  retry:
6  if bt.data[node_w] ≠ null and bt.data[node_w] ≠ updating then
7  |  v_p.data ← bt.data[node_w]
8  |  return
9  end
10
11 if bt.data[node_w] = updating then
12 |  if busy_wait then
13 |  |  goto retry
14 |  else
15 |  |  v_p.data ← bt.data[bt.src_node]
16 |  |  return
17 |  end
18 else
19 |  if atomic_set(bt.data[node_w], updating, null) = success then
20 |  |  pool ← memory_pool_of(node_w)
21 |  |  v_p.data ← alloc(pool, v_p.horizon)
22 |  |  memcpy(v_p.data, bt.data[bt.src_node], v_p.horizon)
23 |  |  atomic_set(bt.data[node_w], v_p.data, updating)
24 |  |  return
25 |  else
26 |  |  goto retry
27 |  end
28 end
```

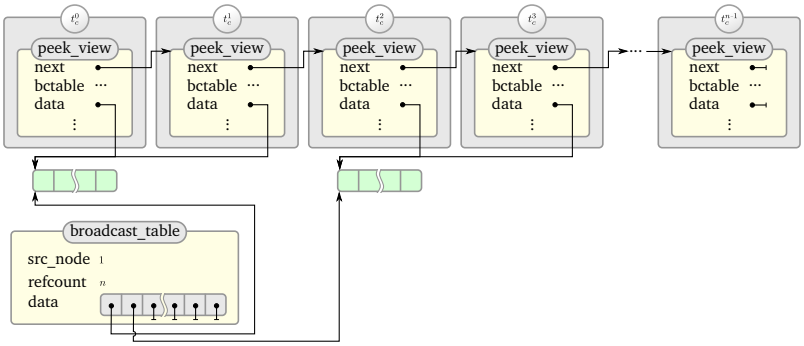
9.4.2 Experimental evaluation

To evaluate the impact of the generalized form of broadcast tables on performance and on the memory footprint, we have executed the *cholesky* benchmark with two configurations of a run-time implementing benchmark tables with per-node copies. The first configuration, named *busy waiting*, uses the busy waiting feature of Algorithm 10 to wait for the completion of a copy, while the other configuration, labeled *nowait*, refers to the copy on the source node if a copy to the local node is in progress. For the comparison with broadcast tables using only a single copy, we have also added a configuration named *single copy* to the graphs, which corresponds to the broadcast tables of Section 9.1 [5].

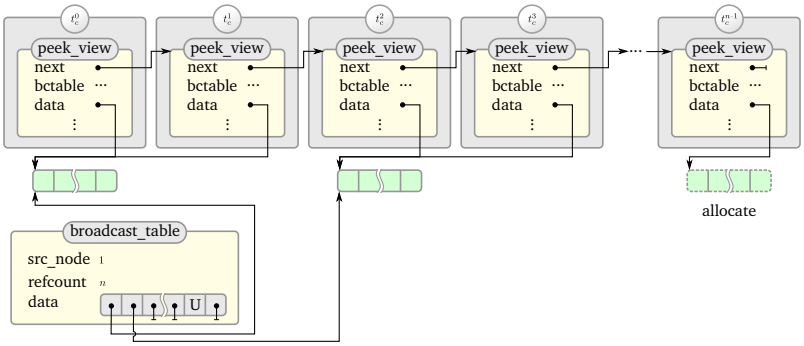
Figure 9.17 shows the median memory footprint of 50 runs of *cholesky* with each configuration of the run-time. Error bars indicate the standard deviation. As expected, broadcast tables with multiple copies increase the memory footprint on both platforms for larger matrices compared to broadcast tables with a single copy. As the number of copies is limited by the number of nodes, this increase is higher on the SGI system than for the Opteron platform. The differences between the configuration with busy waiting and the configuration that does not wait until a local copy is available are negligible.

Figure 9.18 shows the fraction of requests to local memory on the Opteron platform measured

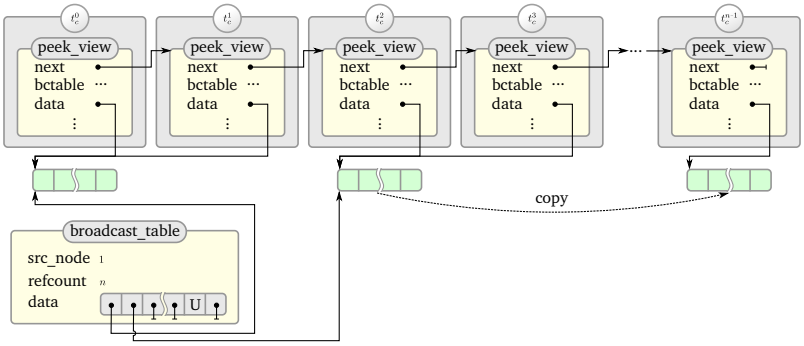
5. Technically, the broadcast tables of Section 9.1 have been implemented with the same data structures as for broadcast tables with multiple copies, but with a modification of Algorithm 10 that forces all readers to use the copy on the source node.



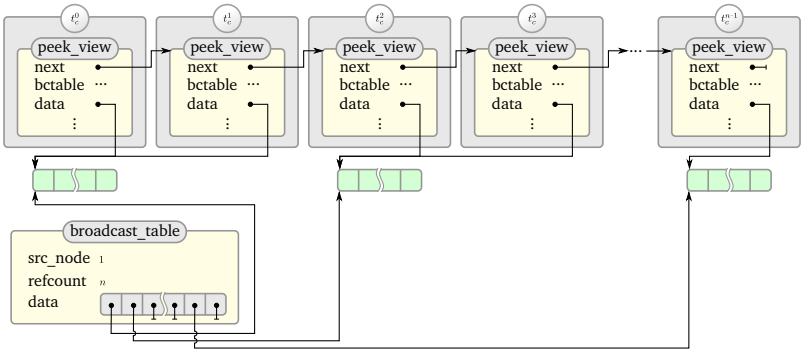
(a) *Initial situation*



(b) *Allocation of an input buffer on the target node and update of the data pointer in the broadcast table to* updating

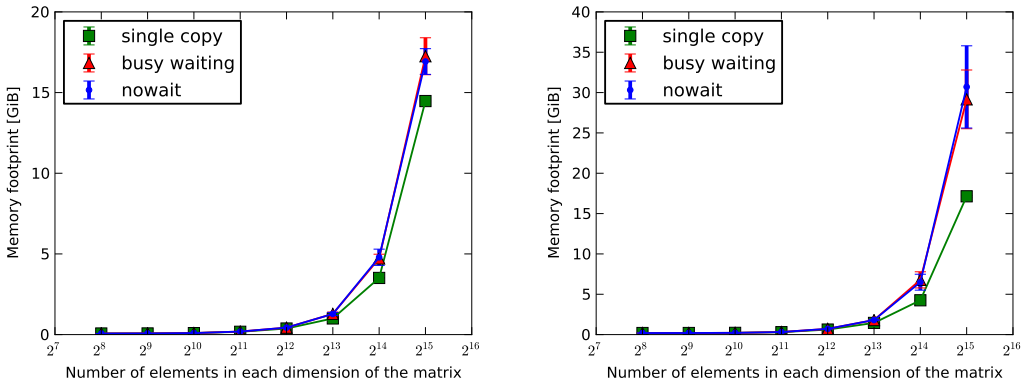


(c) *Transfer of data from the source node to the new buffer*



(d) *Update of the entry in the broadcast table*

Figure 9.16: *Broadcast table with node-local copies*



(a) *Opteron platform* (b) *SGI system*

Figure 9.17: *Memory footprint of broadcast tables with local copies*

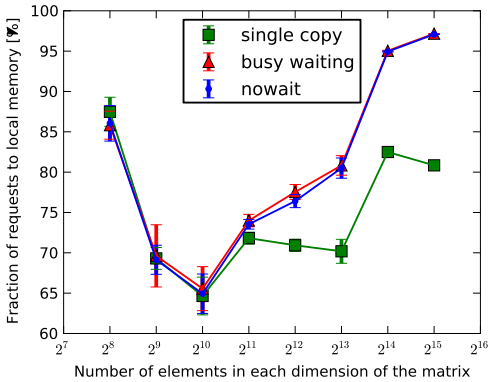


Figure 9.18: *Fraction of requests to local memory of broadcast tables with local copies on the Opteron system*

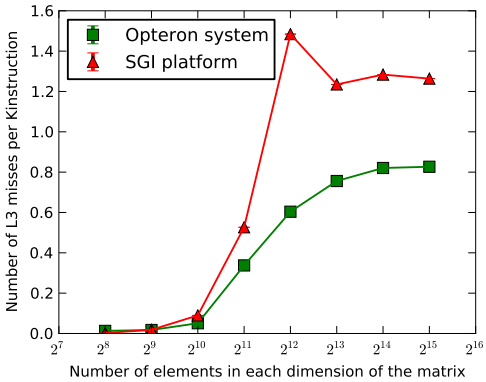
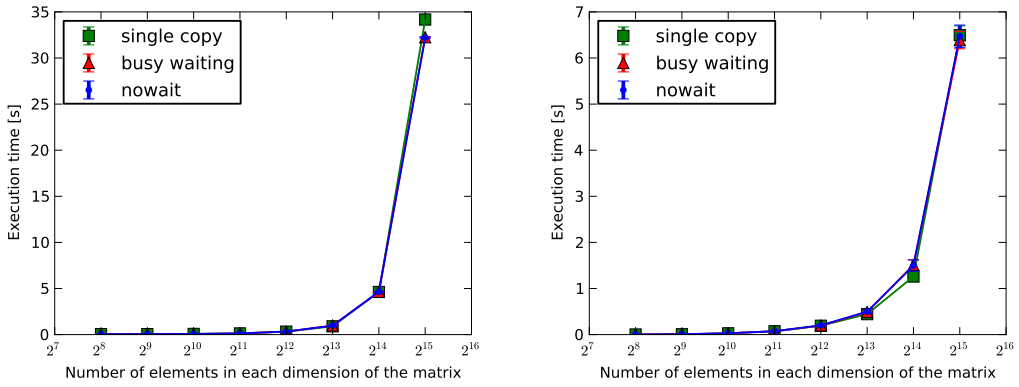


Figure 9.19: *Number of last level cache misses per thousand instructions of* cholesky *using broadcast tables with a single copy*

with hardware performance counters. Due to the absence of these counters on the SGI platform, we only provide statistics on the locality for the Opteron system. For very small matrices, the locality is high due to the low number of blocks (a single block for matrices with $2^8 \times 2^8$ elements, four blocks for $2^9 \times 2^9$ matrices and so on) and a high probability of tasks processing these blocks to be stolen by workers on the same node. The lowest locality is achieved for matrices with $2^{10} \times 2^{10}$ elements. For larger matrices parallelism increases, such that the probability of remote steals becomes lower. However, the configurations with local copies yield a significantly higher locality for accesses to main memory, reaching more than 95% for matrices with $2^{15} \times 2^{15}$ elements.

To estimate the impact of the improved locality on performance, we have also measured the number of last level cache misses per thousand instructions. Figure 9.19 shows the median number of last level cache misses for 50 instructions multiplied by thousand and divided by the median number of instructions for *cholesky* using broadcast tables with a single copy. In comparison with the number of misses per thousand instructions of the other benchmarks, presented in Figure 6.21a and Figure 6.21b in Section 6.5 on page 127, these values are low and *cholesky* can be considered as cache bound. The improvement of local copies on performance is thus expected to be low.

Figure 9.20 shows the median execution time as a function of the matrix size for both test platforms. The values for the three configurations are nearly identical. This confirms the above assumption that data locality only has little influence on the performance of this benchmark. However, applications with a higher frequency of last level cache misses and thus a higher



(a) *Opteron platform* (b) *SGI system*

Figure 9.20: *Execution time of* cholesky *using broadcast tables with a single copy and local copies*

frequency of accesses to main memory could take advantage of the locality improvement and their execution time could be reduced. Furthermore, the performance of these applications might differ depending on whether busy waiting is employed or not.

9.4.3 Conclusion

The analysis of the results for the *cholesky* benchmark showed that broadcast tables with on-demand node-local copies yield a significantly higher fraction of accesses to local memory, which comes with a substantial increase of the memory footprint. However, the execution time of *cholesky* cannot be improved by these techniques due to the application's high cache hit rate. Hence, broadcast tables with a single copy for all readers of the broadcast are the best choice both in terms of performance and the memory footprint for the *cholesky* benchmark.

However, other benchmarks with different characteristics, such as more last level cache misses than *cholesky* or other timings for tasks to become ready might benefit from broadcast tables with multiple copies. The conditions under which this is true are to be determined in future work.

9.5 Summary

In this chapter we analyzed the memory footprint, the execution time and the data locality of broadcasts in OpenStream. We showed that both the footprint as well as the average waiting time of a reader until activation are proportional to the number of readers participating in the broadcast. We then introduced broadcast tables that allow multiple tasks to share a single input buffer and showed that the memory footprint remains constant, independently from the number of readers involved in the broadcast. Depending on the size of the data to be broadcast, this results in an improvement of the average waiting time and a reduction of the memory footprint by more than one order of magnitude. The experimental evaluation involving a dynamic single assignment implementation of Cholesky Factorization using OpenStream, a shared memory implementation for many-core systems based on PLASMA and an implementation using the parallel routine for Cholesky Factorization of the INTEL MATH KERNEL LIBRARY showed that using broadcast tables, OpenStream is able to match the performance of state-of-the-art linear algebra libraries for many-core systems.

As an outlook to further optimizations regarding the locality of memory accesses related to broadcasts, we analyzed the performance, the memory footprint and the data locality of a generalized concept of broadcast tables with on-demand creation of per-NUMA node copies of data. For the *cholesky* benchmark these optimizations yield a higher data locality, but an increased memory footprint and the same performance as broadcast tables with a single copy.

Further research is thus necessary to determine under which conditions these optimizations can be beneficial for the execution time.

10 Performance analysis of task-parallel programs and run-times

The performance of task-parallel programs depends on many aspects, ranging from static code optimizations by the compiler or manual data-layout transformations by the programmer to dynamic optimizations regarding the structure of the task graph, the order of task creation and interactions with the operating system and the underlying hardware architecture. Identifying performance anomalies and finding their cause requires a detailed understanding of all of these aspects. In particular, a programmer needs to understand the complex interactions between the software and hardware components involved in the execution. One way to analyze performance is to collect and record all relevant dynamic events into a trace file and to use a tool for off-line analysis after termination of the program. A visual representation of events, system entities and their relationships is an approach to provide the necessary insight for an accurate analysis, sorting causes and effects and distinguishing application-specific anomalies from inefficiencies in the heuristics used by the run-time system. During the last decades, a multitude of tools for trace-based analysis have been developed, e.g. [69, 67, 1]. However, most of them target distributed applications executing on clusters systems that communicate through message passing and thus do not natively support performance analysis of task-parallel applications and run-time systems.

In this chapter, we present Aftermath, a tool for interactive, off-line visualization, filtering and analysis of execution traces that we have developed during this thesis primarily for performance debugging of OpenStream applications and the OpenStream run-time. The tool has been used extensively during the implementation of the benchmarks presented in Section 6.1 and for performance debugging of the optimizations introduced in Chapters 7, 8 and 9 and allowed us to gain deep insight into the interactions between the application, the run-time, the operating system and the hardware. However, Aftermath provides a large set of tools that apply to performance analysis of task-parallel applications and run-time systems in general and can thus be reused for performance debugging of other task-parallel languages as well. Different key metrics and indicators can be displayed jointly, which accelerates the discovery of significant correlations. For more complex relationships, Aftermath offers powerful filtering mechanisms and is able to match relevant information with the topology of the machine. A responsive graphical user interface gives quick access to all of these features, allowing to explore traces rapidly and to control the degree of detail that is needed for the analysis.

The chapter is organized as follows. In Section 10.1, we identify the requirements for performance analysis in general before we give an overview of Aftermath in Section 10.2. Section 10.3 provides examples of the use of Aftermath for performance debugging of task-parallel applications. Section 10.4 illustrates how Aftermath can be used for performance debugging of the run-time

system. Directions of ongoing and future research on guided performance analysis are pointed out in Section 10.5. A brief discussion of related work is provided in Section 10.6 and a summary is given in Section 10.7. *Parts of this chapter were previously published in [47, 45].*

10.1 Requirements for trace-based performance analysis

The interactions between hardware and software components involved in the execution of a task-parallel program can generate a high number of dynamic events, especially on many-core systems with dozens or hundreds of cores. Deriving statistics from these events or filtering events relevant to a specific kind of performance analysis is thus likely to be computationally intensive, such that on-line analysis during execution can have a significant impact on the system that is being analyzed and thus lead to biased results. For example, to determine the average duration of all tasks belonging to a certain set of task types in a specific interval, it is necessary that each worker checks for each task it executes if the type of the task belongs to the set and if its beginning and end of the execution lay within the interval. Furthermore, the duration of tasks matching these criteria must be calculated and stored at a memory location that is known during calculation of the average duration. Especially the last step might change the timing of events during execution of the program. If the average is calculated by a single worker, this worker cannot execute tasks during the calculation, which decreases parallelism. If the average is calculated concurrently, multiple workers must synchronize on intermediate results, which might lead to differences in the timing or have an influence on micro-architectural events (e.g., on the number of memory accesses if spin-locks are used for synchronization). More complex analyses usually require more processing time and might thus change the timing at execution time substantially. In addition, the types of analyses must be known in advance, before execution of the application and it is impossible to carry out additional analyses for the same execution after termination of the program.

A common strategy to deal with the issues above is to rely on *off-line analysis* of *trace files*. In this approach, a subset of the dynamic events related to the execution of an application is written to a trace which is only analyzed once the application has terminated. Besides the overhead for the collection of relevant events at execution time and for storing the events to the trace file, the analysis does not have any impact on the system that is being examined. This allows a user to investigate all relevant aspects of the execution without limitations for the duration or the amount of memory that is required for the analysis. If additional analyses are needed, these can be based on the events already collected the trace file and do not require re-executing the application.

The presentation below first introduces two recurring scenarios for performance debugging, referred to as trace exploration and hypotheses testing. We then present the requirements on trace visualization and filters for trace data, before we give a brief overview of how trace data is collected in the OpenStream run-time.

10.1.1 Trace exploration and hypothesis testing

We identified two key scenarios frequently occurring in the performance debugging based on execution traces. In the first case, the programmer suspects that there is a performance anomaly or is looking for optimization opportunities, but has not identified any specific issues. Browsing through an execution trace, which we refer to as *trace exploration*, can help build up a hypothesis by identifying program behavior that leaves room for improvement. In the second case, the programmer has already developed one or more hypotheses and tries to confirm or to refute them. In the rest of the chapter, this scenario is referred to as *hypothesis testing*. Performance debugging is often an iterative refinement process, alternating between these two situations, as shown in Figure 10.1. Usually, the programmer starts by executing a program, explores the trace containing the events collected at execution, identifies possible sources of performance anomalies, tests the hypotheses and finally fixes the issues in the application or run-time system. As trace exploration and hypothesis testing are the cornerstones of performance analysis, a program for trace analysis should provide appropriate tools for data selection and examination that fit both situations.

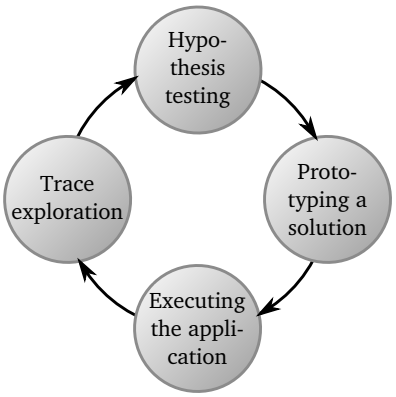


Figure 10.1: *Stages in the development of task-parallel applications and run-times*

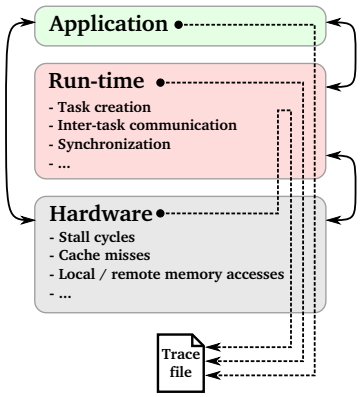


Figure 10.2: *Capturing events related to the interactions between the application, the run-time system and the hardware*

10.1.2 Trace visualization

Execution traces of task-parallel applications generally contain two types of information. The first type relates to static information about the execution context, e.g., the number of cores, the machine topology and the different tasks or work functions, while the second type refers to dynamic information on execution events, e.g, worker state transitions, communication events and samples collected using hardware performance counters. For efficient analysis, the basic topological, temporal and relational aspects need to be represented adequately at the same time. In particular, the user should be able:

- to distinguish the activity of different cores and worker threads,
- to observe activity over time and the evolution of metrics,
- to precisely identify the different types of events, and
- to determine involved entities, e.g., source and destination of data exchanges.

Visualization is an appropriate method to present large quantities of events and provides various means to present multi-dimensional data (e.g., by the position on the screen, colors, patterns, etc.). A graphical representation should provide adequate support to make apparent any strong correlations between events. For example, if a performance issue only occurs on specific cores, in specific intervals or after specific events, this behavior should be directly identifiable on the visual representation.

The interactive exploration of traces is an essential aspect that provides a quick overview of the trace data and helps to develop a working hypothesis. Navigation along the different dimensions, e.g., changing the interval to be displayed or limiting the graphical representation to a subset of cores should therefore be intuitive to the user. With trace files of up to several gigabytes, containing hundreds of thousands of events, rendering needs to be sufficiently fast for interactive trace exploration.

10.1.3 Control over the amount of detail

For the exploration of specific aspects or in order to reduce the amount of data that is visualized, it should be possible to filter the information from the trace, such that only relevant information is displayed. To avoid interrupting the user's work-flow, there should not be any notable delays and the result should be visible immediately when the filter is applied. Filters also represent an essential tool for hypothesis testing. To check if an assumption is correct, the user needs to filter out all situations for which the premise of the hypothesis does not hold. As conditions can be complex, it should be possible to combine filters easily.

However, even with powerful filtering schemes, visual feedback is not always sufficiently precise for a distinct conclusion. In such cases it may be necessary to statistically correlate events, which means that it should be possible to aggregate trace data and display statistical information on event distributions, either presented in separate views or along with the information that was quantified. The latter case might enable the user to draw conclusions on relationships between existing and newly aggregated aspects. If none of the basic statistical counters alone can provide enough information about a relationship, it is essential to be able to combine multiple counters. The user should be guided through this process by a user interface that allows to precisely select which information should be derived and how it should be displayed.

Finally, it must also be possible to obtain detailed information about specific events. This can help to detect outliers or to develop generalized rules from particular situations. For example, the user could select a few corner cases for task duration one after another and then try to figure out the generalized conditions for fast or slow task execution.

10.1.4 Recording execution traces of task-parallel applications

The collection of trace data itself also plays an important role for performance analysis. This involves the methods for data collection during execution as well as the definition of a file format that is suited to store all relevant events. However, as this chapter focuses on methods for the presentation and processing of existing traces, we only motivate basic requirements for the collection of trace data and give a short overview of the instrumentation of programs and the run-time, enabling support for execution traces in OpenStream.

To keep the amount of work for programmers to support execution traces low, tracing should be implemented as a generic, application-independent mechanism. Ideally, the instrumentation of the application should be done fully automatically or tracing should be provided automatically through underlying software and hardware interfaces. For programs whose execution is managed by a run-time system, tracing is thus often implemented transparently to the application within the run-time and does not require any specific support by the application. However, it should also be possible to record application-specific events and high-level information that cannot be derived automatically, e.g., the beginning or the end of measurement intervals. Support for low-level events, such as hardware performance counters or statistics obtained from the operating system is essential for the analysis of the interplay between hardware and software components.

The format of trace files should also be application-independent, such that a generic tool for trace analysis supporting this format can be used to analyze a wide variety of applications. For example, instead of using a fixed set of task names with a predefined meaning, the format should support the definition of a variable set of task types. To keep the size of trace files low, the format should contain as few redundancies as possible. In addition, a binary representation for trace data is preferable as binary representations often require less space than textual representations and can be parsed faster when the trace file is loaded by the performance analysis tool. Support for large quantities of events and large file sizes however are mandatory, especially on large systems with high numbers of cores.

Figure 10.2 shows a basic view of the implementation of tracing in OpenStream. The run-time is responsible for the collection of all events as well as for the creation of the trace file. To avoid time-consuming system calls related to tracing and thus to reduce the overhead of serializing events into the trace file, all events are collected in main memory and are only written to the trace file at termination of the program. The majority of the events are generated by the run-time itself. Examples of such events are task creations, information about accesses to views, task destructions, the beginning and end of the execution of a task, work-stealing and work-pushing events. The only application-specific events that are currently supported are the beginning and the end of measurement intervals, which can be recorded through simple calls to run-time functions as described in Section 6.2.3. Low-level events from the hardware are recorded by the run-time using the PAPI [80] library, which provides access to monotonically increasing counters available on the target system that measure how many micro-architectural events of a certain type have occurred.

The set of events to be sampled is set up at the beginning of the execution, but the counters are only enabled during measurement intervals to avoid including unrelated micro-architectural events in the trace. Although the period for the sampling of hardware performance counters during the measurement intervals can be arbitrary, we have chosen to sample each counter only before and after the execution of each task. While this leads to sampling periods of variable length and a relatively low resolution, this strategy allows a tool reading the trace to derive statistics for each individual task (e.g., the number of cache misses that have been generated during execution of each task). Moreover, a low sampling period helps to keep the size of the trace file small.

In the next section, we introduce Aftermath, a tool that we have developed to meet the requirements outlined above for the analysis of execution traces of task-parallel programs and run-time systems. Examples for the use of Aftermath for performance debugging of task-parallel applications and run-time systems are given in the following sections.

10.2 Aftermath

We have designed and implemented Aftermath [1] for fast, interactive, visual exploration and analysis of traces generated by fine-grained task-parallel applications and their run-time systems, executing on modern many-core architectures. In this section, we give an overview of the design of Aftermath and its features, we present the layout of its graphical user interface, we give an outline of the required trace format and we explain how Aftermath can exploit information from application symbol tables and trace annotations. Although Aftermath has primarily been developed for OpenStream applications and the OpenStream run-time, many of its concepts apply to task-parallel languages and run-times in general.

10.2.1 Organization of the main user interface

Figure 10.3 shows the main window of Aftermath during analysis of a trace file. The various elements of the user interface are grouped into five different parts:

1. The *timeline component* in the center of the user interface shows the activity of each of the cores over time (e.g. the different states of the worker threads associated to the cores, evolution of performance counter data and specific discrete events, such as task creations, and communication between workers).
2. The right side contains a group of *statistical views* aggregating individual events in order to quantify basic information for an interval from the timeline view selected by the user (e.g., a histogram showing the distribution of task durations, a text field indicating the average parallelism and a communication matrix indicating which cores and nodes communicate).
3. A set of *filters* for various basic properties at the left side allows the user to control what is shown in the timeline component and in statistical views (e.g. only tasks of a specific type, tasks whose execution duration is in a certain range, tasks that write to certain NUMA nodes, etc.).
4. The bottom part is reserved for *detailed textual information* about a selected state and the task execution associated to it (e.g. the task and state type, the duration and data-flow-specific information about the producers of the task's input data as well as the consumers of its output data).
5. A menu bar at the top provides access to a set of *generators* for metrics derived from high-level events or metrics that combine existing statistical counters (e.g. the average task duration, number of bytes exchanged between specific NUMA nodes, the ratio of two hardware performance counters, etc.). Selecting the appropriate menu entry opens the corresponding dialog that guides the user through the creation of a derived metric.

Aftermath supports arbitrary zooming and scrolling along the timeline through an intuitive interface. Filters directly affect the information shown in the timeline and the statistical views

1. Available under a GNU GPL license at `http://www.openstream.info`

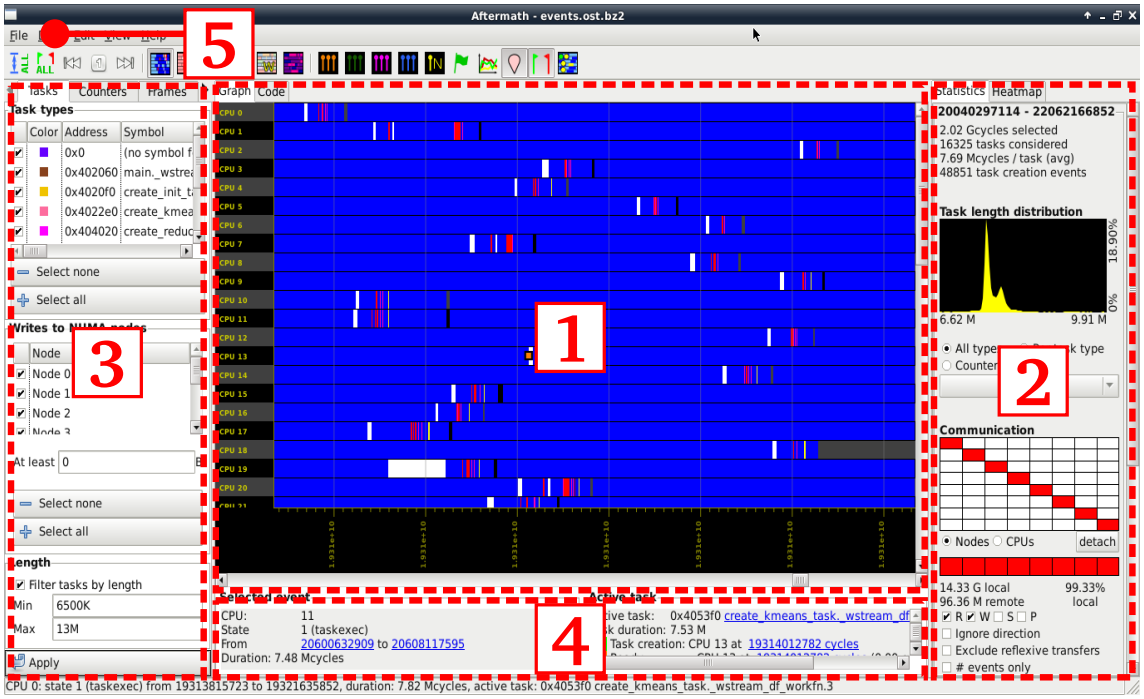


Figure 10.3: *Aftermath's main window: timeline (1), filters (2), statistics (3), information on selected tasks / events (4) and menu bar for derived metrics (5).*

for the selected portion of the trace to provide immediate visual feedback. Rendering has been optimized carefully, such that no delays interrupt the user's work-flow. During development of Aftermath, we found that complete traversal even of multi-gigabyte traces only represents a small fraction of the rendering time. Displaying only information that is visible at the selected zoom level reduced the overall delay sufficiently. For example, instead of rendering all the state changes in the timeline, only states that represent a relevant part of the interval defined by a pixel on the screen are shown. For a set of communication events whose communication lines overlap, only one line is drawn. The resulting rendering operations are carried out by the CAIRO GRAPHICS LIBRARY [81]. For standard user interface components we have used GTK+ [82].

As the size of the different parts suggests, the main visual representation is the timeline component. The user can choose a mode for visualization for the timeline from a set of modes, each of which highlights specific aspects of the trace. The modes currently supported by the timeline are the following.

- The default *state mode* shows which states the workers traverse over time. Aftermath supports a handful of different states, which are mainly related to activities of the run-time. For example, there are states for task execution, task creation, broadcasts and dependency resolving of tasks. The state mode of the timeline allows the user to identify visually which workers and how much time these workers spent in a particular state.
- In *heatmap mode*, the timeline represents the duration of tasks with different shades of red. We refer to the visual representation of the timeline in this mode as a *heatmap* for short. Phases during which a worker executes slow tasks are rendered in the heatmap using dark red, while phases with fast tasks are white. The interval that defines which tasks are considered as slow and which tasks are considered as fast can be set by the user or can be determined automatically by Aftermath from the minimal and maximal duration of all tasks present in the trace or from a subset of tasks defined by a filter. The number of shades that are used for rendering is also configurable.
- The timeline in *task type mode*, also called *typemap*, associates a different color to every task

type found in the trace and shows which type of task each worker executes over time. The term *task type* refers to a task construct in the source code. For example, an application with three task constructs, one for tasks that perform matrix multiplication, one for auxiliary tasks for the initialization of the application and one for the termination defines three task types. Instances of these task types might be rendered using blue, green and yellow, respectively. This allows the user to identify at which moments of the execution the different types of tasks are executed and where they execute.

– When the timeline is in *NUMA mode*, it associates a color to each NUMA node and shows which nodes are targeted by memory accesses performed by the tasks executed by each worker over time. This information is derived from the addresses of memory accesses and information on data placement present in the trace. The graphical representations generated in NUMA mode are called *NUMA maps*. There are two kinds of NUMA maps. The first map only takes into account read accesses and thus indicates which nodes are targeted by reads. The other map analyzes write accesses and thus shows which nodes are targeted by write accesses. More details on these views are provided in Section 10.4.1.
– The last mode that the timeline can be set into is called *NUMA heatmap mode*. Similar to the NUMA modes, Aftermath analyzes memory accesses and placement information in this mode to derive NUMA-related information. However, in NUMA heatmap mode both read and write accesses are taken into account and combined with information about the topology of the machine, also present in the trace. The result is a view that indicates the average fraction of remote memory accesses per interval with different shades from violet to pink. Intervals during which many tasks with a low fraction of remote memory accesses were executed are rendered in violet, while intervals with a high fraction of remote accesses are pink. This view is useful especially when the NUMA read and write modes show that a lot of different nodes were targeted and thus do not show a clear relationship between accessing nodes and the targeted nodes.

The different components of the user interface and the different modes for the timeline require different kinds of information to be present in a trace. In the following paragraphs we briefly discuss the layout and the types of information that can be stored in a trace file for Aftermath.

10.2.2 Trace format

Trace files for Aftermath are organized as streams of data structures, which can either contain events (i.e., state changes, hardware performance counter values, communication events or discrete events, such as the creation of a task or beginning and end of task execution), topological information about the machine (e.g., how cores relate to the system's NUMA nodes), descriptions of hardware performance counters or information about the location of OpenStream-specific dataflow buffers. Structures can appear in any order, e.g., the trace might contain events of different cores in an interleaved fashion, as long as total order with respect to the timestamp for events is preserved for each core. The interleaving of events from different cores keeps the overhead low during collection of trace data and when this data is written to a file as no time-consuming sorting is necessary. The total order of events per core limits the overhead when a trace file is loaded from the disk.

Aftermath currently defines a native trace format, which is optimized for the OpenStream run-time and for OpenStream applications. However, not all types of data structures are mandatory for all kinds of analyses that are supported by the tool. Trace files that omit OpenStream-specific events can still be loaded and only limit the set of analyses that can be carried out on the trace. For example, if a trace file only contains events marking the beginning and the end of the execution of each task but does not include information on accesses to input buffers, Aftermath cannot provide information on the locality of memory accesses but can still be used for analyses based on task durations and is able to visualize data for hardware performance counters. Hence, although Aftermath has been developed primarily for the analysis of OpenStream programs and the OpenStream run-time, it is also suited for the analyses of applications and run-times of other task-parallel

languages.

As traces can contain hundreds of thousands of events, trace data is stored in a binary format to reduce its size and to avoid long parsing delays when a trace is opened. Further reduction of the file size is achieved through the compression of traces with standard GNU/Linux tools, such as GZIP, BZIP2 or XZ. Opening a compressed trace causes Aftermath to call the appropriate tool for decompression and to read uncompressed data from an unnamed pipe.

Finally, the format was also designed to contain only few redundancies. Information not explicitly available in the trace file, but needed for rendering or generation of basic statistics is added to the internal representation when the trace is loaded into main memory or when the information is needed during rendering. For example, the identifiers of NUMA nodes targeted by accesses to input buffers are not stored in each data structure describing these accesses, but are retrieved from the addresses of the accesses and the data structures describing the placement of input buffers. This way, the information on the placement is only stored once for each buffer regardless of the number of accesses.

10.2.3 Symbol tables and annotations

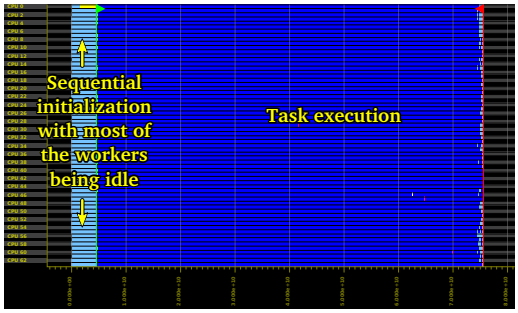
The development cycle for task-parallel applications and run-time systems introduced in Section 10.1 contains steps in which the programmer inspects a trace and locates performance bottlenecks as well as steps in which the code of the implementation needs to be modified. To support these alternations during application development, Aftermath is able to relate the information of the visual representation to the source code of the application. If the user specifies the path to the executable of the application with debug symbols that was executed to obtain the trace in addition to the name of the trace file on startup, Aftermath extracts these symbols and the location of each symbol in the source code using standard command-line tools of the GNU/Linux system, in particular NM. When a task is selected from the timeline, the tool retrieves the address of the associated work-function, looks up the corresponding entry in the table with debug symbols and shows the name of the work-function in the detailed text view. A click on this name starts an editor that opens the source file containing the function and jumps to the correct line.

However, in the default naming scheme, the OpenStream compiler automatically generates the names for the work-functions by concatenating the name of the function containing the `task` construct with the string `_wstream_df_workfn_` and a sequential number, which makes it difficult to distinguish several tasks have been defined within the scope of the same function. Therefore, we have added a new clause to OpenStream called `task_name`, which allows the programmer to define a custom name for the work function associated to a task construct.

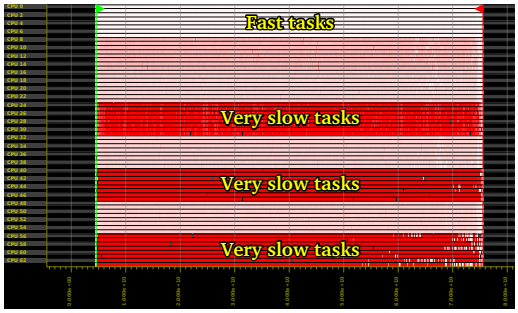
Another useful feature of Aftermath supporting the user in the development cycle are user-defined annotations. A double-click on the timeline opens a dialog that lets the user enter an arbitrary text and choose a color for the new annotation. Each annotation appears as a symbol on the timeline at the time and core that corresponds to the position of the click. When the user moves the mouse pointer over the annotation, Aftermath displays its contents. A double click on an annotation lets the user change or delete the annotation. As Aftermath only displays annotations and does not interpret their contents, annotations can be used to provide arbitrary information on specific events (e.g., to label a specific task execution) or to mark specific moments of the execution of the program (e.g., the beginning or end of an iteration of an algorithm). Since trace analysis can be a time-consuming task taking hours or even days, involving more than one person, annotations can be saved independently from the trace file and loaded for further analysis at a later point in time.

10.3 Debugging application performance

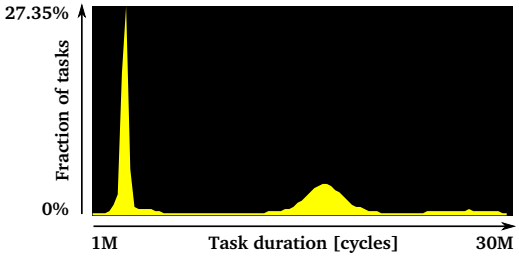
After the overview of Aftermath above, we now present two cases that deal with issues that we have encountered during development of benchmarks for the OpenStream project and that illustrate how Aftermath can be used for performance debugging of task-parallel applications. The



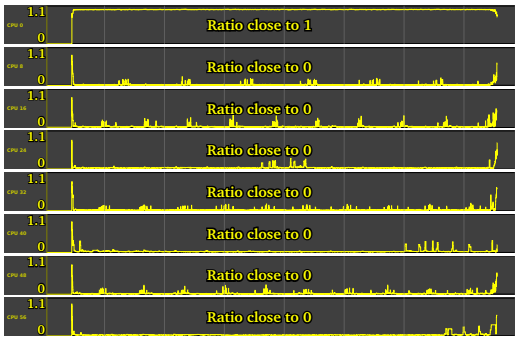
(a) *All of the 64 workers are in task execution state (dark blue) for almost the whole execution*



(b) *Heatmap mode indicating a relationship core → task duration*



(c) *Distribution of the task duration*



(d) *Ratio of local accesses to the total number of accesses*

Figure 10.4: *High-latency memory accesses of* seidel *using a shared matrix*

first example features the detection of a design problem of the shared memory implementation of the *seidel* benchmark that results in inefficient access to main memory. The second example illustrates how multiple views and statistics can be combined to identify an issue related to branch mispredictions in the *k-means* application.

10.3.1 Seidel: detecting contention on memory controllers

As a first example, we show how the cause for the poor scalability of the shared memory implementation of *seidel* with sequential initialization of the global matrix of Section 6.6 can be identified with Aftermath. All trace data has been collected on the Opteron test system with 64 cores grouped into 8 NUMA nodes as described in Section 6.3.1.

Figure 10.4a shows the timeline in its default mode, indicating the states each worker traverses over time. At the beginning of the execution most workers are idle (light blue) due to the sequential initialization of the matrix. This phase is followed by a parallel phase during which the actual computations are carried out. All cores during this phase remain in task execution state until program termination (dark blue), indicating that there is sufficient parallelism available. The beginning and the end of the measurement interval are marked by a vertical green line with a triangle at the top pointing to the right (beginning) and a vertical red line with a triangle pointing to the left (end).

The majority of the tasks in this benchmark should have approximately the same duration due to an identical amount of work, except for a few blocks at the borders of the matrix and auxiliary tasks. However, upon selection of the measurement interval from the timeline to obtain statistical data about task duration, a performance anomaly becomes apparent: the task duration histogram shows an abnormal distribution with several peaks denoting groups of tasks with different execution durations shown in Figure 10.4c.

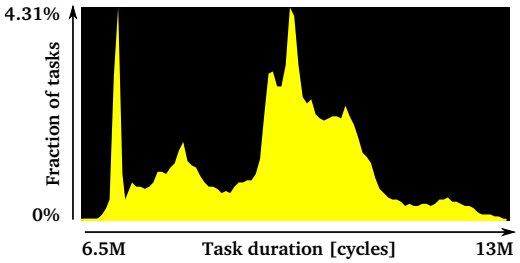


Figure 10.5: *Distribution of the duration of the main computation tasks in* k-means

Figure 10.4b shows the timeline in *heatmap mode*, in which tasks are presented with a different intensity of red according to their execution time, from white for fast tasks to dark red for slow tasks. The presence of large horizontal stripes with the same color indicates that there are no variations of the task duration over time and suggests that the task duration directly depends on the core executing the task. The shortest tasks are executed by cores 0 to 8 (almost white), which are located next to the memory controller of NUMA Node 0. Tasks from cores associated to nodes 3 (cores 23 to 31), 5 (cores 40 to 47) and 7 (cores 56 to 63), which are at a distance of two hops from node 0, have the highest duration (dark red). The shared matrix is thus located on Node 0, which causes memory accesses to be the bottleneck in this application.

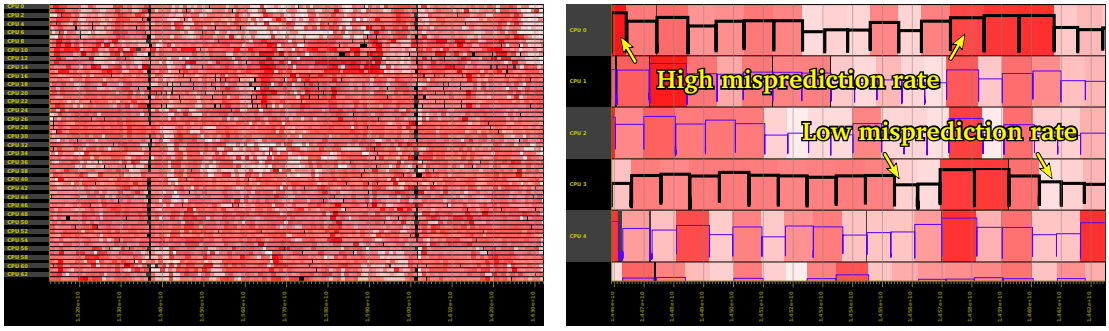
The trace file also contains hardware performance counter data for the number of requests to local and remote memory controllers. On the Opteron platform, these are northbridge-wide counters [11] that aggregate accesses of 8 cores sharing a memory controller. In the trace file, these have been associated to the cores with the smallest identifiers, i.e. cores 0, 8, 16, 24, 32, 40, 48 and 56. Aftermath is able to combine the two counters for remote and local accesses to a derived metric representing the ratio between local accesses and the total number of accesses. Figure 10.4d shows the evolution of this metric extracted from the timeline view, with a vertical plotting range clipped to the interval $[0, 1.1]$. For core 0, associated to Node 0, the value is close to one, indicating a high fraction of local accesses. For the other cores the value is close to zero. Thus, most of the memory accesses are remote, targeting Node 0, which finally explains the abnormal distribution of task durations.

Using interleaved allocation as suggested in Section 6.2.2 partly solves this problem as distributing memory accesses across NUMA nodes reduces the contention on a specific memory controller. After the modification of the benchmark, the task duration is much more uniform, which can be confirmed by loading the trace of the modified benchmark into Aftermath and by verifying that the abnormal distribution is no longer visible in the task duration histogram.

10.3.2 K-means clustering: branch mispredictions

As stated in Section 6.5, the *k-means* benchmark is a compute-intensive application whose performance is mostly insensitive to data placement. However, during development of this application, we encountered a performance anomaly that we first suspected to be related to memory accesses, but which turned out to be related to branch mispredictions. Figure 10.5 shows the task duration histogram after selection of the measurement interval for a trace recorded with our first implementation of *k-means* executing on the Opteron platform. To focus the analysis on the computation, all auxiliary tasks have been filtered out. Although all k-means computational tasks have similar workloads, their execution time is not uniform as indicated by the peaks in the histogram. Contrary to the *seidel* example above, there is no clear and simple relationship between task duration and topology. Each core executes slow and fast tasks during the entire measurement interval as shown in Aftermath's timeline in heatmap mode in Figure 10.6a.

Selecting a slow task from the heatmap and clicking on the task name in the lower part of the main window opens an editor with the task's source code. The innermost loop of the task contains a conditional update of the cluster associated to a point. This results in frequently changing



(a) *The timeline in heatmap mode covering several iterations*

(b) *Zoom with branch misprediction rate*

Figure 10.6: *Heatmap view showing the task duration of* k-means

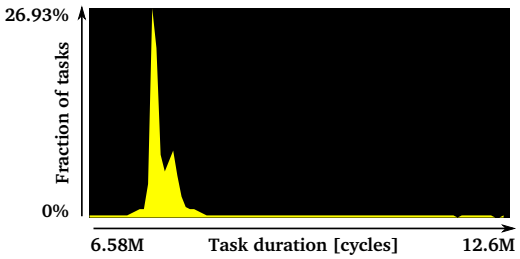


Figure 10.7: *Distribution of the duration of the main computation tasks of the modified* k-means *benchmark*

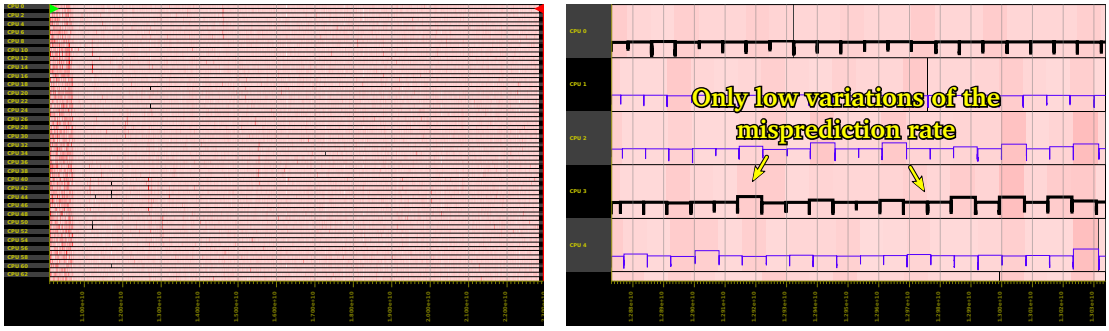
execution paths, which could significantly impact performance if the branch predictor is unable to track the pattern.

Aftermath can display hardware performance counters from a trace file, either directly or after having calculated the (discrete) derivative for each sampling interval. The latter option has been used to generate the branch misprediction count of Figure 10.6b. As the hardware counters for each core are sampled right before and right after task execution, the graph interpolates with a constant value corresponding to the average misprediction rate for each task. The interval represented by the vertical axis has automatically been adjusted to the minimum and the maximum number of branch mispredictions per cycle and corresponds to the interval $[0; 0.009215]$. The combination of the graph and the task duration heatmap immediately reveals a correlation: slow tasks (darker shade of red) have a higher branch misprediction rate than faster tasks (lighter shade).

It is possible to transform the condition in the loop, making the cluster update unconditional, and hoisting the check outside of the time-critical loop. The task duration distribution becomes more uniform, which solves the performance anomaly. Figure 10.7 shows the task duration histogram of the updated version of *k-means* [2]. The number of peaks has been reduced to two close peaks near 6.5 Mcycles. The heatmap shown in Figure 10.8a also confirms that the distribution is much more uniform. Figure 10.8b, showing the zoomed heatmap combined with the average number of branch mispredictions during execution of a task [3], illustrates that the duration of a task is not related to the number of branch mispredictions any longer.

The examples above for *seidel* and *k-means* are excellent examples of performance debugging that does not involve any OpenStream-specific features of Aftermath. The only information that is exploited in these analyses relates to the different states each worker traverses, the duration of tasks

2. The minimum and maximum duration indicated on the left and right side at the bottom are slightly different than those of Figure 10.5. This is due to the fact that Aftermath automatically adjusts this interval according to the shortest and the longest task within the selection on the timeline.
3. The interval represented by the vertical axis has been set to the same interval as in Figure 10.6b.



(a) *The timeline in heatmap mode covering several iterations*

(b) *Zoom with branch misprediction rate*

Figure 10.8: *Heatmap view showing the task duration of the modified version of* k-means *with a lower number of branch mispredictions*

and hardware performance counters, which could be provided by any run-time for task-parallel applications.

10.4 Debugging run-time performance

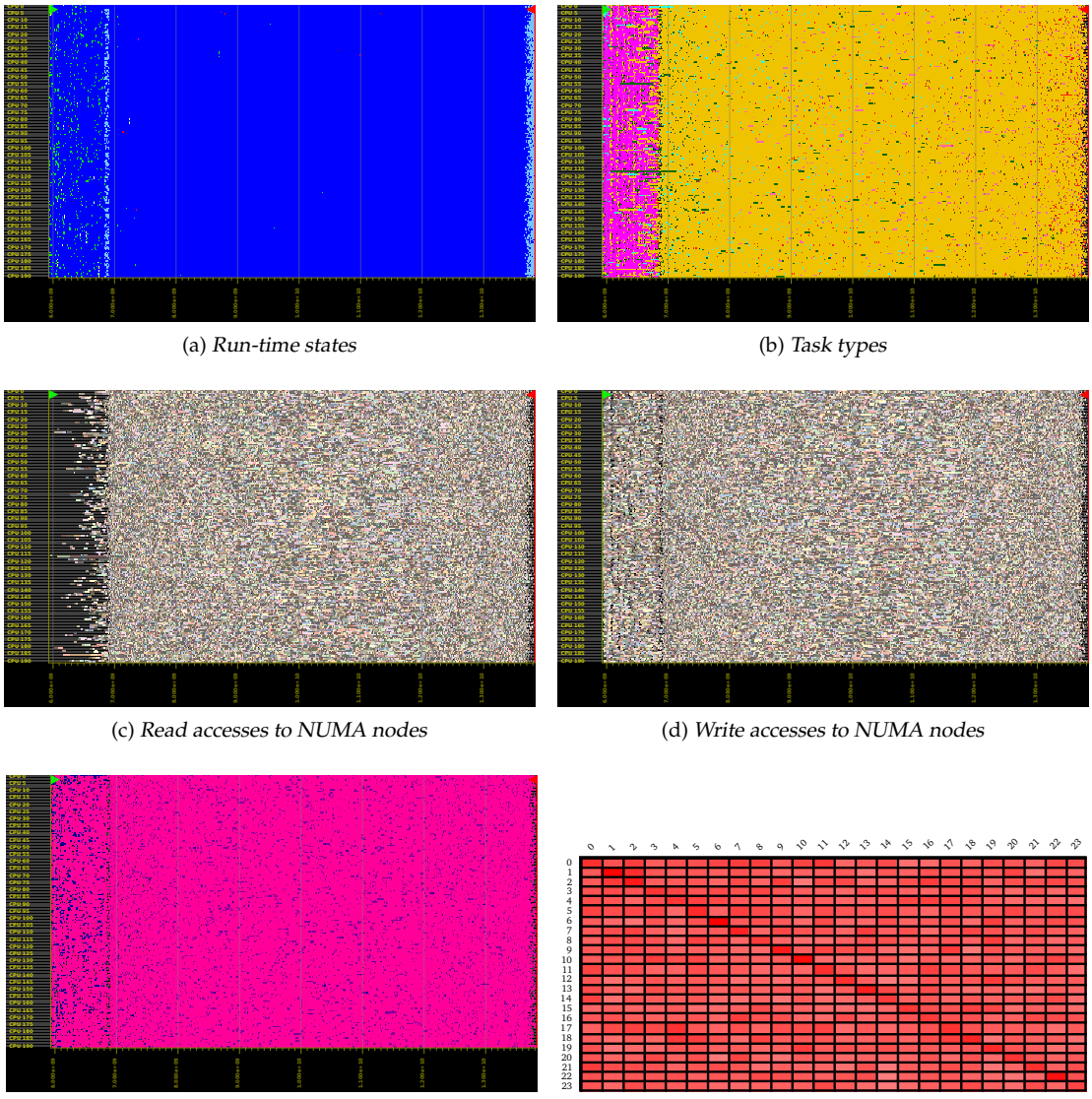
Aftermath has also played an essential role in performance debugging of the OpenStream run-time and for the comprehension and implementation of the optimizations presented in this thesis. The following part starts with an analysis of random work-stealing and shows how Aftermath can be used to verify that topology-aware work-stealing, work-pushing and deferred allocation effectively increase data locality. Afterwards, we use Aftermath to analyze the impact of broadcast tables on the performance of the *cholesky* benchmark.

10.4.1 Deferred allocation and work-pushing

The first use case for performance debugging of the run-time that we present illustrates how Aftermath exploits information on memory accesses from a trace file with topological information on NUMA nodes. We first discuss the behavior of the dynamic single assignment implementation of *seidel* with random work stealing and without work-pushing or deferred allocation and then illustrate how work-pushing and deferred allocation improve the locality of memory accesses resulting in higher performance.

Aftermath's trace format supports the tracing of an arbitrary number of memory accesses as well as an arbitrary number of instances of data structures providing information on relationship between address regions and NUMA nodes. Each memory access in a trace consists of the start address of the memory region that is accessed, its size and the access mode, i.e., whether the access is reading or writing data. Data structures describing the mapping of addresses to NUMA nodes simply consist of a start address, the size of the corresponding data region and the identifier of the NUMA node. The memory region of each memory access must be included entirely in a single data region. If a memory access relates to regions on two or more NUMA nodes, the access must be traced as two or more memory accesses, each targeting only a single node.

Figure 10.9 shows multiple views of a trace of the *seidel* benchmark executed on the SGI platform with a run-time configured for random work stealing and without work-pushing or deferred allocation. The first view, showing the states of the 192 workers over time, indicates that there is only a short phase at the beginning during which parallelism is low. Figure 10.9b shows the timeline in task type mode. As described above in Section 10.2.1, each task construct in the source code is associated to a different color and each pixel on the bar showing the activity of a worker has the color of the construct that took most of the time within the interval represented by



(a) *Run-time states*

(b) *Task types*

(c) *Read accesses to NUMA nodes*

(d) *Write accesses to NUMA nodes*

(e) *Locality of memory accesses*

(f) *Communication matrix*

Figure 10.9: *Trace of* seidel *with random work-stealing and without work-pushing or deferred allocation*

the pixel. Initialization tasks, copying data from the shared matrix to streams are pink, while the main computational tasks are ocher. The phase with a low number of workers in the task execution state is located between the initialization phase and the phase composed of main tasks and is due to low parallelism at the beginning of the algorithm (cf. Section 6.1.1).

The locality of read accesses to main memory is illustrated in Figure 10.9c. For each worker in this view and each interval represented by one pixel on the timeline, Aftermath first calculates the set of tasks whose execution time overlaps with that interval. It then determines for each task the set of read accesses that took place during execution of the task and calculates how much data was accessed per node during the part of the task execution that overlaps with the pixel interval. This calculation is based on the assumption that accesses are not instantaneous, but are continuous operations that cover the whole task execution. When all tasks for the interval have been considered, the tool determines from which NUMA node most of the data has been read.

As an example for the calculation of the amount of data accesses by a task consider Figure 10.10. Assume that one pixel on the timeline represents 5.5 Mcycles and that there are three tasks t_1, t_2 and t_3 whose interval of execution overlaps with the pixel to be drawn. Aftermath assumes that the memory accesses of the tasks are carried out continuously during the entire execution of a task. Hence, for the interval of the pixel, it assumes that t_1 reads $\frac{0.5\,\text{Mcycles}}{3\,\text{Mcycles}} \cdot 128\,\text{KiB}$ from Node 5, t_2 reads 320 KiB from Node 2 and that t_3 does not perform any read access. As Node 2 represents the node which provides most of the data that is accessed in the interval, the pixel in this example would be drawn with the color associated to Node 2.

The untidy appearance of intervals of different colors in Figure 10.9c shows that there is no clear relationship between the nodes targeted by read accesses and the cores accessing the data. The same conclusion can be drawn for the locality of write accesses, illustrated in Figure 10.11b. However, the views presented in Figure 10.9c and 10.9d only show qualitative information, but do not reflect the locality of memory accesses quantitatively. For example, a task that reads only a bit more than half of its input data from a remote node and the rest of the data from the local node would be drawn using the color of the remote node, although a large fraction of memory accesses are local. A view that is more suited to investigate the quantitative aspects of data locality is the NUMA heatmap view, briefly introduced in Section 10.2.1, which uses different shades to represent the amount of memory that is accessed locally, ranging from blue (entirely local) to pink (entirely remote). The majority of the timeline component in this view shown in Figure 10.11c is pink, indicating that most of the memory accesses are in fact accesses to remote nodes. The absence of a clear relationship between the requesting nodes and the targeted nodes of memory accesses can be confirmed by looking at the communication incidence matrix for the measurement interval shown in Figure 10.9f. This graph captures for every pair of nodes how much data has been read and written by the cores of one node that is stored in the memory of the other node. Ideally, the matrix displays a sharp, diagonal pattern indicating that all memory accesses target the memory controller of the same node as the requesting cores. However, the matrix in the example indicates that all nodes communicate with all other nodes. The global statistics for the entire measurement interval and all workers indicate that only 5.36% are local accesses.

Figure 10.11 shows the same views for the *seidel* benchmark after activation of topology-aware work-stealing, work-pushing and deferred allocation. The views indicating the major nodes accessed in read mode (Figure 10.11a) and write mode (Figure 10.11b) show that there is a clear relationship between the cores accessing data and the targeted nodes. The timeline for all cores of the same node (core 0 to 7, 8 to 15 and so on) have the same color almost throughout the entire measurement interval. In addition, this color corresponds to the local node of the cores, indicating that at least half of all data is stored on the local node. This is confirmed by Figure 10.11c, which is almost entirely blue with only a few pink spots, which means that nearly 100% of data transfers are local. The communication matrix of Figure 10.11d with a sharp, diagonal pattern confirms this. Global statistics report that 97.94% of the data transfers found in the trace target local memory.

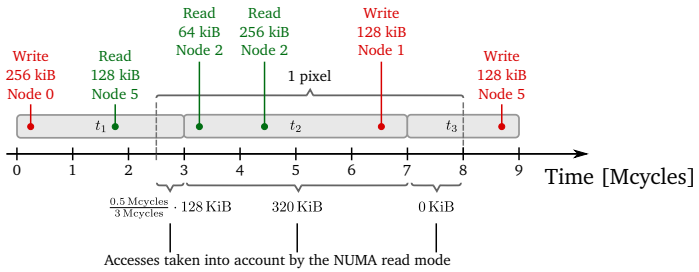
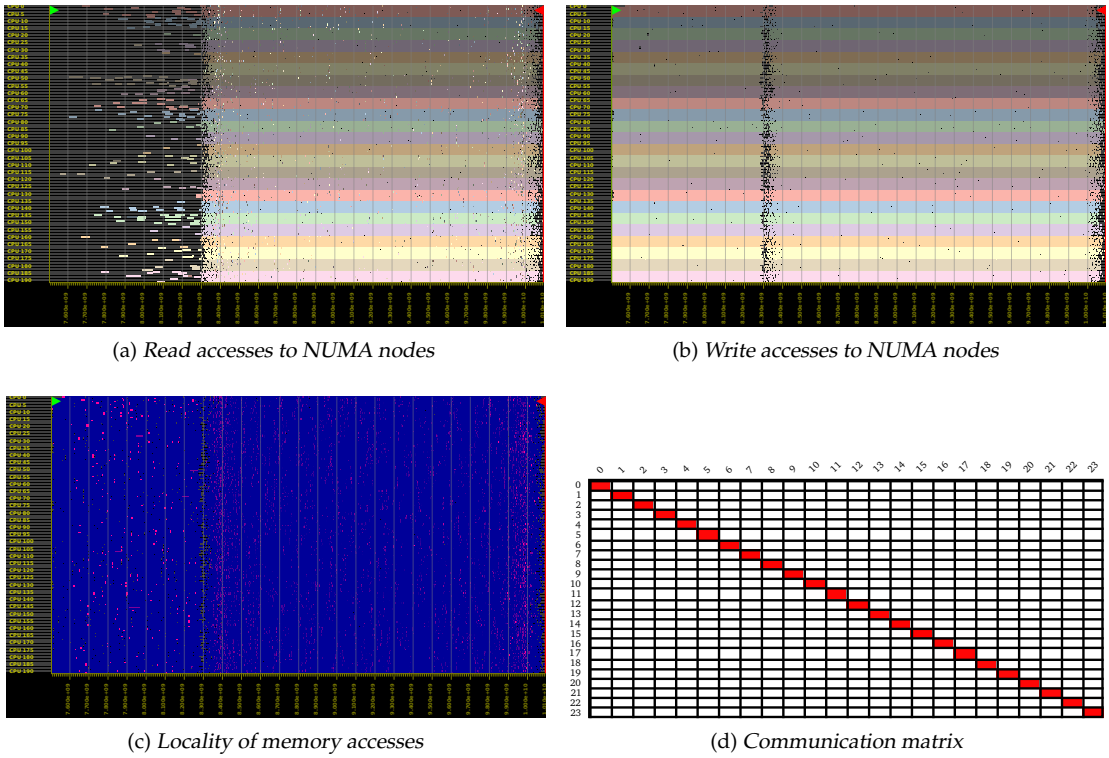


Figure 10.10: *Example of memory accesses*



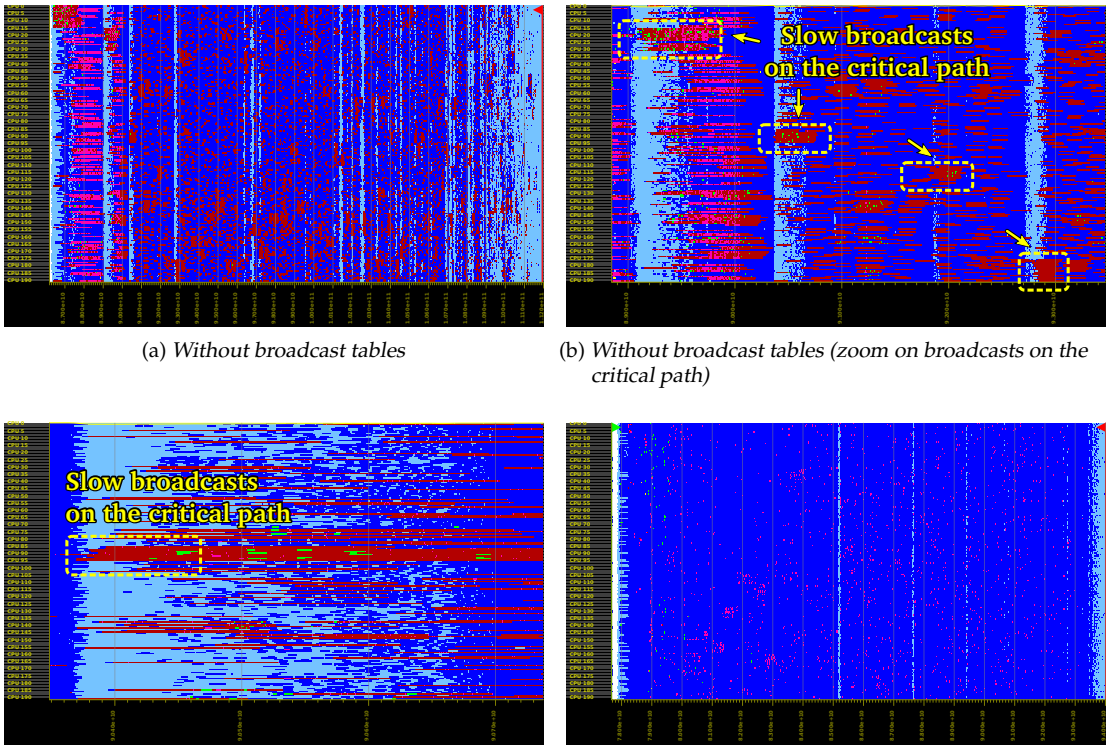
(a) *Read accesses to NUMA nodes*

(b) *Write accesses to NUMA nodes*

(c) *Locality of memory accesses*

(d) *Communication matrix*

Figure 10.11: *Different views for a trace of* seidel *with topology-aware work-stealing, work-pushing and deferred allocation*



(a) *Without broadcast tables*

(b) *Without broadcast tables (zoom on broadcasts on the critical path)*

(c) *Without broadcast tables (zoom on a smaller set of broadcasts on the critical path)*

(d) *With broadcast tables (single copy per block)*

Figure 10.12: *Worker states for each core during execution of* cholesky *without and with broadcast tables*

10.4.2 Broadcast tables

In the next example, we examine the impact of broadcast tables with a single copy presented in Section 9.2 on the *cholesky* benchmark, also executing on the 192-core SGI system. Figure 10.12a shows the activity of each worker during the measurement interval. This unzoomed view already reveals the numerous broadcast phases, indicated by the dark red parts as well as multiple phases during which most of the workers are idle, indicated by light blue. Aftermath reports in the statistical view that only 58% of the measurement interval is spent in task execution. The time spent on broadcasts represents 19% of the interval and 18% of the time workers are idle.

Another feature of Aftermath confirms this poor exploitation of the computing resources. By creating a derived counter, whose samples represent how many workers were in task execution state at the same time, the user is able to display the average parallelism over time. Figure 10.13a shows the graph for this derived counter. Ideally, all the 192 cores of the machine are busy during the entire measurement interval. The graph should thus display a steep increase to 192 at the beginning and should only drop below 192 towards the end of the measurement interval. However, the graph of Figure 10.13a has a different shape. The value of 192 is only reached a few times for a short period. Most of the time, the number of workers that execute tasks in parallel remains below 192 and often reaches values close to zero. Zooming further into the trace reveals that the broadcasts are responsible for phases with low parallelism: Figure 10.12b shows several broadcasts that are on the critical path. Figure 10.12c zooms on the first of these sets of broadcasts and clearly shows that there is only little activity besides the broadcasts themselves.

By using broadcast tables with a single copy per broadcast, the time spent on task execution during the measurement interval can be increased to 91%. Broadcasts only take less than 0.1% of the time in this configuration and idle time can be reduced to about 4.5% of the execution time.

(a) *Without broadcast tables* (b) *With broadcast tables (single copy per block)*

Figure 10.13: *Number of workers in task execution state during execution of* cholesky *without and with broadcast tables*

Figure 10.12d shows the activity of all cores during the measurement interval. There are still phases with few activity, but these can essentially be found at the beginning and at the end of the execution when the number of tasks ready for execution is low due to limited parallelism of the algorithm. Figure 10.13 summarizes how many workers are in task execution state in parallel. In contrast to the graph without broadcast tables, the number of workers executing tasks in parallel is near 192 almost during the whole measurement interval.

The examples above showed that Aftermath can be used to debug the performance of task-parallel applications and run-time systems. However, many of the steps involved in this analysis are repetitive and time-consuming due to the large number of potential causes for bottlenecks and the large number of available metrics (e.g., hardware performance counters). The next section presents a perspective to automate some of these tasks and to guide the user through the process of performance analysis.

10.5 A perspective for the automation of performance analysis

Currently, the user has to detect bottlenecks manually, i.e., by selecting appropriate views and by applying the filters that highlight a performance anomaly and emphasize its cause. For example, if the user suspects that low performance is due to a high cache miss rate of a certain type of tasks (e.g., tasks carrying out matrix multiplications), it is necessary to visualize the task duration, to limit the view to tasks of that type, to inspect the cache miss rate of slow and fast tasks and to develop the hypothesis of a correlation between task duration and the cache miss rate. The user must repeat these steps for every potential source of performance anomalies. Some of the steps in this process could be automated and the user could concentrate on the essential parts of performance analysis. In this section, we illustrate two scenarios for automated performance analysis. In the first scenario, the trace analysis tool analyzes the time spent in the different run-time states and informs the user about insufficient parallelism or high overhead, while the second scenario deals with the automatic detection of correlations between performance indicators. Future versions of Aftermath could integrate these mechanisms and thus guide the user through the process of performance analysis.

10.5.1 High-level analysis based on thresholds

Ideally, each core of the machine effectively contributes to the overall computation and spends nearly all of its time in the task execution state. Assume that there are n cores and an interval of a duration d. Let T_{exe} be a user-defined threshold indicating the targeted fraction of time in the task execution state; e.g., $T_{\text{exe}} = 0.95$ indicates that at least 95% of the time should be spent in that state. Let further $D_{e,i}$ be the overall duration a core i spends in task execution state within the interval. If the inequality $\sum_{i=1}^{n} D_{e,i} < T_{\text{exe}} \cdot n \cdot d$ holds, there is not enough parallelism to saturate the machine with running tasks.

The root cause for insufficient parallelism can be refined with further threshold-based analysis. For example, if more than a fraction T_{create} is spent in the task creation state, task creation overhead is likely to be too high or if a fraction of time T_{idle} is spent idling, then there might simply not be enough parallelism exposed by the application or there might be a load balancing problem.

The exact rules and thresholds are to be determined in future work. However, these simple examples show that basic high-level performance analysis can be carried out without intervention

of the programmer. This can be particularly convenient when comparing a larger number of implementations of an application or combinations of optimizations in the run-time. A trace for each of the configurations can be examined automatically and the results could be summarized in a report.

10.5.2 Correlating performance indicators with task durations

The analysis above does not cover performance anomalies that occur during the execution of tasks, such as the performance anomaly related to branch mispredictions of Section 10.3.2. Hardware performance counters can provide insight on what happens during task execution. Correlating per-task values for these counters with other performance indicators, e.g., correlating the branch misprediction rate with the execution time, can help develop a hypothesis to explain the anomaly. As shown in the examples, Aftermath already provides a generic interface for analysis of arbitrary performance counters, but it is up to the user to select the appropriate event type, to configure the run-time to capture hardware performance counter samples and to write the samples to the trace file. Manually determining the relevant event types by correlating the per-task values for each of the available counters with the execution time of tasks can be time-consuming due to the multitude of events that can be monitored on modern architectures. Hence, automating this process is highly desirable. We propose a method based on linear regression in order to determine automatically whether a specific hardware event should be considered for performance analysis and under which conditions it is relevant. This method can be outlined as follows:

1. Selection of a micro-architectural event that might be relevant (e.g., branch mispredictions, cache misses or accesses to remote memory).
2. Execution of the application that is to be analyzed and generation of a trace file with hardware performance counter samples for the selected event.
3. Calculation of per-task values for the duration and the increase of the number of the selected micro-architectural events by analyzing the hardware performance counter samples in the trace (e.g., the number of cache misses that occurred during execution of each task).
4. Analysis of the variation of the task execution time. This can be done by calculating the coefficient of variation of the execution time and by comparing it to a threshold. If the variation is sufficiently high, the analysis continues with the next step. If the variation is low, a performance anomaly that shows up in a subset of all tasks is unlikely and the analysis stops.
5. Correlation of the per-task values for the micro-architectural events with the task durations using linear regression. If the coefficient of determination exceeds a user-defined threshold, the correlation is considered to be relevant and the user is informed. Otherwise, the analysis restarts at the first step with a different micro-architectural event.

A performance anomaly does not necessarily concern all types of tasks of the application and all cores of the machine. For example, the performance anomaly related to branch mispredictions above only affected main computational tasks and the performance problem of the *seidel* benchmark described in Section 10.3.1 was only present on cores of nodes 1 to 7. Hence, steps 3, 4 and 5 should be repeated for user-defined subsets of task types and cores.

As steps 1 and 2 involve the configuration of the run-time system and generation of the trace and are thus specific to a particular implementation, the description below focuses on steps 3, 4 and 5.

Calculation of per-task hardware performance counter values

Hardware performance counter values are usually captured per core and first need to be associated with tasks before they can be correlated with other performance indicators. Assume that $v(c, i, t)$ returns the absolute value of a counter c for core i at a time t. Determining how much a value has changed during execution of a task is done by comparing v at the beginning T_s and the end T_e of the task. Figure 10.14 illustrates this for a counter c_{misp} indicating the number of branch mispredictions and a counter c_{cyc} counting the number of processor cycles. The number of

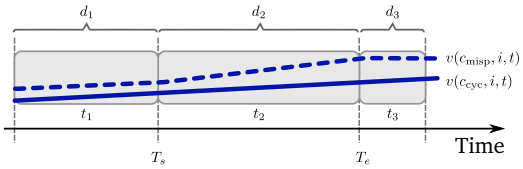


Figure 10.14: *Evolution of the values of hardware counters for branch mispredictions (c_{misp}) and cycles (c_{cyc}) on core i*

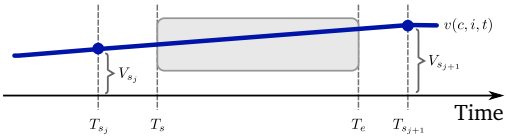


Figure 10.15: *Samples that do not exactly match the beginning and end of a task*

mispredictions generated by the task t_2, is $N_{\text{misp},t_2} = v(n_{\text{misp}}, i, T_e) - v(c_{\text{misp}}, i, T_s)$ and the number of cycles is $N_{\text{cyc},t_2} = v(c_{\text{cyc}}, i, T_e) - v(c_{\text{cyc}}, i, T_s)$. Hence, the branch misprediction rate R_{t_2} of t_2 is:

$$R_{t_2} = \frac{N_{\text{misp},t_2}}{N_{\text{cyc},t_2}} = \frac{v(n_{\text{misp}}, i, T_e) - v(c_{\text{misp}}, i, T_s)}{v(c_{\text{cyc}}, i, T_e) - v(c_{\text{cyc}}, i, T_s)}$$

The misprediction rates of t_1 and t_3, R_{t_1} and R_{t_3}, can be determined similarly.

The calculation of the misprediction rate above assumes that the number of branch mispredictions and the number of cycles at T_s and T_e are known. However, as hardware performance counter data is not available as a continuous function, but as a set of samples, these values might not be available directly in the trace file. Figure 10.15 shows the most frequent case, where the timestamps T_{s_j} and $T_{s_{j+1}}$ of the nearest samples do not exactly match T_s and T_e. However, assuming that the values for each counter are monotonically increasing, it is possible to approximate the required values through linear interpolation. For example, the value $v(c, i, T_s)$ in the figure can be approximated as:

$$v(c, i, T_s) \approx V_{s_j} + \left(T_s - T_{s_j}\right) \cdot \frac{V_{s_{j+1}} - V_{s_j}}{T_{s_{j+1}} - T_{s_j}}$$

where V_{s_j} is the value of c captured in the first sample and $V_{s_{j+1}}$ is the value of the second sample. The value for $v(c, i, T_e)$ the end of the task execution can be approximated similarly.

Correlation of performance indicators

We consider that a performance indicator is relevant for performance analysis if the variation of task durations is sufficiently high according to the coefficient of variation and if the duration correlates with the performance indicator according to a linear model. In Figure 10.14, the durations d_i differ substantially and depend on the misprediction rates R_{t_i} with $d_i \approx \alpha \cdot R_{t_i} + \beta$, α and β being constant. Such relationships can easily be detected automatically by performing linear regressions and by comparing the coefficients of determination to a threshold value.

As a simple example, consider the correlation between the misprediction rate and the duration that has been determined manually in Section 10.3.2 by plotting the branch misprediction rate over the timeline in heatmap mode. Figure 10.16 shows the duration of the main computation tasks in the benchmark as a function of the rate of branch mispredictions. Outliers with an execution time below 1 Mcycles have been filtered out. The dashed line in the graph represents the regression line and clearly indicates the linear correlation between the misprediction rate and the duration.

10.5.3 Status of the implementation

Aftermath currently provides only basic support for the techniques described above. High-level analysis based on thresholds is supported, but requires manual interpretation by the user: after the selection of an arbitrary interval from the timeline the tool shows the fraction of the time spent in the different states, but whether these exceed the thresholds must be detected by the user.

For the correlation of performance indicators with task durations, Aftermath is able to calculate per-task values using linear interpolation and supports the export of these values to a data file. However, the analysis of the variation and linear regression must be performed by external tools

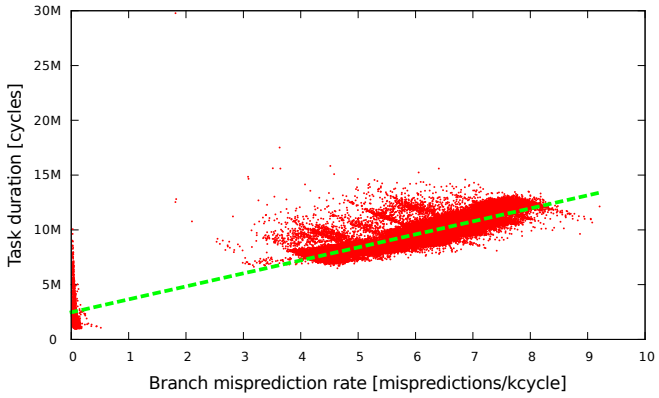


Figure 10.16: *Task duration as a function of the number of branch mispredictions per thousand cycles in* k-means

ran by the user. Also, the configuration of the run-time to sample specific micro-architectural events and automation of the execution of the application whose traces are to be analyzed has not yet been implemented.

10.6 Related Work

Visualization and analysis of trace files are common techniques, critical for performance analysis and debugging in high performance computing, for which many tools have been developed. As performance analysis is not the main topic of this thesis, providing a complete survey of this field is out of the scope of this document. For the following overview of related work, we have selected four representative tools, PARAPROF, PARAVER, VAMPIR, and VITE. We briefly explain why they do not fully meet our requirements for performance analysis in the OpenStream project and thus why we implemented Aftermath.

PARAVER [69] is a tool for interactive trace analysis, providing powerful filtering mechanisms for different graph types and independent views on trace data. Earlier versions of OpenStream included support for trace files in Paraver's native format. However, the tool's resource model focuses on computation and does not model memory-related resources and task communication patterns, which are essential to the characterization of performance anomalies on many-core NUMA architectures.

PARAPROF [17], a profile visualization tool of TAU [78], is a retargetable framework for writing trace analysis applications rather than a single tool for a specific type of trace files or performance analysis. It provides a set of extensible components for data sources, data management, analysis and visualization that can be used as a basis for new tools. The overlap between functionality of existing components of Paraprof and those required for data-flow analysis in the OpenStream project is small, such that the implementation cost for an OpenStream-specific tool using Paraprof would have been close to the cost for Aftermath.

VAMPIR [67] is a well-known commercial tool that has been used in high performance computing for almost two decades. It provides a rich user interface for interactive exploration and analysis of huge traces and has a highly elaborated filter interface. Multiple connected views with different granularity from cluster level to function calls are supported. But unlike Aftermath, the tool is optimized for analysis of massively parallel applications based on message passing. Neither NUMA resources nor tasks are modeled explicitly, making fine grained task-based and memory-related analysis impossible.

VITE [1] is a freely available tool for trace-based analysis of parallel programs focusing on fast rendering. However, the tool lacks support for NUMA topologies and analysis filters.

10.7 Summary and conclusions

After an analysis of the requirements for performance debugging of task-parallel applications and run-time systems we presented Aftermath, a tool that we have implemented for trace visualization and interactive trace analysis. We illustrated the strengths of Aftermath on several examples based on genuine situations encountered during the development of the OpenStream run-time system, the optimizations presented in this thesis and OpenStream benchmarks. Aftermath fulfills the requirements mentioned at the beginning, and has proven invaluable when simultaneously tracking the sources of performance anomalies in a task-parallel application and its supporting execution environment.

While initially designed for our specific needs for the analysis of OpenStream, only some of the graphs and metrics covered in this analysis are specific to OpenStream or to data-flow languages. Aftermath can thus also be used for performance debugging of other task-parallel languages and run-times. In future releases, we plan to add support for data dependences in OpenMP 4 similar to the support for input buffers for OpenStream.

At the end of the chapter, we showed how a user can be guided through the process of performance analysis by automating recurring tasks and by detecting certain types of performance anomalies fully automatically. We presented a first technique for high-level analysis based on thresholds that is able to detect insufficient parallelism and high overheads. The second technique correlates performance indicators using linear regression and detects which performance indicators are relevant for performance analysis. Aftermath provides basic support for a subset of the mechanisms employed by these techniques but requires the intervention of the user. We plan to fully integrate the above functionality in future releases. Other features developed in future work are the support for very large trace files that do not fit into main memory and the development of indexes that allow to calculate statistics without the need to traverse the entire trace.

11 Conclusion and perspectives

This chapter summarizes the work presented in this thesis and provides a conclusion on our findings. The chapter closes with a discussion of directions and opportunities for future research.

11.1 Summary of the thesis

Power dissipation has become a driving factor in the processor industry and led to a shift towards more energy efficiency designs integrating multiple cores on a single chip. Such multi-core designs are now used in a wide variety of machines ranging from embedded systems to high performance servers. The growth of the number of cores per machine is expected to carry on and many-core systems with dozens or even hundreds of cores are emerging. Task-parallel programming has become an increasingly popular approach to address productivity, scalability and portability in these environments, but leaves decisions for the efficient mapping of parallelism to an optimized run-time system. As many-core systems usually integrate multiple memory controllers with non-uniform memory access, efficient strategies must provide both efficient mappings of computations to cores and efficient mappings of data to memory controllers. As shown in Chapter 2, only little work has been done on transparent, portable and fully automatic on-line mapping mechanisms for task-parallel programs executing on many-core NUMA machines. The purpose of this thesis is to investigate how such mappings can be implemented based on information on point-to-point data dependences readily available in the run-time systems of modern task-parallel programming frameworks. We explored the main factors on performance and data locality and proposed fully automatic, transparent and portable run-time mechanisms for the mapping of data to memory controllers and the mapping of tasks to cores.

As a representative for a modern task-parallel language with point-to-point data dependences, we have chosen OpenStream, a data-flow extension to OpenMP based on the concepts of short-lived, fine-grained tasks and streams. Communication and synchronization between tasks in this model is achieved by accessing streams. The elements of streams become accessible to a task through views, which appear as finite arrays within the task body. The actual synchronization is carried out by the run-time by matching output views with input views on the same streams. Chapter 3 provided an overview of the concepts of OpenStream, its syntax and its execution model. The main formal construct for OpenStream programs used throughout this thesis are task graphs, capturing the producer-consumer relationships of an OpenStream application.

As efficient mechanisms for data-aware scheduling and memory allocation require a NUMA-aware run-time, we investigated in Chapter 4 how a run-time can determine the placement of data

on nodes efficiently and how it can be provided with fine-grained control over memory allocation and data placement. The focus on this chapter lay on the integration of these techniques with the default first-touch data placement mechanism of many operating systems and the avoidance of frequent time-consuming system calls. We introduced input buffers that store the stream elements read by a task and described methods for low-overhead placement and low-overhead determination of the placement of these buffers.

In Chapter 5, we introduced dynamic single assignment to gain control over the placement of *application* data and to be able to determine its placement. In this programming style, tasks communicate and synchronize exclusively using streams, such that all application data is stored in streams whose data is in turn stored in input buffers. Through the placement of input buffers the run-time has full control over data placement of the entire application and is able to determine the working set of each task reliably before a task executes.

The experimental setup for this thesis was given in Chapter 6. This included a description of a set of high performance scientific benchmarks implemented using dynamic single assignment, a description of the hardware and software environment and a description of the methodology used for the experiments. We introduced three baselines for the benchmarks, namely a dynamic single assignment baseline, a shared memory baseline implemented using token synchronization in which streams are only used to enforce data dependences and a sequential baseline that does not make use of OpenStream. We characterized the memory access behavior of the dynamic single assignment versions to classify the benchmarks into memory-bound applications and cache-bound applications.

Chapter 7 introduced two scheduling techniques that aim at improving the locality of accesses to main memory. The first of these techniques is work-pushing, which transfers a task to a core associated to the node that contains the data that will be accessed by the task. This mechanism heavily relies on information on the working set of tasks derived from accesses to stream elements in dynamic single assignment. The choice of the target core depends on a heuristic. We evaluated the *input only* heuristic, which transfers a task to a core of the node containing the majority of its input data, the *output only* which chooses a core on the node containing the majority of the tasks output buffers and the *weighted* heuristic, which takes into account the placement of both input and output buffers. As a complementary technique, we proposed a second scheduling mechanism called topology-aware work-stealing, which acts as an architecture-aware load balancing mechanism favoring steals from nearby cores in the memory hierarchy. The evaluation of the scheduling optimizations on the benchmarks of Chapter 6 showed that the locality of memory accesses could be improved significantly, resulting in speedups for memory-bound applications of up to $2.36\times$ on 192 cores grouped into 24 NUMA nodes compared to random work-stealing without work-pushing.

Work-pushing reacts to a given data placement resulting from local allocations and the interplay of initial task placement, task creations and work-stealing. To decouple data placement from the control program, from initial data placement and to react to work-stealing, we introduced deferred allocation in Chapter 8. The key concept of this mechanism is to delay the allocation of input buffers from task creation to the time the producers of a task become ready for execution. This allows the run-time to place buffers according to the node on which the producers are executed and prevents input buffers from being placed at task creation. The results are a reduced memory footprint, increased data locality and improved load balancing across memory controllers. The approach can be combined with the *input* only heuristic for work-pushing as well as topology-aware work-stealing and yields speedups of up to $3.57\times$ on 192 cores grouped into 24 NUMA nodes compared to a run-time with random work-stealing and using neither work-pushing nor deferred allocation.

Broadcasts, in which the data of a producer is read by multiple readers also benefits from work-pushing, topology-aware work-stealing and deferred allocation, but suffers from a high overhead in time and the memory footprint for copying data to all readers. These issues were addressed in Chapter 9, introducing broadcast tables. Broadcast tables allow the readers of a broadcast to share a single input buffer, keeping the amount of memory required for a broadcast

constant and avoiding the overhead on execution time for copying the data of the broadcast to all readers. The evaluation on the *cholesky* benchmark, which uses broadcasts extensively, showed that the memory footprint and the execution time could be decreased by more than an order of magnitude compared to the default broadcast mechanism without any of the optimizations of previous chapters. We showed that using broadcast tables, OpenStream is able to match the performance of state-of-the-art high performance implementations for Cholesky Factorization. To increase the locality of memory accesses during broadcasts, we added a mechanism to broadcast tables that creates on-demand copies on the NUMA nodes executing the readers of a broadcast. We showed that this strategy can increase the locality of memory accesses significantly, but due to the high cache hit rate of the *cholesky* benchmark the improvement of the locality does not result in improved performance for this benchmark.

In the last chapter, we presented Aftermath, our tool for trace-based performance analysis and visualization. We have used this tool extensively for performance debugging of the OpenStream run-time, in particular during the development of the optimizations presented in this thesis, as well as for performance debugging of the benchmarks presented in Chapter 6. Although we have originally implemented the tool specifically for OpenStream, many of its concepts apply to task-parallel applications and run-time systems in general. As a perspective for future work, we presented two approaches for automating recurring tasks and guiding the user through the process of performance analysis. The first of these approaches detects insufficient parallelism and high overheads, while the second approach identifies performance indicators that are relevant for performance analysis.

11.2 Contributions

The contributions of this thesis can be grouped into three categories. Contributions belonging to the first category are the key contributions of this thesis. The second category consists of technical concepts that form the basis for NUMA-aware scheduling and memory allocation. Practical contributions that are the result of the implementation of the concepts presented in this thesis or that helped during their development are summarized in the third category of contributions.

11.2.1 Key contributions

The key contributions of this thesis are mechanisms for efficient and portable, on-line placement of tasks and data for task-parallel applications executing on many-core systems. The proposed scheduling mechanisms are called work-pushing and topology-aware work-stealing, while the method for data placement is named deferred allocation. For broadcasts we proposed broadcast tables. All of the techniques were evaluated on a set of scientific benchmarks.

Work-pushing and topology-aware work-stealing (Chapter 7) We proposed work-pushing, a data-aware and NUMA-aware scheduling mechanism for task-parallel application that transfers tasks to cores associated to the nodes that contain the data that will be accessed by the tasks during their execution. The decision to which core a task is transferred is taken before the task is executed and is based on precise knowledge on the placement of the task's working set. This knowledge is derived from point-to-point data dependences readily available in the run-times of modern task-parallel languages and an efficient memory-management layer that caches information on data placement. Unlike existing approaches for NUMA-aware scheduling of tasks, work-pushing neither relies on a specific structure of the computations carried out by the task-parallel application, nor on specific types of data structures and does not require profiling. The approach thus supports a wide variety of applications. As work-pushing operates at execution time, it is able to react to dynamic changes of the program behavior. The approach is entirely transparent to the application and is carried out fully automatically.

We also introduced topology-aware work-stealing as a complementary technique to work-pushing for load balancing, stealing tasks from an incrementally widening neighborhood with respect to the memory hierarchy. This mechanism is similar to hierarchical work-stealing and relies

on a static description of the machine topology describing the levels of the memory hierarchy and the siblings of each core of each level.

Deferred allocation for the NUMA-aware management of task input buffers (Chapter 8) We proposed deferred allocation, a technique that delays the allocation of an input buffer of a task until the node of the producer writing to this buffer is known. Similar to the decisions for task scheduling above, deferred allocation relies on precise knowledge on data accesses of each task specified by point-to-point data dependences available in the run-time. Unlike other approaches for data placement, deferred allocation neither represents a static partitioning of the data nor migrates data. The principles of dynamic single assignment allow the run-time to choose a different node to store data each time it is passed from one task to another, allowing the run-time to react to dynamic changes in program behavior.

Broadcast tables for NUMA-aware broadcasts (Chapter 9) We proposed broadcast tables, a NUMA-aware technique to broadcast data produced by a single task to multiple readers. By sharing a single input buffer for all readers, this approach compensates the large amount of memory and the time spent on copying data for the distribution of broadcast data when using a naive implementation of dynamic single assignment. We also introduced broadcast tables with on-demand copies on nodes executing the readers of a broadcast, resulting in a significant increase of the data locality.

Experimental validation of the key contributions (Chapters 7, 8 and 9) We evaluated all of the techniques above on a set of high performance scientific benchmarks executing on two many-core NUMA platforms with 64 cores (8 NUMA nodes) and 192 cores (24 NUMA nodes), respectively. We demonstrated that work-pushing can increase the fraction of memory accesses targeting local memory above 90% and speed up execution by a factor of up to $2.36\times$ compared to the parallel baseline with random work-stealing. The speedup over the shared memory baseline with interleaved allocation over all memory controllers of the machine can be as high as $2.50\times$. Deferred allocation also results in significant improvements on data locality and can speed up execution by up to $3.57\times$ compared to random work-stealing without work-pushing. The speedup over the shared memory baseline can be as high as $4.17\times$.

The use of broadcast tables can reduce the memory footprint as well as the execution time of Cholesky Factorization by more than one order of magnitude. We showed that the OpenStream implementation matches the performance of two state-of-the-art parallel implementations of Cholesky Factorization provided by PLASMA and the INTEL MATH KERNEL LIBRARY.

11.2.2 Contributions that form the theoretical and technical basis for the key contributions

The following contributions from the theoretical and technical base required for the main contributions above. The identification of factors with an influence on data locality and performance form the basis for the development of techniques for data and task placement. From these findings we derived work-pushing, topology-aware work-stealing, deferred allocation and broadcast tables. The technical support for these techniques is provided by the second contribution, which consists in the concepts for NUMA-aware run-time systems.

Identification and analysis of factors with an influence on data locality, the memory footprint and performance (Chapters 4, 5, 7, 8 and 9) Throughout this thesis we identified and analyzed factors with an influence on data locality, on the memory footprint of an application and on performance. These are:

- the interaction between the run-time and the operating system during memory allocation and placement of pages on nodes,
- the structure of the task graph, the order of creation of tasks by the control program and the order in which tasks are executed,
- parallelism in the control program creating all tasks,
- initial placement of input buffers and
- steals by remote workers upon work-stealing.

We illustrated the influence of these factors on examples and quantified the impact of a subset of them on synthetic benchmarks.

Concepts for NUMA-aware run-times for task-parallel applications (Chapter 4) We analyzed first-touch allocation, the default mechanism of many operating systems for memory allocation and proposed a NUMA-aware memory management layer based on this mechanism for run-time systems of task-parallel languages. The key factor for efficiency in this solution are NUMA-aware memory pools, which reduce of the number of time-consuming system calls by reusing buffers that have been allocated from the operating system and by caching information on data placement in small metadata sections in front of these buffers. Fine-grained control over data placement is achieved by allocating a buffer of the requested size from a memory pool associated to the target node.

11.2.3 Practical contributions

The contributions below are the result of the practical evaluation of the scientific concepts, but do not represent purely scientific contributions on their own. However, as the utility of these contributions goes beyond the prototyping of scientific concepts, we present them separately.

Integration into the run-time of a state-of-the-art task-parallel language We implemented and integrated the techniques for NUMA-aware allocation and the determination of data placement of Chapter 4 as well as work-pushing, topology-aware work-stealing, deferred allocation and broadcast tables into the run-time of OpenStream. Our contributions to the run-time and the compiler have been integrated into the official release of OpenStream.

A set of high performance scientific applications implemented using OpenStream (Chapter 6) We implemented a set of high performance, scientific benchmarks using OpenStream, based on dynamic single assignment presented in Chapter 5. These are:

- *seidel*, a five point two-dimensional, iterative stencil operating on a two-dimensional matrix of double precision floating point elements,
- *jacobi-1d*, a three point one-dimensional iterative stencil operating on a vector double precision floating point elements,
- *jacobi-2d* a five point two-dimensional iterative stencil operating on a two-dimensional matrix of double precision floating point elements,
- *jacobi-3d*, a seven point three-dimensional iterative stencil operating on a three-dimensional matrix of double precision floating point elements,
- *blur-roberts*, a benchmark for image processing that implements a blur filter, followed by an edge detection using the Roberts Cross Operator,
- *bitonic*, a bitonic sorting network capable of sorting 2^N arbitrary 64-bit integers,
- *cholesky*, performing a Cholesky Factorization on a symmetric, positive definite matrix of double precision floating point values and
- *k-means* that partitions a set of n-dimensional points into k clusters using the K-means algorithm.

All of these benchmarks except *cholesky* were also implemented as a shared memory version based on token synchronization as well as a sequential version. Future versions of the OpenStream run-time can be evaluated using these benchmarks.

Aftermath, a tool for trace-based performance analysis and visualization We developed Aftermath, a tool for trace-based off-line performance analysis and visualization and used this tool extensively during the work for this thesis to understand the aspects with an influence on data locality and performance, to debug the performance of the run-time system, to develop the applications used for the evaluation of our concepts and to validate our concepts for data and task placement. Although the tool has been designed primarily for OpenStream programs and the OpenStream run-time, many of its concepts apply to task-parallel programs and run-times in general. As a valid trace file does not necessarily have to contain OpenStream-specific information, the tool can already be used for the analysis of other task-parallel applications and run-times.

11.3 Conclusions

From the analyses and the experimental results presented in this thesis, we draw the following conclusions regarding task and data placement.

- By exploiting point-to-point data dependences, readily available in the run-time systems of modern task-parallel languages it is possible to determine the working set of tasks and the placement of the working set on NUMA nodes reliably and efficiently.
- Run-time systems of task-parallel languages can be provided with fine-grained and efficient control over the placement of application data.
- Many static aspects (e.g., partitioning of data among tasks) and a multitude of dynamic events (e.g., the order of task creation, load balancing, interaction with the operating system) have a strong influence on the memory footprint, the data locality and the performance of task-parallel applications.
- It is possible to provide transparent, fully automatic and dynamic run-time techniques for data and task placement improving data locality and performance of memory-bound applications executing on many-core NUMA machines.
- The performance improvements of task and data placement are higher for larger machines with more cores and a higher number of memory controllers.

This leads us to more general conclusions on task-parallel applications and run-times, in particular:

- task parallel programming allows for the efficient exploitation of the computing resources and the memory bandwidth of many-core NUMA systems.
- data-flow programming can be beneficial for the efficient exploitation of shared memory many-core NUMA systems and outperform shared memory implementations.

In the remainder of this chapter, we discuss directions for future research, starting with ongoing work with preliminary results.

11.4 Future work and perspectives

The work on this thesis lead to a multitude of opportunities for future research. The following axes of research seem to be the most appealing and promising opportunities to us.

Control over the memory footprint through automatic throttling at task creation A side effect of step-by-step construction of a dynamic task graph by a parallel control program is that the maximum number of co-existing tasks can be substantially smaller compared to a sequential control program. For example, when using a sequential control program to construct a chain of n dependent tasks t_0, ..., t_{n-1}, the maximum number of tasks that might co-exist is n. Using a parallel control program in which each task t_i creates t_{i+2} as proposed in Section 5.5.5, the maximum number of co-existing tasks is only two. As the memory footprint grows linearly with the number of co-existing tasks, developing a control program that limits the number of co-existing tasks can be essential for large task graphs to prevent an application to exceed the amount of available main memory. However, in contrast to the trivial example of a chain of dependent tasks, the development of a control program for large task graphs with complex dependences might involve a trade-off between parallelism and the number of co-existing tasks. The development of an appropriate control program is a complex task and represents a burden on the programmer, such that automatic throttling of task creations is desirable. However, such a mechanism must take into account the structure of the task graph as naive throttling might result in deadlocks due to the fact that a task remains blocked if a subset of its consumers has not been created.

Reducing the memory footprint and increasing cache utilization As the memory footprint plays an important role for the performance of task-parallel applications, it is crucial to keep it as low as possible. The inout_reuse clause presented in Section 8.5 is one possibility to reduce the size of the working set of a task, resulting in better utilization of caches, and to reduce the

application's global memory footprint. However, to be effective, this clause requires a trade-off between data locality and the overhead for copying when migrating data from one node to another. Early results for the performance of applications using this clause showed that the overhead for copies is likely to cancel the performance improvement of increased locality. One way to mask this delay could be the specification of an alternate implementation for each task using the inout_reuse clause that relies on dynamic single assignment and to let the run-time choose between the two versions of a task. If the producer and a consumer execute on cores of the same node, the run-time would choose the version with the inout_reuse clause and in case of a migration the dynamic single assignment version reading from one node and writing to the target node would be chosen.

However, there are many other aspects of program execution have an influence on the footprint, which we did not explore in this thesis (e.g., the order of task execution). Future concepts for task-parallel run-times could include algorithms that estimate the footprint for multiple orders of task executions based on deeper inspection of the task graph and choose the order with the smallest memory footprint.

Software prefetching of input data Compulsory cache misses at the beginning of the execution of a task can have a serious impact on performance as they might stall the executing core during the execution of instructions on the critical path of the task. One possibility to reduce the number of these misses is to fetch the task's input data to a cache by executing appropriate prefetch instructions before it starts execution. As dynamic single assignment provides the run-time with the ability to determine the working set of a task before it is executed, prefetching could be carried out automatically by th run-time. However, aggressive prefetching might lead to cache pollution and evict data from the cache that is referenced by tasks executing on cores sharing the cache. In addition, the probability that the input data of a task is still present in the cache when the task starts execution highly depends on the order of task execution. The conditions under which software prefetching is effective are to be determined in future work.

Dynamic adjustment of the task granularity Task granularity, i.e., how much data is processed by each task, has a strong influence on the amount of available parallelism, on the overhead for inter-task communication as well as on the exploitation of the memory hierarchy. The optimal granularity can therefore vary between different applications and machines. Currently, the programmer has to choose and implement task granularity manually. If an application should provide multiple granularities, the programmer must develop a parametric model that allows the user to choose a granularity at execution time (e.g., as in Section 5.3.1). While this is already a challenging task for applications with regularly-structured parallelism, the implementation of such a model is even more complicated for applications with irregular parallelism and less regularly-structured task graphs. Ideally, the run-time is able to trade-off the overhead for inter-task communication, the exploitation of the memory hierarchy and parallelism by adjusting the granularity automatically. An approach towards this solution is to let the programmer specify the program with a very fine-grained granularity and to dynamically fuse tasks at execution time in the run-time.

Improving the performance of cache-bound applications The mechanisms for task and data placement presented in this thesis mainly improve performance by reducing the amount of accesses to remote memory. The performance of cache-bound applications remains unaffected by these optimizations. In future work, we would like to investigate techniques for cache-aware scheduling and cache-aware memory allocation for task-parallel applications that improve the performance of cache-bound applications. The development and analysis of such concepts is considerably more challenging due to frequent changes of the contents of each cache. Hence, a method for optimized task placement must be able to estimate whether the data needed by the task that becomes ready is still available in a cache or if the data must be fetched from main memory.

Validation on applications with irregular parallelism All of the applications studied in this thesis display regular patterns of parallelism. However, none out optimizations relies on a specific pattern of parallelism. To demonstrate that our approaches also apply to applications with irregular patterns for parallelism additional benchmarks are needed.

Integration into other task-parallel run-times To emphasize the portability of our solutions it would be interesting to integrate them into run-times of other task-parallel languages. The recently appeared OpenMP 4 standard adds point-to-point data dependences on memory regions to OpenMP tasks. Run-times implementing this standard are thus excellent candidates for the implementation of our concepts.

Comparison with shared memory implementations using static placement The baseline for shared memory for the experimental evaluation uses interleaved allocation across all nodes of the machine. While this solution yields an excellent distribution of requests to main memory and thus high memory bandwidth, it only achieves poor data locality. For a totally fair comparison between dynamic single assignment and shared memory versions, hand-tuned shared memory implementations with manual data placement are needed.

Hybrid static / dynamic approaches for optimization The solutions presented in this thesis all operate dynamically at execution time. However, the work in this thesis also showed that many static aspects have an influence on performance. As mentioned above, the granularity of the data processed by tasks has an influence on the available parallelism, cache hit rates and the overhead for task creation. Hybrid approaches combining optimizations at compile time, such as the automatic partitioning of regular data structures among tasks, with dynamic optimizations at execution time could improve performance significantly and reduce the burden of the programmer to take decisions for static optimizations manually.

Bibliography

[1] http://vite.gforge.inria.fr/. Accessed 10/2013.

[2] Intel Math Kernel Library. https://software.intel.com/en-us/intel-mkl, accessed 01/2015.

[3] Intel Xeon Processor E5-4600 Series. http://download.intel.com/support/processors/xeon/sb/xeon_E5-4600.pdf, accessed 02/2015.

[4] MaMI: Marcel memory interface. http://runtime.bordeaux.inria.fr/MaMI/, accessed 10/2014.

[5] METIS. http://pdos.csail.mit.edu/metis/, accessed 11/2014.

[6] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html, accessed 11/2014.

[7] Omni Compiler Project. http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/, accessed 11/2014.

[8] Parallelism in the Intel Math Kernel Library. https://software.intel.com/en-us/articles/parallelism-in-the-intel-math-kernel-library, accessed 02/2015.

[9] PARSEC. http://parsec.cs.princeton.edu/, accessed 11/2014.

[10] The CSU Face Identification Evaluation System, Version 5.1. http://www.cs.colostate.edu/evalfacerec/algorithms5.php, accessed 11/2014.

[11] AMD. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors*, 2013. http://support.amd.com/us/Processor_TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf, accessed 09/2014.

[12] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd edition, 1999.

[13] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[14] Vishal Aslot, Max J. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, WOMPAT '01, pages 1–10, London, UK, UK, 2001. Springer-Verlag.

[15] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, and V. Venkatakrishnan. The NAS parallel benchmarks. Technical report, 1994.

[16] K. E. Batcher. Sorting networks and their applications. In *Proc. of the April 30–May 2, 1968, Spring joint Computer Conf.*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[17] Robert Bell, Allen D Malony, and Sameer Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In *Euro-Par 2003 Par. Processing*, pages 17–26. Springer, 2003.

[18] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[19] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C.A. Nelson, and C.D. Offner. Extending openmp for numa machines. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 48–48, Nov 2000.

[20] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). 28:135–151, 2001.

[21] Susan Blackford and Jack J. Dongarra. Installation guide for LAPACK. Technical Report 41, LAPACK Working Note, June 1999. Originally released March 1992.

[22] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. of the 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[23] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*, May 2008.

[24] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.1*, 2011.

[25] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.0*, July 2013.

[26] Shekhar Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[27] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[28] François Broquedis, Thierry Gautier, and Vincent Danjean. LIBKOMP, an efficient OpenMP runtime system for both fork-join and data flow paradigms. In *Proc. of the 8th Intl. Conf. on OpenMP in a Heterogeneous World*, IWOMP'12, pages 102–115, Berlin, Heidelberg, 2012. Springer-Verlag.

[29] François Broquedis, Olivier Aumage, Brice Goglin, Samuel Thibault, P-A Wacrenier, and Raymond Namyst. Structuring the execution of openmp applications for multicore architectures. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[30] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. Hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italie, February 2010.

[31] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. ForestGOMP: An efficient OpenMP environment for NUMA architectures. *Intl. J. of Parallel Programming*, 38(5):418–439, 2010.

[32] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic task and data placement over numa architectures: An openmp runtime perspective. In Matthias S. Müller, Bronis R. de Supinski, and Barbara M. Chapman, editors, *Evolving OpenMP in an Age of Extreme Parallelism*, volume 5568 of *Lecture Notes in Computer Science*, pages 79–92. Springer Berlin Heidelberg, 2009.

[33] Zoran Budimlič, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Taşirlar. Concurrent collections. *Scientific Programming*, 18:203—217, 2010.

[34] M. Castro, L.G. Fernandes, C. Pousa, J. Mehaut, and M.S. de Aguiar. Numa-ictm: A parallel version of ictm exploiting memory placement strategies for numa machines. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, May 2009.

[35] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[36] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proc. of the 20th Annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[37] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proc. of the 17th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[38] Quan Chen, Minyi Guo, and Haibing Guan. Laws: Locality-aware work-stealing for multi-socket multi-core architectures. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 3–12, New York, NY, USA, 2014. ACM.

[39] Quan Chen, Minyi Guo, and Zhiyi Huang. CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *Proc. of the 26th ACM Intl. Conf. on Supercomputing*, ICS '12, pages 163–172, New York, NY, USA, 2012. ACM.

[40] The GCC Developer Community. *The GNU OpenMP Implementation*. Boston, Massachusetts, 2006. `http://gcc.gnu.org/onlinedocs/gcc-4.2.0/libgomp.pdf`, accessed 10/2014.

[41] Jonathan Corbet. AutoNUMA: the other approach to NUMA scheduling. `http://lwn.net/Articles/488709/`, accessed 11/2014.

[42] Jonathan Corbet. Transparent huge pages in 2.6.38, 2011. `http://lwn.net/Articles/423584/`, accessed 09/2014.

[43] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: Recording Microprocessor History. *Commun. ACM*, 55(4):55–63, April 2012.

[44] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proc. of the 18th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, New York, NY, USA, 2013. ACM.

[45] Andi Drebes, Karine Heydemann, Antoniu Pop, Albert Cohen, and Nathalie Drach. Automatic detection of performance anomalies in task-parallel programs. *CoRR*, abs/1405.2916, 2014.

[46] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. *ACM Trans. Archit. Code Optim.*, 11(3):30:1–30:25, August 2014.

[47] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach-Temam. Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. In *MULTIPROG '14*, 2014.

[48] Paul J. Drongowski. *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*. Advanced Micro Devices, November 2007.

[49] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the ACM SIGPLAN 1998 Conf. on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[50] B. Goglin and N. Furmento. Enabling high-performance memory migration for multithreaded applications on linux. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–9, May 2009.

[51] Compaq Information Technologies Group. *Compaq Fortran Parallel Processing Manual for Tru64 UNIX Systems*, 2002.

[52] Mark D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, December 1990.

[53] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[54] Intel Corporation. Threading Building Blocks. `http://gcc.gnu.org/onlinedocs/gcc-4.9.0/gccint.pdf`, accessed 10/2014.

[55] International Organization for Standardization. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[56] Ravishankar Iyer, Hujun Wang, and Laxmi Narayan Bhuyan. Design and analysis of static memory management policies for cc-numa multiprocessors. *J. Syst. Archit.*, 48(1-3):59–80, September 2002.

[57] H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, NAS System Division, NASA Ames Research Center, 1999.

[58] Andreas Kleen. A NUMA API for Linux, April 2005.

[59] Martin Kong, Antoniu Pop, Louis-Noël Pouchet, R. Govindarajan, Albert Cohen, and P. Sadayappan. Compiler/runtime framework for dynamic dataflow parallelization of tiled programs. *ACM Trans. Archit. Code Optim.*, 11(4):61:1–61:30, January 2015.

[60] Jakub Kurzak, Piotr Luszczek, Asim YarKhan, Mathieu Faverge, Julien Langou, Henricus Bouwmeester, and Jack Dongarra. Multithreading in the plasma library. In Sanguthevar Rajasekaran, Lance Fiondella, Mohamed Ahmed, and Reda A. Ammar, editors, *Multicore Computing: Algorithms, Architectures, and Applications*, Chapman & Hall/CRC Computer and Information Science, pages 119–142, Boca Raton, Florida, USA, December 2013. CRC Press.

[61] Henrik Löf and Sverker Holmgren. Affinity-on-next-touch: Increasing the performance of an industrial pde solver on a cc-numa system. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 387–392, New York, NY, USA, 2005. ACM.

[62] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. Feedback-directed page placement for ccnuma via hardware-generated memory traces. *J. Parallel Distrib. Comput.*, 70(12):1204–1219, December 2010.

[63] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[64] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 162–, New York, NY, USA, 2004. ACM.

[65] Jason Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–180, 2003. `ftp://gcc.gnu.org/pub/gcc/summit/2003/GENERIC%20and%20GIMPLE.pdf`, accessed 09/2014.

[66] Eduardo Henrique Molina da Cruz, Marco Antonio Zanata Alves, Alexandre Carissimi, Philippe Olivier Alexandre Navaux, Christiane Pousa Ribeiro, and Jean-Francois Mehaut. Using memory access traces to map threads and data on hierarchical multi-core platforms. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing*

Workshops and PhD Forum, IPDPSW '11, pages 551–558, Washington, DC, USA, 2011. IEEE Computer Society.

[67] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing scalable applications with vampir, vampirserver and vampirtrace. In *Proc. of ParCo '07*, volume 15 of *Advances in Par. Comp.*, pages 637–644. IOS Press, 2008.

[68] D. S. Nikolopoulos, E. Artiaga, E. Ayguadé, and J. Labarta. Exploiting memory affinity in openmp through schedule reuse. *SIGARCH Comput. Archit. News*, 29(5):49–55, December 2001.

[69] Vincent Pillet, Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. Technical report, In WoTUG-18, 1995.

[70] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical task-based programming with StarSs. *Intl. J. on High Performance Computing Architecture*, 23(3):284–299, 2009.

[71] Antoniu Pop. *Leveraging streaming for deterministic parallelization: an integrated language, compiler and runtime approach*. Theses, École Nationale Supérieure des Mines de Paris, September 2011.

[72] Antoniu Pop and Albert Cohen. A stream-computing extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 5–14, New York, NY, USA, 2011. ACM.

[73] Antoniu Pop and Albert Cohen. Control-Driven Data Flow. Rapport de recherche RR-8015, INRIA, July 2012.

[74] Antoniu Pop and Albert Cohen. Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs. Rapport de recherche RR-8001, INRIA, June 2012.

[75] Louis-Noël Pouchet. PolyBench/C. `http://web.cse.ohio-state.edu/~pouchet/software/polybench/`, accessed 08/2014.

[76] Christiane Pousa Ribeiro and Jean-François Méhaut. Minas: Memory Affinity Management Framework. Research Report RR-7051, 2009.

[77] Christiane Pousa Ribeiro, Jean-Francois Mehaut, Alexandre Carissimi, Marcio Castro, and Luiz Gustavo Fernandes. Memory affinity for hierarchical shared memory multiprocessors. In *Proceedings of the 2009 21st International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '09, pages 59–66, Washington, DC, USA, 2009. IEEE Computer Society.

[78] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[79] Richard M. Stallman and the GCC Developer Community. *GNU Compiler Collection Internals*. Boston, Massachusetts, 2010. `http://gcc.gnu.org/onlinedocs/gcc-4.7.0/gccint.pdf`, accessed 10/2014.

[80] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with PAPI-C. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173. Springer Berlin Heidelberg, 2010.

[81] The Cairo Graphics Team. Cairo graphics. `http://www.cairographics.org/`. accessed 10/2013.

[82] The GTK+ Team. The GTK+ project. `http://www.gtk.org/`. accessed 10/2013.

[83] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building portable thread schedulers for hierarchical multiprocessors: The BubbleSched framework. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 42–51. Springer Berlin Heidelberg, 2007.

[84] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 1–116, London, UK, UK, 1995. Springer-Verlag.

[85] Asim YarKhan, Jakub Kurzak, and Jack Dongarra. *QUARK Users' Guide – QUeueing And Runtime for Kernels*, 2011. `http://ash2.icl.utk.edu/sites/ash2.icl.utk.edu/files/publications/2011/icl-utk-454-2011.pdf`, accessed 10/2014.

[86] Richard M. Yoo, Christopher J. Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. Locality-aware task management for unstructured parallelism: A quantitative limit study. In *Proc. of the 25th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, SPAA '13, pages 315–325, New York, NY, USA, 2013. ACM.

[87] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? *SIGPLAN Not.*, 45:203–212, January 2010.

Index

Symbols

A

B

C

D

E

F

G

H

N

O

P

Q

R

S

T

U

V

W

X

Z

A Personal Publications

[1] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. *ACM Transactions on Architecture and Code Optimization*, 11(3):30:1–30:25, August 2014.

[2] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach-Temam. Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. In *MULTIPROG '14*, 2014.

[3] Andi Drebes, Karine Heydemann, Antoniu Pop, Albert Cohen, and Nathalie Drach. Automatic detection of performance anomalies in task-parallel programs. *CoRR*, abs/1405.2916, 2014.

B About this document

This document has been created exclusively using free software, mainly provided by the DEBIAN GNU/LINUX distribution. The purpose of this section is to provide a list of these software packages.

B.1 Typesetting and editing

Typesetting was done using LaTeX (pdfTeX 3.1415926-2.4-1.40.13 (TeX Live 2012/Debian)) with the following packages:

- algorithm2e
- amsmath
- amssymb
- babel
- caption
- colortbl
- enumitem
- fancyhdr
- float
- fontenc
- geometry
- graphicx
- hyperref
- idxlayout
- ifthen
- inputenc
- listings
- makeidx
- microtype
- multicol
- multirow
- palatino
- siunitx
- subfig
- tikz
- titlesec
- xstring
- xcolor

Bibliographical information has been processed by BibTeX and the index has been generated using MAKEINDEX.

LaTeX files have been edited using the EMACS editor. For basic spell checking we have used the FLYSPELL mode using the HUNSPELL spell checker and for more advanced grammar checks we have used the LANGUAGETOOL proofreading software.

A helpful tool for continuous proofreading is DIFFLATEX, which is able to highlight differences between two versions of a LaTeX file. All intermediate versions have been stored in a GIT repository.

Grayscale versions to check that figures remain readable when printed in black and white have been generated using GHOSTSCRIPT.

For the visualization of the PDF files generated by PDFLATEX, we have used the OKULAR and EVINCE document viewers. Occasionally, we have used the SYNCTEX tool, generating a file with metadata that allows the user to click on a word in the document viewer to open an editor at the corresponding line of the source file.

B.2 Figures and graphs

Most of the figures have been created with INKSCAPE. The screenshot of the main user interface of Aftermath (Figure 10.3) has been created using XFCE4-SCREENSHOOTER. All other illustrations of Aftermath were directly created using the PDF export function of Aftermath. Annotations on these figures and small changes to improve the readability (e.g., thicker lines for graphs) have been added using INKSCAPE.

The data for bar graphs and graphs showing lines has been extracted from the log files of the experiments using custom PYTHON scripts. The actual graphs have been generated by passing this data to functions of the MATPLOTLIB package. Statistical functions, e.g., to calculate means, medians and the standard deviation have been provided by the NUMPY package.

The graph showing the linear regression on the frequency of branch mispredictions and the task duration in Figure 10.16, as well as the linear regression itself have been created using GNUPLOT.

The build process for the PDF file is based on a Makefile using extensions of GNU MAKE.