DISS. ETH NO. 23823

Machine-aware memory allocation and synchronization

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

STEFAN KAESTLE

Diplom-Informatiker, Karlsruher Institut für Technologie (KIT)

born on 25.04.1986

citizen of Germany

accepted on the recommendation of

Prof. Dr. Timothy Roscoe (ETH Zurich), examiner
Prof. Dr. Gustavo Alonso (ETH Zurich), co-examiner
Dr. Timothy Harris (Oracle Research Labs, Cambridge, UK), co-examiner

2016

Abstract

The performance of parallel programs on multicores critically depends on hardware-conscious implementations that considers low-level characteristics of the machine. Among others, this applies to the implementation of memory allocation and synchronization primitives.

However, as a result of hardware being increasingly complex, it is hard to understand the implications of their characteristics on software's performance. Worse, hardware diversity makes it non-trivial to apply optimizations that are promising for one piece of hardware to another. Finally, fast paced changes in hardware-design require programmers to adopt their programs frequently to keep up with trends dictated by hardware making manual tuning an unattractive solution.

One possible solution to the problem is to provide systems that model hardware characteristics and automatically configure themselves accordingly. For *memory allocation*, we investigate this with Shoal, a smart machine-aware memory allocator that applies data distribution, partitioning and replication automatically when allocating memory and further uses advanced hardware features such as DMA engines and large/huge pages where applicable. The choice depends on the application's memory access patterns, which we propose to extract automatically from high-level parallel programs. We implement and evaluate this approach for Green-Marl, a domain specific language in the field of graph analytics.

The second part of this thesis investigates *synchronization* implemented on top of message-passing-based group communication primitives. Optimizing them gains importance even within single machines as a result of an increase in software-parallelism and resulting scalability challenges of shared-memory implementations. We propose the use highly tuned multicast trees as a building block for higher-level applications. If made machine-aware, high level protocols such as barriers or agreement protocols built on top of these trees then directly benefit from optimizations applied on the lower level. As a result, they are simpler to program while being competitive or better than hand-tuned state-of-the art implementations. Further, it relieves programmers from having to understand intricacies of multicore hardware. The challenge with optimizing multicasts is to select the tree topology and the order in which messages are sent from each core. Our approach to solving

this problem is to generate the tree from a small number of intuitive heuristics while simulating it's execution. Simulation requires a detailed model of a machine's message-passing performance. We found that available hardware information is too coarse grained and propose to acquire the model from a small number of carefully crafted micro-benchmarks automatically.

With both our memory allocator and our synchronization framework we show two examples of how to built a system that achieves good performance on a wide range of multicore machines without programmers having to manually tune their implementations and without them having to understand the intricate details of modern multicore hardware.

Zusammenfassung

Die Leistung paralleler Programme kann leiden wenn Speicherallozierung und Synchronisierung nicht optimal konfiguriert sind. Leider machen es zunehmend komplexe und vielfältige Hardware schwierig und aufwändig die Auswirkungen ihrer Charakteristiken auf die Leistung von Programmen zu verstehen. Da eine gute Konfiguration abhängig von der Maschine ist können Optimierungen einer Maschine nicht direkt auf eine andere übertragen werden. Die daraus folgende Notwendigkeit manueller Optimierungen für jede einzelne Maschine sind wenig attraktiv.

In dieser Arbeit schlagen wir automatische Speicherallozierung vor, die intelligent Daten verteilt, partitioniert und repliziert. Darüber hinaus werden erweiterte Beschleuniger wie DMA Hardware und größere Speicherseiten automatisch verwendet. Unser System basiert auf der Verwendung von Informationen über die Struktur von Speicherzugriffen. Diese können von Programmierern angeben werden oder im Idealfall automatisch aus parallelen Programmiersprachen höherer Ebenen extrahiert werden. Wir stellen ein System vor das dieses für ein Graphanalyse-Framework durchführt.

Im zweiten Teil dieser Dissertation untersuchen wir Primitive zur Synchronisierung basierend auf Gruppenkommunikation. Wegen der zunehmenden Parallalität von Software und der daraus resultierenden Schwierigkeiten bei der Skalierung von shared-memory Programmierung erlangt dies immer mehr an Bedeutung. Unser Lösungsansatz für dieses Problem sind hochoptimierte Bäume für Gruppenkommunikation. Wenn diese so konstruiert werden, dass sie die Eigenheiten von Rechnern verstehen und berücksichtigen, können syntaktisch höherwertige Programme mit vergleichbarer oder besser Leistung auf diesen aufgebaut werden. Das ist möglich ohne das Programmierer manuell Optimierungen für die Eigenheiten einzelner Maschinen durchführen müssen, da diese automatisch von den Optimierungen auf niedrigeren Ebenen profitieren. Als Beispiele zeigen wir Barrieren und Protokolle zur gegenseitigen Absprache.

Die Herausforderung bei der Verwendung von Kommunikation auf der Grundlage von Bäumen ist die Auswahl einer geeigneten Baumstruktur und der Reihenfolge der Sendeoperationen in jedem Knote des Baumes. Wir zeigen wie wenige speziell entworfene Benchmark-Programme verwendet werden können um ein Modell des Rechners automatisch zu generieren. Darauf

aufbauend kann die Baumstruktur und Sendereihenfolge dann offline mit einer Event-basierten Simulation bestimmt werden. Die Simulation sagt den Systemstatus auf Grundlage der Benchmarks voraus und implementiert Heuristiken zur Konfiguration der Topologie.

Unsere Optimierungen von Speicherallozierung und Synchronisierung zeigt wie eine gute Leistung von parallelen Programmen erzielt werden kann ohne dass Programmierer die Eigenheiten von moderner Hardware verstehen und von Hand Anpassungen vornehmen müssen. Wie wir in dieser Dissertation zeigen funktioniert das auf einer breiten Palette von Rechner.

Acknowledgments

I would like to thank all members of the Systems Group here at ETH Zurich. Most of all, Mothy for giving me the opportunity to pursue my research and his patience on the way there. Special thanks also go to Reto, Roni, Sabela, Kornilios and Moritz for fruitful and fun collaborations. Further, I thank Besa, Ingo, Jana, Darko, Pravin, Gerd, Simon and Lukas for generally making the time at ETH such a wonderfully joyful experience and making the last few years such a memorable phase of my life. Finally, I thank Gustavo for always having an open ear for my concerns and for being my co-examiner.

My gratitude also goes to Tim for inspiration and advice not only during my internship at Oracle Labs Cambridge, but also afterwards and for agreeing to be on the my PhD committee.

Ein besonderer Dank auch an meine Familie, inbesondere meine Eltern für ihre unendliche Geduld und Unterstützung über all die vielen Jahre.

Finally, very special thanks go to Molináa for always supporting me, being by my side and her patience during difficult times.

Contents

1 Introduction

Multicore background

In this thesis, we investigate the programmability of multicore machines in the light of increasing hardware complexity and diversity. We focus on memory allocation and synchronization. As we will show, when configured suboptimally both are causing penalties for the performance of parallel programs. The miss-configuration is frequently (*i*) a result of programmers not understanding the complexities of modern hardware, (*ii*) them not keeping up with changes of hardware characteristics driven by recent development in this area or (*iii*) simply a lack of resources to tune algorithms to a wide range of such machines.

Modern multicore machines are increasingly complex and diverse hardware platforms. Among others, this is especially true for the memory hierarchy consisting of several memory controllers placed across the machine and an increasing number of processors that are often connected by hierarchical interconnect network. Figure 1.1 shows an example of such a machine with eight memory controllers (one per socket) and a enhanced twisted-ladder interconnect.

Processor cores are typically grouped in sockets sharing NUMA nodes and often the last-level cache. Communication between cores on the same node are normally significantly faster than communication across nodes. Since the interconnect network exhibits similar properties as traditional networks, multicore machines have been treated as distributed systems [BPS+09].

Memory access latency and bandwidth then depend on which core accesses which memory location [BPS+09, BWCC+08] and further the interconnect may suffer congestion when access to memory controllers is unbalanced [DFF+13]. We will discuss multicore machines in greater detail in Chapter 2.

In order to mitigate high access cost, processors access main memory

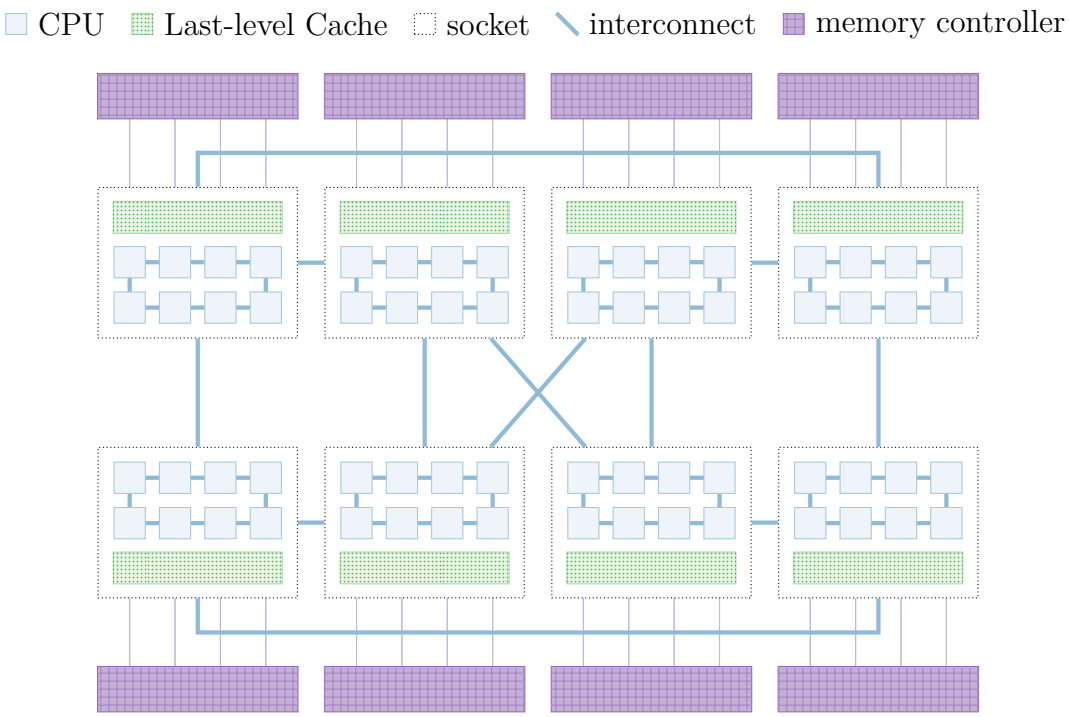


Figure 1.1: Architecture of a modern multicore machine.

through a multi-level cache hierarchy. Caches replicate data stored in main memory. If data has spacial locality, the smaller caches can help to avoid unnecessary expensive memory accesses and reduce traffic on the interconnect that would otherwise be needed to fetch data from remote memory controllers.

Replication, as with caching, introduces consistency issues: data is now stored several times in different locations. Care has to be taken when updating replicated state to ensure that updates to it are visible to processors in a consistent manner. On multicores, the hardware takes care of providing this consistency by means of a built-in cache-coherence protocol.

Problem statement

Multicore machines pose significant challenges to programmers when writing software due to their increasing complexity and diversity. Where to allocate and how to access memory as well as synchronization between threads are two examples: if configured suboptimally, the performance of parallel programs can suffer.

We illustrate this with an example: the popular Streamcluster benchmark. We evaluate the memory allocation aspect in Detail in Section 4.6.1 and the synchronization in Section 5.5.7).

Streamcluster allocates memory using a low-level `malloc` call, which pro-

vides no guarantees about where memory is allocated or other details such as the page size to use. Linux, for example, currently employs a first-touch memory allocation strategy. In that case memory is allocated on the NUMA closest to thread requesting it. If memory is initialized sequentially from just one thread, all memory ends up on the same NUMA node causing contention on the interconnect and an imbalance in memory controller usage.

For synchronization, Streamcluster relies on glibc's pthread barriers. The implementation relies on a single shared counter [gli16], which is atomically incremented until the required number of threads is reached. pthread barriers are unaware of hardware specific performance characteristics and hence fail to achieve optimal performance on current multicores. This is due to collisions when accessing the cache-line storing the counter exclusively causing many unnecessary cache-line transfers.

Table 1.1 shows a preview of the performance of a parallel benchmark program (PARSEC's Streamcluster) in a native configuration compared to our machine-aware memory management and synchronization optimizations that we are going to introduce throughout this thesis.

		machine	synchronization	
		(cf. Section A)	native/pthreads	optimized
memory access	native/`malloc`	I SB 4x8x2	236.5	130.7
	optimized		66.4	35.8
	native/`malloc`	A IL 4x4x2	215.6	197.9
	optimized		51.8	40.3

Table 1.1: Execution time of Streamcluster [seconds] using Streamcluster's default configuration ("native") compared to what can be achieved with machine-aware optimizations ("optimized"). We list machine configurations in Section A.

This example shows, that the performance of parallel programs critically depends on the implementation of memory allocation and synchronization implementations:

Memory allocation Firstly, due to NUMA characteristics of multicores, care has to be taken when allocating and accessing memory. Programmers now have the choice of where to allocate memory and how to distribute a program's state on NUMA nodes. The cost of accessing memory depends on where memory is allocated. The goal is to avoid expensive remote memory accesses in favor of local ones to reduce access cost. Moreover, memory accesses should be balanced across memory controllers, so that all of them can be saturated to achieve the highest accumulated bandwidth when accessing memory in parallel. Balancing memory accesses also helps to avoid contention on individual interconnect links for cross-nodes memory transfers.

Replication can be used, or memory can be partitioned, but the configuration depends on the NUMA hierarchy of the machine and application access patterns. Additionally, additional hardware features such as DMA engines might be available on some platforms, but the available is machine-dependent and hence often not considered as viable optimization.

Synchronization Secondly, as a consequence of an increasing number of cores and a more distributed nature of the machine, communication between parallel execution contexts is gaining importance and can harm performance if implemented sub-optimally. Rather than using shared-memory and consequently relying on the cache-coherence protocol, one solutions is to use a message-passing based programming model. Here, each pair of cores uses distinct buffers avoiding contention. The challenge then is to build efficient group communication on top of individual peer-to-peer channels. If for example a tree topology was to be used, the optimal outdegree in each node depends on the ratio of the costs for sending and receiving messages and the order in which to send messages depends on the cost of communicating on links. However, this information is not available to the OS or application runtime. For example, send and receive costs depend on the cores that are communicating and is highly specific to the machine and as such hard for the programmer to configure manually.

Good memory allocation and synchronization are tricky to achieve on today's multicores: Firstly, detailed knowledge of hardware characteristics and a good understanding of their implications on algorithm performance is needed for efficient and scalable implementations. However, programmers are often oblivious to the intricate details of their hardware and fail to reason about implications for their algorithm's performance.

Secondly, hardware evolves quickly meaning that design choices must be constantly re-evaluated to ensure good performance on modern hardware. Thirdly, because hardware is diverse, even if a program is optimized well for a certain machine, machine-specific optimizations are unlikely to work on other machines that programmers might be targeting. Both, hardware diversity and evolution impose high engineering and maintenance costs, even for expert programmers that understand the complexities of their hardware.

Lastly, the performance of program does not only depend on the machine, but the choice of algorithms also depends on the program itself. For example, memory access patterns for example should be considered when deciding where to allocate memory.

While manually tuning programs is worthwhile in high-performance computing or niche markets, where hardware changes less rapidly, general purpose machines have too broad a hardware range for this to be practical for many domains. The result is suboptimal performance on most platforms due to programmers failing to keep up with hardware changes.

Hence, as a consequence, of suboptimal memory management and inefficient synchronization the performance of parallel programs can be severely limited on general purpose multicores. This can be seen in many programs today. Programmers take little care of where memory is allocated and how it is accessed and use synchronized primitives that are not tuned to the machines they are running one.

In this thesis, we investigate how memory allocation and synchronization can be configured automatically to relief programmers from having to understand intricacies of complex and diverse multicore hardware.

Goals and contributions

To aid programmers in coping with constantly evolving hardware, we aim to provide a framework for the development of parallel programs that achieves good performance without requiring any low-level manual tuning of memory allocation and synchronization primitives. This releases programmers from the burden of having to repeatedly re-write applications due to changes in the hardware landscape.

To address the problems described in the previous section, this thesis explores and combines several ideas (visualized in Figure 1.2):

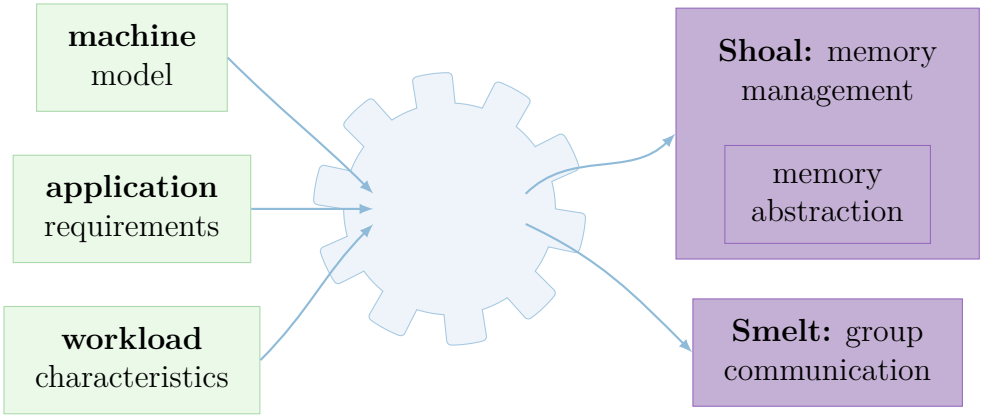


Figure 1.2: Thesis overview

Machine model A model of the machine that makes relevant aspects of the hardware and their application on algorithm performance available to programmers and the runtime system. We show how good performance for synchronization primitives and memory access can be achieved across a wide range of machines if the model is used to tuned their implementations.

Application requirements A set of compiler extensions that extract application resource requirements automatically out of high-level parallel languages. We show this for memory access patterns.

Workload characteristics Application resource requirements often not just depend on algorithms, but also the workload there are executing. Memory access patterns, for example, can be expressed in a more fine-grained manner based on characteristics of the input workload. For graphs, we currently express memory access patterns depending on the number of nodes and edges in the graph.

Shoal's memory management A runtime that smartly tunes memory allocation and access based on memory access patterns and the machine model without programmers involvement. The enable that, we provide a memory abstraction that allows to swap the low-level implementation for memory allocation and access without having to recompile the program.

Smelt's group-communication A machine-aware tree topology exposing basic broadcast primitives to programmers that can be used to implement scalable higher-level algorithms in a simple fashion, yet achieving performance competitive to complex state-of-the art algorithms.

We believe that our approach allows to develop robust programs that are less sensitive to changes introduces by evolving hardware.

Collaborative work

In addition to my advisor Prof. Dr. Timothy Roscoe, work presented in this thesis has done in collaboration with other researchers. Shoal was inspired by my internship with Dr. Tim Harris at Oracle Labs in Cambridge, UK and later continued in collaboration with Reto Achermann, who was leading the efforts to port Shoal to Barrelfish.

Work in the later stages of Smelt has been in collaboration with Reto Achermann, Roni Haecki and Sabela Ramos from the Systems Group.

Structure of the dissertation

The structure of the thesis is as follows: Firstly, in Chapter 2, we introduce our machine model. It unifies hardware information that we utilize to configure memory allocation and synchronization primitives such good performance and scalability of parallel programs can be achieved across a wide set of multicore machines.

In the context of this work, we have been focusing on memory management and synchronization based on message passing. Our approach to memory management is to leverage memory access patterns implictly encoded in domain-specific languages or provided manually by the programmer. We use these patterns to choose online at program execution time depending on the

machine configuration whether replicate, partition or distributed memory. Further, if appropriate, we make use of additional hardware features such as DMA engines and large pages. We discuss access patterns in Chapter 3 and Shoal our smart memory allocator in Chapter 4. The content of these two chapters has been published in part in [KARH15].

Following that, we introduce our machine-aware broadcast trees in the offered by Smelt in Chapter 5. We motivate the usefulness of a highly tuned multicast trees for the use of high level applications such as barriers and agreement protocols that then automatically benefit from the machine-aware low-level implementation. Together with parts of the machine model (Chapter 2), this has been accepted for publication in [KAH+16].

Finally in Chapter 6, we summarize our work and give an outlook on future work.

2
Machine model

Introduction

This section introduces our multicore system model, which captures hardware details and their implications on application performance. The need for this arises from trends in hardware: multicores are increasingly complex and diverse. Hardware also changes at an increasingly fast pace. As a consequence, performance of applications has to be reevaluated constantly to keep up with hardware trends. We give an overview of modern multicore hardware in Section 2.3).

As a result of these hardware trends, programmers struggle to capture the implications on their software's performance when running on a particular machine [WWP09]. Hardware models are useful to guide programmers, but there is a trade-off between a model's simplicity and accuracy [CKP+93a]. Our model is designed specifically to aid memory allocation and tune synchronization primitives based on message-passing.

Our goal with the machine model is to to provide a simple way for programmers to acquire an accurate representation of a machine's performance automatically and programmatically without programmers having to dive into the specific of that hardware. This is in contrast to other work [RH13, RH16] trying to be more accurate, but also requiring more effort to establish the model.

A consequence of this is that we take the results from micro-benchmarks we run to capture machine characteristics "as is" rather than trying to reason for each individual machine which concrete hardware feature might causes what effect observed in these benchmarks. This strategy makes it simpler to acquire the machine model, avoids having to understand intricate machine characteristics and their interplay and simplifies analysis because the execution of micro-benchmarks closely follows the expected execution of applications intended to be evaluated with the model.

Motivated by the focus of this thesis, we intend to capture characteristics of multicore systems that are specifically relevant for them. This is in contrast to some other work that aims to be more generic [CKP+93b, WWP09]. Here, we focus on:

Memory allocation With multicore hardware's shift towards NUMA architecture, programmers have to consider where to allocate memory to achieve good memory latency and throughput. Relevant hardware characteristics here are mostly the number and size of NUMA nodes, but also the availability of superpages and memory-copy hardware (DMA). This information is mostly available from hardware registers and libraries such as `libnuma`, but we extend it by information about the availability of DMA hardware and various page sizes.

Message-passing-based synchronization Parallel applications require synchronization such as barriers. If synchronization is implemented on peer-to-peer message-passing channels and group-communication has to build on top of individual channels, the topology of message dissemination matters. For a good choice of topology, the pairwise send and receive cost between processor cores matters, but this information is not available from hardware registers on a sufficiently fine grain level.

Our machine model as presented in this chapter is useful to capture characteristics of multicore machines and especially designed to aid programmers with implementing memory allocation and message-passing-based synchronization. Based on the knowledge encoded in the machine model, algorithms can be selected and optimized to the characteristics of a concrete machine. This releases programmers from the burden of having to understand complexities of multicore machines themselves.

The idea behind the machine model is to capture as much of the machine details as possible in a model. We fill the model from two sources:

- static machine characteristics read from information provided by hardware in registers and
- low-level micro-benchmarks to measure specific performance aspects of a concrete machine where statically provided hardware information is not readily available or insufficiently precise.

After a discussion of related work in Section 2.2, we give an overview of current trends of multicore hardware in Section 2.3. This includes a summary of the characteristics of the interconnect network enabling communication between cores. In Section 2.4, we then describe the characteristics we capture in our machine model. This includes characteristics available from hardware registers, but also describes our micro-benchmarks to evaluate a machine's support for memory copy operations and pairwise messaging performance.

For message-passing, we then discuss in Section 2.5 the implications for programmers following from multicore message-passing characteristics and work out the differences to existing work in the field of traditional distributed systems. Finally, Section 2.4.2.2 describes how we represent this information to the programmer.

We use the machine model discussed here to select and tune memory allocation (Chapter 4) and synchronization primitives (Chapter 5) to a concrete machine without programmers having to understand the characteristics of the machine. In addition to the machine model, memory access patterns are crucial for tuning memory allocation. We present a detailed discussion about memory access patterns and how to acquire them in Chapter 3.

In the context of our synchronization work, we evaluate the accuracy of our model for message passing (Section 5.5.3).

Related work

Machine models

Various attempts to modeling computers exist with different purposes. First, some machine model try to capture general computer performance characteristics, often focusing on the trade-offs between computation and memory access. While they often address communication, they typically do not model communication in detail, but focus on how communication affects computation. The cost for communication is generally assumed to be uniform. As such, they are not suitable to apply fine-grained tuning of synchronization primitives.

Another class of models focuses on communication. Such models normally originate from the field of networks or distributed systems research and often do not directly apply to multicore hardware.

Finally, there are approaches suggesting to model the cache-coherence protocol in full detail. The motivation for this is that communication on multicores is implemented using load and store instructions that implictly trigger the cache-coherence protocol to send cache lines between caches. If the state machine of each cache line in each cache is fully modeled, the performance of the multicore communication can be accurately predicted and algorithms developed based on that model. We now give details on all of these classes of models.

Generic machine models Culler *et al.* [CKP+93b] introduce the LogP model. It expresses critical trends of parallel computers and is used as basis for developing fast, portable parallel algorithms. The model is based on computing as well as communication bandwidth, delay of communication and, finally, efficiency of coupling communication and computation. However, it

assumes that receive and send time are equal and hence does not seem to be good fit for multicore machines.

The roofline model [WWP09] is a simplified model to predict the performance of programs on modern multicore machines. However, it is limited to modeling only computation and memory throughput. Performance of communication between processors cannot be deduces as it is normally executed by the cache-coherence protocol and does not directly involve memory. Furthermore, the roofline model focuses on memory throughput and does not express latency.

Even if extended to capture memory access latency [CP14], the model is of limited use. In cases where message buffers are not cached and have to be fetched from memory, the roofline model cannot accurately model communication cost as it does not encode the differences in memory access performance as a result of NUMA effects.

A step further go approaches such as gem5 [BBB+11], which provide a full-system emulation. If the interconnect would be modeled with all its details, full-system emulation could be used to execute programs offline to find performance characteristics for a concrete algorithm. However, this still does not fully solve the problem, as information present in a low-level model such as used as an input to a simulator does automatically help designing machine-aware algorithms; it could merely be used to evaluate them offline.

Communication models Another related communication model is the telephone model [SLL10]. In contrast to other models, it has a sequential aspect in setting up connections to other peers: only one connection can be opened at the time, so opening n connections takes n times longer than a single connection. Within the telephone model, finding a minimal broadcast topology is known to be NP-complete [SCH81]. In this chapter, we will related to multicore broadcasting problem to the telephone model and use that to prove the NP-completeness.

Models of the cache-coherence protocol Ramos and Hoefler [RH13] apply the LogP model to cache-coherent SMP systems and evaluates their results on the Intel Xeon Phi. They compare the performance of several trees for broadcast performance and build an optimal tree that performs better than all other trees the considered for evaluation. They also evaluate in detail the hardware characteristics of the Xeon Phi. We take a different approach. Rather than understanding details of the hardware (which is tedious, and due to diversity as well as fast pace at which multicores are changing, not practical on the long run), we take what we measure in our offline benchmarks as given, and find ways of using it without having to understand the cause for them in detail.

Another alternative is to model a machine in all its details [RH16], including especially the cache-coherency protocol. We think that, for our pur-

pose, the effort of doing this is too tedious, especially because we found the measurements to be accurate enough. Also, due to hardware diversity, in-depth modeling of one machine does not necessarily apply to other machines. Furthermore, hardware details required for such an accurate model of the machine are often not publicly available from vendors.

Machine knowledge bases

Barrelfish's system knowledge base (SKB) [SPB+08] was designed to store a rich representation of the machine, but still being easy to use give a high-level query interface. The idea is to store as much raw information about a multicore machine in the model. On top of that, programmers (and the operating system itself) provides high-level queries that reason on top of this information for ease of use.

Infokernel [ADADB+03] argue that having information about hardware unified in one database allows to use resources more efficiently in higher-level services. As example, they mention how an application can tune memory management to avoid trashing memory by processing data in multiple passes such that each of them fits in the memory and no disk accesses are required. Analogous to that, we argue that our model of the machine serves the same purpose, but is more geared towards implementing low-level memory management and synchronization services.

Complexity and programmability

Multicores are hard to program: memory latency and bandwidth depend on which core accesses which memory location [BPS+09, BWCC+08]. The interconnect may suffer from congestion when access to memory controllers is unbalanced [DFF+13]. Performance can also suffer from false sharing[Bry04]. It often hinders performance of parallel programs if not machine characteristics are considered when programming them.

Another challenge when programming NUMA machines is the choice of page size. Gaud *et al.* [GLD+14] discuss the trade-offs when using large pages. While increasing the TLB coverage, large pages make the granularity at which NUMA memory optimizations can be applied more coarse grained. They refer to this as *page-level false sharing*, which means that unrelated data shares the same page preventing it from being split up and stored on different NUMA nodes preventing for example fine grained partitioning. In many cases this harms performance more than gained from a reduction in TLB misses.

Unfortunately, ignoring hardware details can severely harm performance significantly [RH15]. All of this is motivation for us to encode available hardware information into one single database to aid programmers and runtime systems in understanding hardware they are executing on.

Sources for hardware information

Machine information can often be retrieved from hardware registers. Several software frameworks exists to ease access to that information. Typical Linux systems provide libnuma [Sil15a] for information concerning the hierarchy of a machine as well as a programming interface to specify where exactly memory should be allocated. Compared to that, *sysfs* [Moc05] provides a more low-level interface for hardware information. It has a wider scope, but use is harder since there are no high-level access functions to access it.

Trends in multicore machine hardware

For decades, performance of computers has been improved by increasing the frequency of processors. Applications often had an immediate benefit from that, without the need to be rewritten. Since then, multicore computers have long reached physical limitations for scaling up the frequency of individual cores. Instead, Moore's law translates into an increase in the number of cores rather than improving the speed of individual processors. This causes several trends in modern multicore machines hardware:

Parallelism of computation Hardware provides more parallelism [Bry04, BEA+08, FKMM15]. As a consequence, programmers are force to parallelize their algorithms to benefit from advances in hardware design. However, not all software can easily be parallelized, and even if it can, this process is often error-prone and tedious. Parallel software typically requires synchronization between its execution contexts, which in many cases limits scalability of parallel programs.

Hierarchical memory As a consequence of more parallelism and the need for a higher memory bandwidth, more memory controllers are added and memory is now distributed within a multicore machines. While hardware still provides the illusion of a single cache-coherent shared-memory, in reality memory is split up in multiple memory controllers. Processors are then more local to some of these memory controllers than others. As a consequence, the choice of where memory is allocated matters for the performance of parallel programs. To hide high cost of accessing memory, many processors employ the concept of Simultaneous multithreading (SMT), where one physical processor exposes several virtual processors: fast context switches between these co-processors and rescheduling of other virtual processors in the presence of memory stalls allows to hide high memory access latency. SMT is, for example, implemented as Intel's hyper-threading technology or on SPARC.

Complex interconnects Processor and memory controller are connected by interconnect networks. They are often hierarchical: intra-socket

links have typically have a lower bandwidth and higher throughput compared to inter-socket ones. Interconnect performance is crucial for accessing data as well as synchronization between threads. Examples are Intel's QPI and AMD's Hypertransport.

Figure 2.1 shows an example of a machine eight socket multicore machine, six cores each, and one memory controller per socket. The interconnect in that case is a "twisted ladder" topology, with communication between sockets being slower than inter-socket communication.

Multicores provide an explicit caching mechanism to allow CPUs to execute computation without having to fetch data from much slower memory banks for each instruction. In addition to per-processor Level 1 and Level 2 caches, multicores often provide a shared last-level cache per socket. Caching relies on spatial and temporal locality. Only a limited number of data entries can be kept in the much smaller caches compare to much bigger main memory. However, if the same data is accessed repeatedly, the required data might already be cached from previous accesses.

Furthermore, since the size of transfer between memory and caches is typically much bigger than the size of transfer between caches and CPUs and the size of data a CPU can operate on in one cycle, several instructions can be executed on the same cache line if accesses with spatial locality. All these optimizations help programs that sequentially access memory. However, there is no hardware optimization that helps with more intricate access patterns. With increasingly distributed multicores computers, it becomes inevitable to come up with more sophisticated ways of programming such machines.

Following these hardware trends, multicores behave increasingly like distributed systems [BPS+09]. Communication cost is non-uniform as a consequence of the hierarchical interconnect network and its links are susceptible to congestion, for example as a result of imbalanced access to memory when allocation is restricted to only a small number of memory controllers.

Due to these similarities, techniques known from traditional distributed systems such as replication and partitioning of data can then help to overcome scalability problems on multicore computers as well, but which strategy to apply is highly machine-dependent. To help choosing, it is crucial for programmers to be able to access a meaningful machine model that acts has an ontology of hardware characteristics for a certain machine.

However, multicores are also different from what has been explored traditionally in the field of distributed systems. For example, one difference is that links on today's multicores are reliable and deliver messages in order. Furthermore, propagation time — the time from sending to receiving a message — is almost instantaneous. This is also in contrast to traditional distributed systems where the propagation time often dominates the cost for a message transfer. In Section 2.5, we show a more detailed comparison of multicore interconnects and distributed systems.

Future machines will likely be even more complex: they may not provide

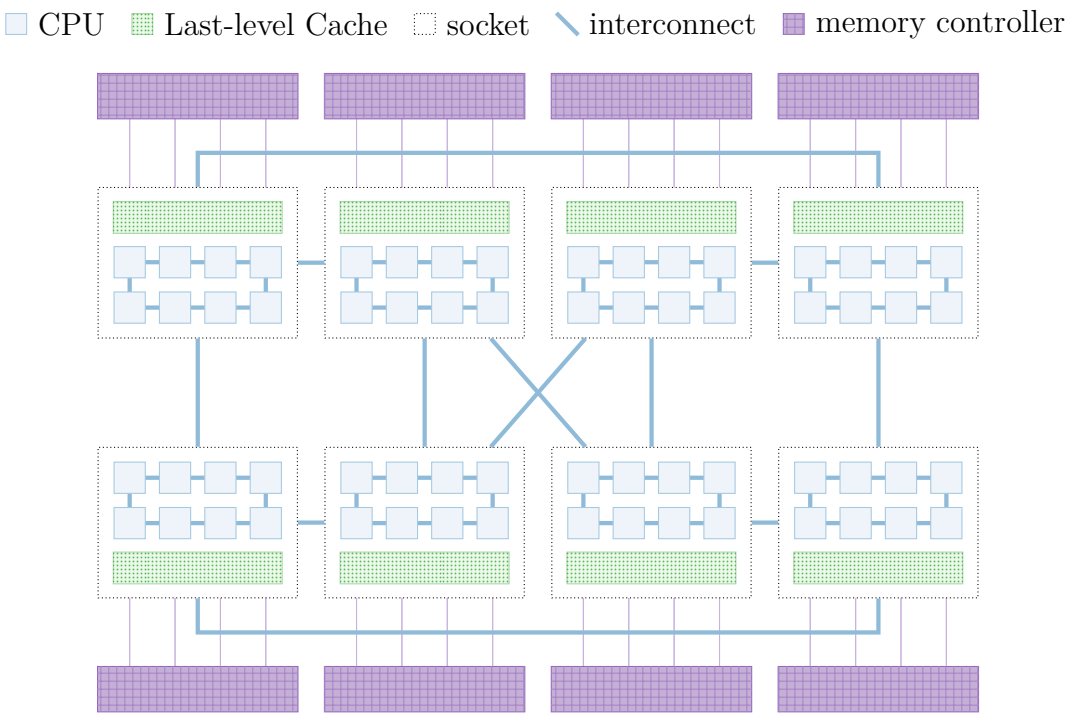


Figure 2.1: Architecture of a modern multicore machine.

global cache coherence [HDV+11, MVdWF08] — although others [MHS12] think differently — , or even shared global physical addresses [Ach14]. Even today, accelerators like Intel's Xeon Phi, GPGPUs, and FPGAs have higher discrepancy for the performance of local vs. remote memory accesses [Ach14]. Such hardware demands even more care in application data placement.

Communication on multicores

We now describe the message-passing characteristics of multicore machines. Figure 2.2 shows an overview of message round-trip between two threads and the following is a detailed description of the times involved:

t_{send} denotes the time to send a message and is normally actively perceived on the sender's core as the cost of the corresponding memory store instructions. On multicores, sending a message involves invalidating the cache line to be written and therefore often depends on which cores the cache line is shared with. This typically takes a few hundred cycles. However, t_{send} may also depend on the sequence of previously sent messages from the sender. Moreover, it may vary with the state of the cache line to be written. As a simplification, we assume that the reader is already polling the cache line since, as we will see later, this closely resemble our message-passing workload.

$t_{receive}$ denotes the time to receive an already queued message. That message

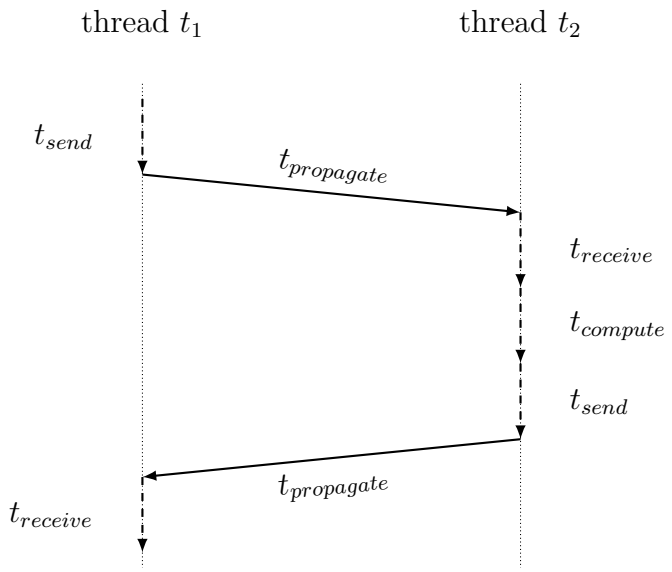


Figure 2.2: Message-based communication between threads t_1 and t_2

is either still held in the sender's cache in exclusive state or evicted to memory. When the receiver issues a memory load to read the message, the delay until the message is transferred to the receiver's cache shows up as the cost of the load instruction issuing the transfer. The corresponding store operation takes multiple hundred to thousand cycles. With the transfer, the state of the corresponding cache line changes in the receiver's cache from invalid to shared, and from modified to shared in the sender.

$t_{propagate}$ time it takes to propagate a message on the interconnect. Propagation time is neither visible on the sender nor receiver, but charged as part of the t_{send} and $t_{receive}$ as discussed before. We assume $t_{propagate} = 0$, as propagation times on multicores are typically very small due to t he short distance between hardware components inside of a single machine. If needed, they can be treated as part of the $t_{receive}$.

Based on these individual send- and receive times, Figure 2.3 visualizes sending multiple such messages on a multicore system, specifically the time that it takes for a thread v_i to send a message to two threads v_j and v_k. Note that sending multiple messages as executed from v_i is a sequential operation: sending the second message can only be started after the first one has been sent. However, receive operations on different nodes execute in parallel. Both, t_{send} and $t_{receive}$ depend on who is sending and receiving the message.

Unlike classical networks, t_{send} and $t_{receive}$ times dominate the total transmission time. This is especially significant as the sending and receiving threads are blocked for t_{send} and $t_{receive}$ respectively. This implies that the cost of sending n messages grows linearly with the number of messages to be

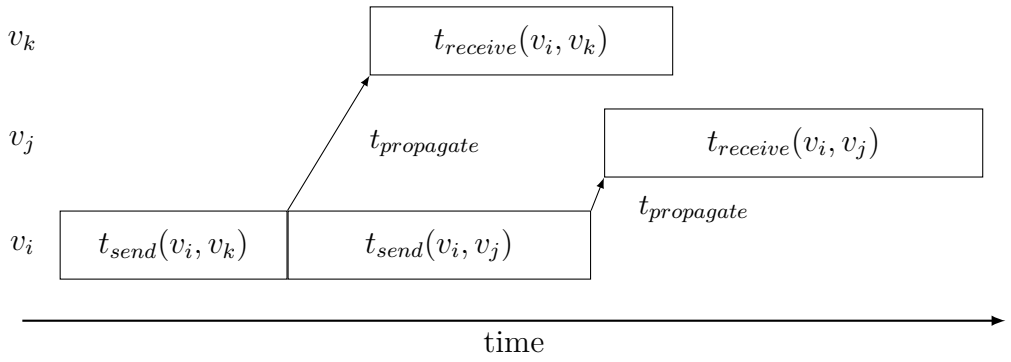


Figure 2.3: Visualization of a send operation: thread v_i sends a message to v_k followed by another message to v_j. Send operations are sequentially executed, while the receive operations can be processed in parallel on threads v_j and v_k.

sent, whereas in classical distributed systems, $t_{propage}$ normally dominates independent of how many messages are sent in a single round trip. Section 2.5 further discusses differences of communication in multicore machines with classical networks and their effects on programming them.

Note that t_{send} and $t_{receive}$ as shown in Figure 2.3 might actually overlap on real hardware as a result of complex interactions of the cache-coherence protocol on sender and receiver.

Characteristics captured by the model

Our model captures information given by hardware and enriches that with low-level machine-characteristics acquired from microbenchmarks.

General information from querying hardware

The intention of the machine model is to obtain information about the system architecture. Many details are available from hardware registers and can directly be collected from there. Here, we give a list of these.

Memory controllers For main memory, we are mostly interested in the number of memory controllers, the size of memory attached to each of them, as well as their hierarchy. On Linux, this information can be obtained using *libnuma* [Sil15a]. In Barrelfish, hardware information is stored in the system knowledge base [SPB+08].

Page sizes and tlb configuration Modern multicore machines offer various page sizes. For big working sets, the choice of bigger page sizes can reduce the number of TLB misses, hence benefiting overall performance of a program. However, it can also harm it [GLD+14]: if

pages a bigger, they have to be allocated on the same single memory controller, which can lead to load-imbalance of memory accesses and might reduce memory locality when accessed concurrently from several threads on different NUMA nodes. Furthermore, it can lead to fragmentation (since a multiple of the page size has to be allocated, even for data structures that are smaller than the page size). Larger page sizes can also reduce performance due to a higher cache-conflict rate. Even worse, modern hardware has several translation lookaside buffers (TLB) for various page sizes. The TLB configuration is important to consider for choosing the page size for large data structures.

Processors We store the number of processors, each core's affinity to main memory, as well as the availability and number of SMT threads. This is crucial for deciding how many threads to use, e.g. for synchronization-heavy algorithms such as barriers, where using more software threads than physical cores normally hinders performance. However, communication between hyper-threads is often relatively cheap.

Caches The unit of transport between the memory and caches as dispatched by the cache-coherence protocol is a cache line. If data has to be exchanged efficiently on a multicore machine, the alignment of data to the start of the cache line as well as its size as a ideally multiple of the cache line is crucial. Furthermore, a cache line is also the unit of consistency: updates to a cache line cause the cache-coherence protocol to invalidate other copies of it in all other caches of the system. Semantically independent programs operating on the same cache line trigger cache line invalidations that would not strictly be necessary. Consequently, suboptimal allocation of memory to cache lines that severely limit the performance of parallel programs.

However, the information given by hardware registers and consequently software libraries on top of it is often too coarse grained. For example, ACPI gives information about the communication cost between NUMA nodes. However, this information does not distinguish send from receive cost, which is a crucial parameter for configuration message-passing topologies.

Micro benchmarks

Since hardware-given machine-characteristics are not always sufficient, we now present a set of microbenchmarks we are using to enrich our model.

DMA engines

This work was executed in collaboration with Reto Achermann.

Modern multicores often provide direct memory access (DMA) hardware that supports memory-to-memory copy operations. These can be used to

efficiently offload copy operations from CPUs. Our model expresses whether a machine is capable of such hardware support, but also the efficiency of it compared to the parallel use of CPUs to copy the data.

We conclude that the latter is often faster, as DMA engines typically provide a limited number of channels, less than there are processors available in typical system. This reduces the number of parallelism available for copying data. However, if DMA engines are used for copying data, processor are available for computation elsewhere.

When initializing memory to a constant value, or copying data between several memory locations, there are several choices on how to execute an operation like that:

- sequentially: process each element with a single processing unit, for example using the `memcpy` operation.
- leveraging data parallelism: process elements in parallel, for example with OpenMP parallel loops
- hardware support: the use of hardware copy units to aid copy operations.

In order to help programs decide when to use DMA engines or to allow libraries and runtime systems to dynamically choose whether to use hardware support for copying data, our model encodes the performance of using DMA engines versus using processors to copy data in software.

Figure 2.4 shows a raw throughput evaluation for copying memory. We compare the time required for copying memory in parallel on a subset of the processors to the time for offloading the same copy operation on DMA engines with a varying number of channels. Surprisingly, our results show that the use of DMA engines does not reduce the runtime of the copy-operation compared to using a large number of threads.

We conclude that offloading to DMA engines does not improve the performance of a program that depend on the completion of the copy operation. This is not surprising, as memory bandwidth is typically a bottleneck and the CPUs should be able to saturate it. In such a case, the program's execution depends on the result of the copy operation and cannot execute any other work for the same application. However, when copy operations can be executed asynchronously, processors are available for other computations. We show later in Section 4.6.3 for a real-world example application how a DMA engine can be used nevertheless for efficient memory copy operations, even if it cannot be executed asynchronously.

Pairwise send- and receive cost

An essential element of parallel programs is synchronization between threads. On multicore machines, the cost for communication between threads depends

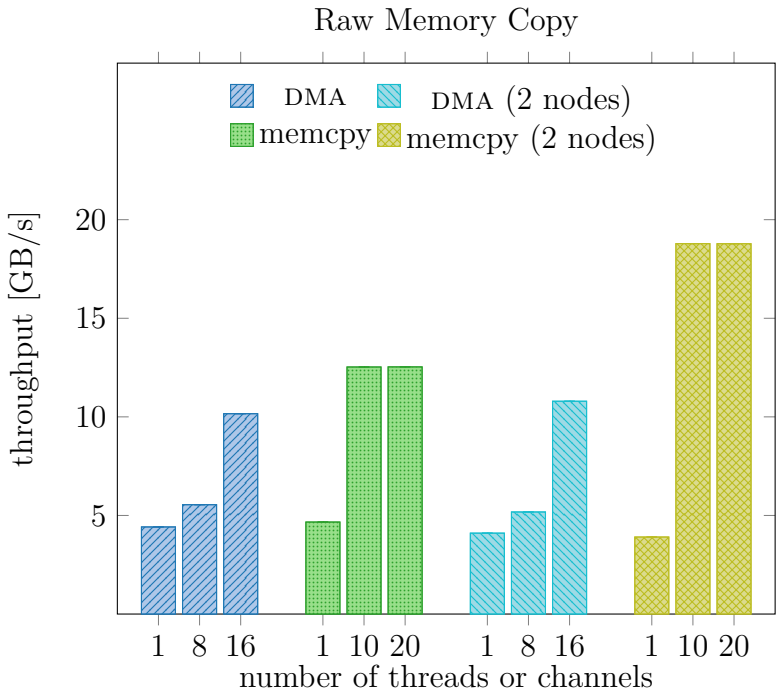


Figure 2.4: Comparison of parallel OpenMP copy and DMA copy on I IB 2x10x2 for large buffers (≫ cache size)

on which cores are communicating with each other. For example, the latency of exchanging data between NUMA nodes is typically (but not always) higher than within nodes, since messages have to be transferred via the interconnect. Similarly, the bandwidth for communicating across NUMA nodes is typically lower.

Synchronization protocols are normally optimized for low latency. For the overall performance of higher-level synchronization protocols, it is essential to know the low-level cost of a single message transfers for a pair of individual cores. Since this cost is machine-specific, we capture the time for sending and receiving messages as well as the message round-trip time offline on an idle machine. Note that the propagation time is hard to measure in practice since it is very small and observed in software as send and receive costs, but we found it sufficient to simple assume it to be 0. $t_{compute}$ is application specific. We use `rdtscp` on all our machines, which in contrast to `rdtsc` is serializing and cannot be executed out-of-order [Adv13, Int10]. This is with the exception of I KNC 1x61x4, where `rdtscp` is not available, but it is safe to use of `rdtsc` without the `cpuid` instruction for serializing instructions as its processor has an in-order pipeline.

```
static inline uint64_t get_tsc(void)
{
  uint32_t eax, edx;
  __asm volatile ("rdtscp" : "=a" (eax), "=d" (edx) :: "ecx");
```

```
  return ((uint64_t)edx << 32) | eax;
}
```

In our microbenchmark, we measure t_{send} and $t_{receive}$. Listing 1 shows the code executed on the sender's side in the pairwise benchmark. For each pair of cores s and r, the sender s sends a batch of `BATCH_SIZE` messages to receiver r and waits for the receiver to acknowledge completion of the benchmark by receiving a notification message. This assures that only one pair of cores is evaluated at a time. The system is idle otherwise.

Since the cost an individual transfer is hidden by the write buffer on many machines, we send multiple messages in a batch. The write buffer is typically small and, once filled up, later messages will be exposed to the actual time needed for sending the message. We currently use a batch size of `BATCH_SIZE = 8`, since that is in the order of the number of messages a typical message-passing based broadcast would send and is large enough to mitigate the effects of the write buffer. Choosing the batch size is tricky and will likely have to be revisited in the future as hardware changes. For the machines we have been looking at in the context of this thesis, the results achieved with that however seem to be sufficient.

Since normally the reader is already polling the cache line to be written already, the cache-line will most likely be in the receivers cache in shared state. Hence, before writing, the cache-line has to invalidated before the write can be executed contributing to the send cost as measured in this benchmark.

Listing 1 Pseudo code of the pairwise benchmark on the sender's side

```
// Reset TSC timers
uint64_t t_start = get_tsc();

// batch-send messages
for (unsigned j = 0; j < BATCH_SIZE; j++)
    send(/* .. */);

t_send = get_tsc() - t_start; // record TSC diff for send

// Wait for completion
receive(/* .. */);
```

Listing 2 shows the code as executed by the receiver r. The challenge here is to measure the cost of the actual receive operation excluding the wait-time, i.e. the cache-line transfer that triggers the messages to be transferred to the receiver's local cache. We explicitly do not want to include the wait time as it is application specific and depending on the algorithm's state. We achieve this by resetting a `rdtsc`-counter after each failed polling operation until the first message is received and calculate the difference in the `rdtsc` values after receiving the message.

Listing 2 Pseudo code of the pairwise benchmark on the receiver's side

```
// Wait for messages; reset TSC if no new message was
// available in last poll cycle
uint64_t t_start;
do {
    t_start = get_tsc();
    err = try_recv(/* .. */);
} while (err != SUCCESS) ;

for (unsigned j = 1; j < BATCH_SIZE; j++) {
    receive(/* .. */);
}
t_receive = get_tsc() - t_start; // record TSC diff for receive

send(/* .. */); // notify completion to sender
```

Note that the times t_{send} and $t_{receive}$ measured here are as they would be perceived by threads in a real application making them a good fit for programmers to use as a foundation to predict the performance of their message-passing based algorithms when executed on a certain machine.

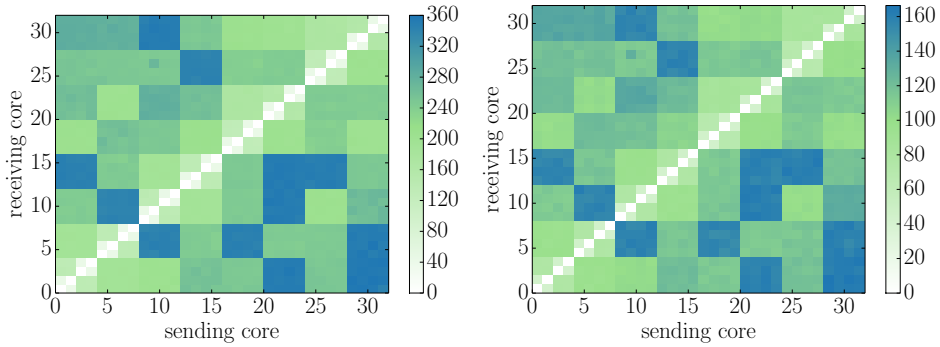


Figure 2.5: Pairwise latency on A IL 4x4x2 — left: receive, right: send. We list machine configurations in Section A.

Figures 2.5 and 2.6 show an example of pairwise receive cost for two typical multicore machines (A IL 4x4x2 and I SB 4x8x2). Shown is the pairwise receive cost when receiving and sending a message between the cores shown on the x and y axis respectively. We calculate this from the cost of sending and receiving a batch of messages divided by the batch size. The color of the heat map plot shows the cost in cycles.

Both machines clearly show that communication within nodes is cheaper that across. NUMA nodes in all figures show up as rectangles grouping the cost into clusters. While I SB 4x8x2 shows a symmetric cost, A IL 4x4x2 does not. On both machines, the cost of sending a messages is cheaper than receiving one, most likely partially because the actual cache-coherency invalidation required for init the send is hidden by the write buffer for the

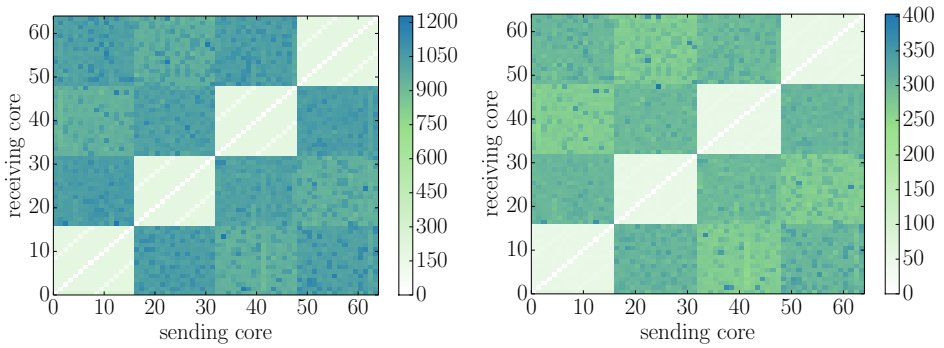


Figure 2.6: Pairwise latency on I SB 4x8x2 — left: receive, right: send (cores reordered according to `libnuma` such that contiguous cores are on the same NUMA node). We list machine configurations in Section A.

first few messages.

Pairwise send and receive times accurately reflect the intricate details of message passing performance on a concrete machine. No detailed understanding of the cache-coherence protocol in use is required. Instead, our micro-benchmark has to be executed only once to capture required details. This is in contrast to information given by hardware vendors, such as encoded in the ACPI table.

Since our measurements are done a-priori, they cannot encode runtime effects such as interconnect traffic and CPU load.

Later in Chapter 5, we will show how we can use pairwise message passing cost to build a machine-aware broadcast tree automatically.

Representation in a graph Communication in a multicore machine can be represented as a fully connected graph $G = (V, E)$. Vertices v_i correspond to hardware threads in the multicore, and edges $e = (v_i, v_j)$ model communication between threads v_i and v_j, with edge weights as a tuple of $w(e) = (t_{send}, t_{receive})$. We show an example of such a graph in Figure 2.7.

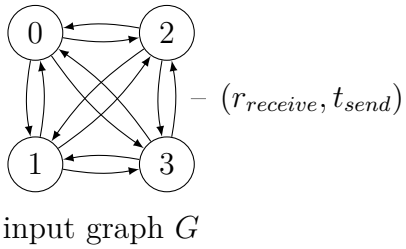


Figure 2.7: Example of fully connected input graph for four CPUs.

With a few differences, our model is similar to the telephone model [SLL10]. In the telephone model, each participant has to dial other participants sequentially before transmitting data. Similarly, our model has a sequential

component when sending: the thread is blocked for the duration of t_{send}, and consecutive sends cannot be executed until the previous ones are completed. However, note that several threads can send messages concurrently and independently of each other.

The weight of edges in our model is non-uniform: the cost of receiving messages from cores that are further away (NUMA distance) is higher. Depending on the cache-coherence protocol, the cost of sending messages might also depend on the distance of the receiving core, since cache lines may have to be invalidated before they can be modified.

Implications for programmers

This work was executed in collaboration with Kornilios Kourtis.

Today's and future multicores are distributed systems. However as seen from Section 2.4.2.2, their network is different from what has been evaluated in the context of classical distributed systems. This causes problems when applying traditional distributed algorithms to program them. In this section, we describe multicore networks, enumerate differences to traditional distributed systems, and analyze their implications on algorithm performance.

Propagation time is not dominating on multicore systems. To illustrate how this affects complexity measures of distributed algorithms, we now analyze the complexity of a example algorithm: a quorum. One application of quorum protocols is to retain consistently when updating replicated state. For updates, acknowledgments need to be collected from a majority of cores. To acquire a quorum, acknowledgments need to be fetched from a subset of n cores in the system. Conflicting updates do not reach a quorum and are aborted.

As a first step, we conduct a simplified analysis of the time required for sending n round-trip messages with a send time t_s, a receive time t_r and a propagation time t_p. In contrast to traditional distributed systems, multicore interconnects have the following characteristics:

- CPUs cannot concurrently send and receive messages
- communication channels are reliable
- the propagation time t_p is small compared to the time for send t_s and receive t_r

If n message round-trips are executed sequentially, the cost of the quorum is $2n(t_s + t_p + t_r)$. A common optimization is to overlap propagation time with send operations of consecutive messages. The cost then depends on whether or not all messages can be send before the first response is received. If the accumulated send time is smaller than the cost of a round-trip, the sending core has to wait for the remainder of the round-trip time (RTT) before

starting to receive messages. This is $2t_p$ for the propagation time in both direction, $2t_s$ for the send operation on both sides and t_r for the receive on the remote core, in summary $2t_s + 2t_p + t_r$, before replies are processed. If the accumulated send time nt_s dominates, the core starts receiving replies immediately after sending the last message. For both cases, the time required to receive all messages is nt_r. This yields:

$$\text{overlap:} \quad \max\Big(nt_s,\ (2t_s + t_r + 2t_p)\Big) + nt_r \tag{2.1}$$

Traditional networks In traditional distributed systems, the propagation time t_p is the dominating cost for sending messages. Send time t_s and receive time t_r are orders of magnitude smaller. We show a breakdown of sending a UDP message in Table 2.2.

Equation 2.1 resolves to $2t_s + (n+1)t_r + 2t_p$ and since the propagation time t_p dominates it is further reduced to $2t_p$. In that case, the number of messages send during a round is irrelevant for the time it takes an algorithm to complete.

Consequently, the complexity of distributed algorithms is typically expressed in rounds — i.e. the time required to send a message, wait for the reply and receive it. Propagation time can be hidden by overlapping it with successive sends. Propagation is often expensive enough to allow send a message to every peer while waiting for the first one to reply.

Multicores interconnect networks Expressing the complexity of algorithms in rounds, as with traditional distributed systems, does not work very well for multicores as the propagation time on multicore machines is in the order of package processing time in software (Table 2.1).

In contrast to traditional systems, where the message is already locally buffered when the receive function is called, it first has to be transmitted via the interconnect on multicore machines. This is a consequence of the cache line containing the data being invalidated elsewhere when modified by the sender. The load instruction triggered by a consecutive receive operation then has to fetch this cache line again from a potentially far-away NUMA node. This cost is actively perceived by the receiving thread, as it will be blocked waiting for the cache line to arrive.

Equivalently, a store operation on a cache line can only be executed after the cache coherency protocol ensures its exclusive access on the sending core. This is a expensive operation. While often hidden by a write buffer, this cost will eventually be perceived by software with an increasing number of updated cache lines in flight.

With these observations, equation 2.1 resolves to: $nt_s + nt_r + nt_p$. Algorithm complexity is determined by the number of messages. As an example, we now show the implications of our observation to quorum applications. In classical networks, propagation time is hidden by proactively sending mes-

what	time [nsec]
send t_s	199.3
receive t_r	34.9
user-level	298.9
$\sum$	537.0
propagation t_p	93.9

Table 2.1: Communication channel cost breakdown on A SH 4x4x1. The cost for message processing t_p cannot easily be measured and is calculated as the remainder of the other listed costs. We list machine configurations in Section A.

what	time [μsec]
send t_s	5.5
receive t_r	5.4
propagation t_p	∼100-1000

Table 2.2: Breakdown of UDP round-trip time [KRSS12]. The receive is based on polling. The propagation is orders of magnitude higher than processing time on cores. The numbers given are for a fast data center Ethernet network.

sages to all cores in such a subset while waiting for replies. As all requests can be sent while waiting for the first reply, the cost to apply an update is then one round independent of the numbers of cores n. As discussed earlier for the multicore case, messages cannot be processed in parallel. That means that the cost of acquiring the quorum is linear to the number of cores. Proactively sending messages does not make sense due to the per-message complexity metric. Note that the receive operations in this case can be executed by the receiving cores in parallel, but since messages are send sequentially, only once core has to send all messages and hence experiences a linearly growing accumulated send cost.

Another example is a serializer. Pre-Serialization has been shown to improve throughput and latency for quorum based protocols [SMDR08]. This is because of batching of concurrent request, which overlaps high propagation time. On multicores, a serializer will have the reverse effect. Since send time is dominating, it does not make sense to serialize sending messages on a single core instead of sending these requests from different cores in parallel.

In conclusion we observe that although multicore machines are distributed systems, they look different to classical ones. Our observation suggests that it does not make sense to measure the complexity of algorithms in rounds. Traditional complexity measures do not apply to multicore machines.

Instead of trying to provide a hand-tuned implementation for all of these diverse machines, our goal is to systematically construct a representation of the system's topology and make it available to applications and runtime.

Conclusion and Limitations

In this chapter, we described our machine model. It serves as a solid foundation of our machine-aware memory management (Chapter 4) and synchronization primitives (Chapter 5).

We designed our model such that it can be filled completely automatic by querying hardware registers to determine hardware's capabilities. Where this information is not sufficient, we enrich our model with the results of carefully crafted micro benchmarks.

Limitations Our model is currently restricted to a static system. We do not consider contention nor changes in application workload. We do not deal with machine failures at this point. If hardware fails, we treat this as a failure of the entire machine. Furthermore, we assume communication links to be reliable and in-order.

The level of detail for a machine model is a trade-off. With this work, our goal was to define a model that can be automatically generated by a parsing hardware information and a small number of micro benchmarks that capture machine-relevant details that are not available from hardware registers. We found our model to be detailed enough for this work, but anticipate that an even more fine-grained model — for example modeling the cache-coherence protocol— might be beneficial for other applications, or to achieve an even better performance.

3
Memory Access Patterns

Introduction

Memory access performance of parallel programs depends on the structure of the machine as captured by our machine model (Chapter 2) as well as characteristics of the workload executed by applications. In this chapter we now focus on how to characterize application characteristics, specifically the classification of memory access patterns. Based on that, we show in Chapter 4 how memory allocation can be optimized automatically based on the machine model and access patterns.

In Chapter 2, we discuss the increasing complexity and diversity of multicore hardware: multicores have multiple memory controllers, some even several of them per socket. Individual sockets are connected by an interconnect network, which exhibits many of the same properties as traditional networks do. On such machines, access latency and throughput vary depending on where memory is allocated and accessed from. Here, two problems are especially relevant:

Congestion on the interconnect If many cores in the system access memory on the same socket, interconnect links close to that socket are likely to suffer from congestion, which limits the memory access bandwidth available to each individual core and hence reduces application performance [Raz11, DFF+13].

Imbalance of memory controller If many of the memory accesses are executed on only on few of the memory controllers, the bandwidth of the unused ones is wasted.

Locality Memory should be allocated local to the data it is accessing. Otherwise, when accessing remote memory, access latency is higher and throughput lower on many machines.

Several approaches to mitigate these effects exists: The challenge then is to choose which strategy to apply for memory allocation. In addition to machine characteristics, the choice also depends on *access patterns*. The following is a selection of popular strategies for memory allocation:

Data distribution A simple initial strategy is to allocate memory randomly but approximately equally on controllers to achieve load-balancing for uniformly accessed data. However, equally distributed data can still cause non-uniform use of memory controllers if some data items are accessed more frequently than others (e.g. if they follow a power-law distribution).

Partitioning Distribute data items that semantically belong together are allocated close to each other. If combined with a scheduler, which is aware of the data partitioning, can additionally guarantee accesses to be local for some workloads.

Replication On top of that, replication achieves local accesses and load-balancing even for random accesses, but can cause a slowdown as a result of having to maintain consistency in the presence of updates.

To illustrate potential performance gains, Table 3.1 shows the runtime of Streamcluster as a example. Streamcluster is an online clustering algorithm from the PARSEC benchmark suite: if memory allocation is carefully optimized, performance improves by up to 4× compared to its naïve memory allocation that is oblivious to hardware characteristics. We present a more detailed analysis of our Streamcluster results in Section 4.6.1.

		machine	
		A IL 4x4x2	I SB 4x8x2
memory allocation	native/`malloc`	215.6	236.5
	optimized	51.8	66.4

Table 3.1: Execution time [seconds] of Streamcluster for the default and an optimized memory allocation. We list machine configurations in section A.

The choice of memory allocation strategies depends on an applications access patterns. Even worse, it might also depend on the input of the program itself: for example if an algorithm such as PageRank was to be executed on a power-law graph such as for web links [BKM+00], access patterns are different from a uniform graph. We show this in detail in Chapter 4, where we use access patterns to automatically determine which array allocation strategy to use for each data structure based on its access patterns.

Here, we discuss how they can be extracted. In some application domains, expert programmers already know these patterns for their programs: one

example are database developers. Unfortunately, in many cases programmers do not know memory access patterns of their applications. However, we observe that they are often implictly encoded in a program's source code. Although manually analyzing the source code to determine access patterns is possible in some cases, it can be tedious for large problems.

Ideally, the extraction should be done automatically to keep the life of programmers simple. For general purpose languages such as C/C++ or Java, it is hard to extract that information since a higher level meaning of a group of instructions is often hard to deduce from the low-level source code: pointer arithmetic, for examples, allows programs with almost arbitrarily complex to understand access patterns. This makes it hard to deduce any meaningful data from general purpose languages.

Domain specific languages

Memory access patterns can be automatically extracted for one important class of programs: domain specific languages. They are implictly encoded in a program's high-level description. In contrast to general-purpose languages such as C/C++ and Java, *domain specific languages (*DSL*s)* are designed with a specific application domain in mind, which allows their statements to be more expressive compared to general purpose languages.

When working with DSLs, algorithms written in a high-level language have to be translated to their low-level representation by a high-level compiler. The output of the translation can be a general-purpose language such as C/C++, machine-code, or even a different high-level language. The choice of output-language often depends on the target machine to execute the program on. The variety of backends is important due to the increasing diversity of hardware-platforms that have to be supported by programmers. Here, we focus on C/C++ as a backend and show the typical workflow in Figure 3.1.

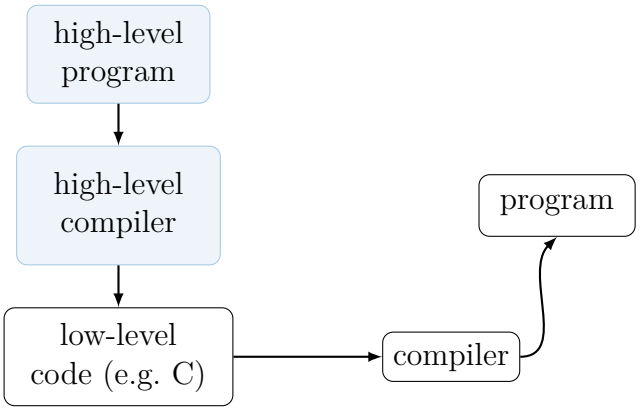


Figure 3.1: Execution of a program written in a DSL

As an example, we use Green-Marl [SLB+11, HCSO12], a graph analytics framework. Graph processing increasingly gains popularity as a result of a

growing demand for large-scale graph processing applications. Green-Marl is a domain specific language with an intuitive set of language constructs available to programmers. This is not only easier to program, but also presents opportunities for data-level parallelism not typically available in low-level general-purpose programming languages. Despite our focus on Green-Marl, we believe that our approach is applicable to other DSL's as well and given an intuition about other uses in Section 3.5.

As an example, Listing 3 shows PageRank for the Green-Marl graph analytics DSL. Pagerank attempts to iteratively calculate the importance of each node in a graph depending on the importance of neighboring nodes. It executes iteratively until the ranks stabilize or a maximum number of iterations has been executed. This shows up as the do-while loop in lines 8 to 19. It then iteratively calculates each node's rank by averaging its neighbors rank divided by the neighbor's outdegree (lines 11 to 13). This is implemented with the node iterator `Forach (t: G.Nodes)` iterating over all nodes of the graph (line 10) and `t.InNbrs` to iterate over that node's neighbors (line 12). Note also how node properties such as `pg_rank` can easily be accessed in a way that does not restrict the implementation of the underlying data structure for node properties. This example shows, how Green-Marl's graph-specific set of statements allows programmers to write a clean, readable, and short implementation of a complex algorithm.

Note how the Green-Marl programming paradigm defines a more relaxed consistency model: writes from the the loop body *must* not be visible until the next iteration of the loop. This motivates the use of double buffering: one copy of the data is uses for reading in the current loop iteration with updated values being written to a copy of it. This permits an arbitrary order in which elements of the `Foreach` statement are executed and allows the compiler to generate efficient parallel code.

In contrast, a general purpose low-level compiler such as `gcc` is not allowed to perform optimizations like that as it has to provide generality across a wide set of programs and has to assume that the order of execution matters for a concrete program.

Our observation and the reason for us to look into domain specific languages is that they encode memory access patterns in a way that lets us extract them from the input program with relatively simple modifications to the language's high-level compiler.

If memory access patterns are known, they can be used — together with machine model encoding machine characteristics as described in Chapter 2 — to automatically tune memory allocation at runtime. Together with the memory abstraction needed to allow dynamic selection of the implementation for memory access methods, we explain our runtime in Chapter 4.

In this chapter, after discussing related work in Section 3.2, we first given an overview of the types of memory accesses we consider in this work (Section 3.3). In Section 3.4, we then show to apply our approach to the Green-Marl graph analytics framework.

Listing 3 PageRank written in Green-Marl

```
Procedure pagerank(G: Graph, e,d: Double, max: Int;
                   pg_rank: Node_Prop<Double>)
{
    Double diff;
    Int cnt = 0;
    Double N = G.NumNodes();
    G.pg_rank = 1 / N;
    Do {
        diff = 0.0;
        Foreach (t: G.Nodes) {
            Double val = (1-d) / N + d*
                Sum(w: t.InNbrs) {
                    w.pg_rank / w.OutDegree()} ;

            diff += | val - t.pg_rank |;
            t.pg_rank <= val @ t;
        }
        cnt++;
    } While ((diff > e) && (cnt < max));
}
```

Related work

Domain specific languages

Domain specific languages such as Green-Marl [HCSO12], a graph analytics framework, and OptiML [SLB+11], a machine learning DSL, provide a rich set of powerful high-level operators. They use a compiler that generates code that is highly tuned to the target machine and makes heavy use of data parallelism. While all of these languages encode memory access patterns in their high-level languages, none of them uses this information at runtime to optimize memory allocation.

Spiral [XJJP01, PMJ+05] aims to research automatic generation of digital signal processing (DSP) algorithms and other numerical kernels. The output is an entirely autonomously generated platform-tuned implementation of algorithms such as discrete Fourier transformations, discrete cosine transformations and others.

DSLs are known for their ease of programming since their semantics closely match a specific application domain and the set of statements offered by DSLs is on a higher level compared to classical general-purpose programming languages such as C/C++ or Java. In addition to the ease of programming, applications written in domain-specific languages allow good automatic parallelization and algorithm-specific optimizations.

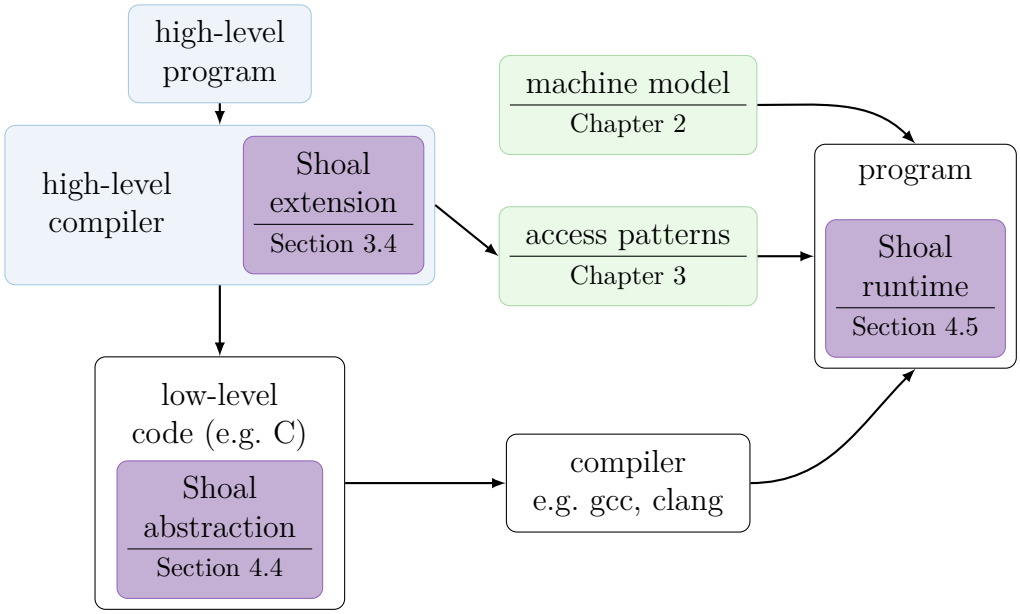


Figure 3.2: System overview

Access patterns

Tracking memory access patterns is not a new idea. Others propose to detect them online rather than executing a static analysis of source code offline. Carrefour [DFF+13] provides a kernel module that analyses data from performance counters measuring memory accesses and uses this information to infer programmer's intentions from that. LAPT [CDA+14] suggests additions to page tables entries to track memory accesses.

Popcorn Linux [BSA+15] also uses a model of memory access patterns per function call to decide whether or not two functions should be scheduled on the same kernel instance as a consequence of frequent data exchange on shared memory pages. They analyze the memory access patterns at runtime using instrumentation code that is automatically inserted by an LLVM extension.

In contrast to all of these, we observe that access patterns are frequently already implicitly encoded in high-level languages. Rather than loosing this information in the compilation process, we propose to collect them with a small set of extensions to the compiler.

Malicevic *et al.* [MDS+15] explore the use of non-volatile memory (NVM) for graph processing. NVM provides large byte-addressable memory at higher latency and lower bandwidth compared to traditional DRAM memory. In order to mitigate resulting performance penalties for graph processing workloads, they investigate a hybrid memory system, where the most frequently accessed data structures are allocated in DRAM while less important regions of the working set can be stored in NVM. This confirms the importance of determining data access patterns. However, the decision on where data is

allocated in that case is done manually by programmers rather than static analysis of the graph data.

Classification of memory accesses

We now given an overview of memory access patterns and explain their implications in terms of memory allocation and access. What is important for choosing a good memory allocation strategy is the ratio of read- to write accesses, but also the the access patterns. As we will show later in Chapter 4, certain memory access patterns are especially important. When a classification to either of them is available at runtime, the memory allocation can be tuned accordingly.

Here, we introduce the these memory access patterns and show later in this chapter how we can automatically classify which of them applied for a certain region of memory.

Access pattern

We now give an overview of the types of access patterns we are considering for this work. We will show in Chapter 4 classifying memory accesses as one of these patterns is sufficient to automatically choose memory allocations for a Green-Marl and Streamcluster such that the modified version of these programs significantly outperforms the original version of both workloads.

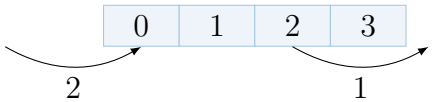
Beyond optimizing memory allocation, access patterns can also be used to optimize scheduling. Work items can then be scheduled to threads being pinned close to memory controllers storing the data it accesses.

Note that even if work is dynamically split up into chunks, the access patterns as discussed here are still valid, e.g. a data structure with sequential accesses still has sequential access within each chunks.

Sequential access

Sequential data accesses are widely used in parallel programs. It can be skewed or strided, which means that locality is not necessarily given if the same array is accessed from various loops, with different skews and strides.

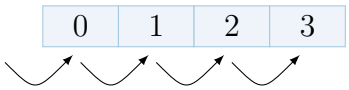
```
#pragma openmp parallel for
#define LEN 1000
for (int i=0; i<LEN; i+=2) {
  int j = (i+LEN/2) % LEN
  do_something(&data[j]); // skewed
}
```



Indexed access

Indexed accesses are based on the observation that a common pattern in parallel programs is to iterate over an region of memory in sequential order with a stride of 1 from the beginning to the end of a region of memory.

```
#pragma openmp parallel for
for (int i=0; i<len(data); i++) {
  do_something(&data[i]); // indexed
}
```



In this example, access to memory of array `data` is what we call *indexed*. Every element is accessed only once in sequential order and the access provides locality.

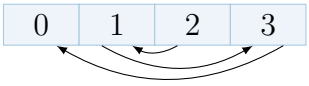
While this type of access pattern seems narrow and specific, it turns out that it is widely used for programs based on parallel loops (such as OpenMP), where the index variable of a for-loop is also used as an array-index. As the level of parallelization in the case of parallel loops are batches of loop iterations and access is sequential and local, this access pattern is a natural fit for partitioning.

Random access

We classify everything else as random access. Random access is problematic. Since accesses do not follow any logic, hardware cannot prefetch data, which leads to CPUs stalling and waiting for new data to arrive from memory. Due to the lack of locality, caching and replication also do not help to hide high memory access latency.

A data structure that is inherently suffering from random accesses is a linked list, where pointers have to be dereferenced to find the next entry of the list. Additionally, many algorithms suffer the same problem even if the data structure itself is linear as with arrays, for example the neighbor relationship in many graphs.

```
#pragma openmp parallel for
void *element;
while (element);
  do_something(element->data);
  element = element->next;
}
```



Case Study: Green-Marl

Green-Marl is a domain-specific language for graph processing. As a proof of concept, we modified the high-level compiler of Green-Marl to show how

memory access patterns can be extracted with moderate changes.

In contrast to other DSL compilers, Green-Marl has a fairly simple toolchain and is compact enough to allow immediate modifications. OptiML, for example, has a complex toolchain based on Scala [OR14] to build the compiler. Modifications to OptiML's logic are complex and rebuilding the compiler is tedious. In contrast, Green-Marl's compiler for translating high-level Green-Marl programs is written in C/C++. Analyzing the code and tracking memory access patterns could either be done in the backend-independent lexer, or directly in the backend. Since modifications directly in the C/C++ backends seemed simpler than in the backend-independent lexer, we decided to apply our modifications directly there.

As other DSLs, Green-Marl has a small set of application specific statements. This makes it a perfect fit for extracting memory access patterns: only a comparably limited number of annotations are necessary in the Green-Marl compiler to track memory accesses.

In addition to PageRank (Figure 3), we also use triangle-counting and hop-distance as examples for graph workloads. We show the code for both of them in Figures 4 and 5.

In this section, we describe our modifications to the Green-Marl compiler in detail. In Section 3.4.1, we starts with a detailed analysis of Green-Marl's internal data layout for the graph. We then describe how we annotate Green-Marl statements with their memory access patterns (Section 3.4.3) and how we track loops to determine the access frequency for each of these statements (Section 3.4.2). In Section 3.4.4 and 3.4.5 we list the result of our modifications for three graph algorithms, and its correctness. Finally in Section 3.5, we give an intuition how similar ideas could be used to modified other domain specific languages.

Graph storage

Green-Marl stores graphs internally in the compressed sparse row format (CSR). Figure 3.3 shows the CSR representation of a small sample graph.

The CSR format is used as a compact representation of a graph, especially sparse ones. The core idea is to group edges by their source node and store them contiguously in an array `node_idx`. As the name suggests, this array stores the node index of vertices. Since edges are grouped by source node, only the destination of an edge has to be stored. In order to find the range of entries in `node_idx` representing edges starting in a node n, the index of the first edge of node n's batch is needed. This is stored in array `begin`. The last of the range of destination identifiers for a node n's edges can be calculated from looking up its successor node's start index.

For example, node 1 in the sample graph displayed in Figure 3.3 has two outgoing edge to nodes 0 and 2. `node_idx` hence must have two entries for node 1, namely 0 and 2, the destination ids of the these edges. `begin[1]` then must point to the relevant index in the `node_idx` array, which is 1 in this case.

Listing 4 hop-distance written in Green-Marl

```
// This routine is, in fact, BFS
Procedure hop_dist(G:Graph, dist:N_P<Int>, root: Node)
{
    N_P<Bool> updated;
    N_P<Bool> updated_nxt;
    N_P<Int>  dist_nxt;
    Bool fin = False;

    G.dist = (G == root) ? 0 : +INF;
    G.updated = (G == root) ? True: False;
    G.dist_nxt = G.dist;
    G.updated_nxt = G.updated;

    While(!fin) {
        fin = True;

        Foreach(n: G.Nodes)(n.updated) {
            Foreach(s: n.Nbrs) {
                // updated_nxt becomes true only if dist_nxt is actually updated
                <s.dist_nxt; s.updated_nxt> min= <n.dist + 1; True>;
            }
        }

        G.dist = G.dist_nxt;
        G.updated = G.updated_nxt;
        G.updated_nxt = False;
        fin = ! Exist(n: G.Nodes){n.updated};
    }
}
```

A lookup of all edges of 1 would then return elements `node_idx[begin[n]]` to `node_idx[begin[n+1]]-1`, which with a node index of $n = 1$ translates to elements `node_idx[begin[1]]` to `node_idx[begin[2]]-1`. This results in 0 and 2, which are the destination node indices of 1's outgoing edges. As can be seen from above code, a guard element is needed pointing to the end of array `node_idx`. In our case, this is element 5 visualized with a dashed border.

For an efficient lookup of incoming edges, Green-Marl additionally implements a reverse lookup with two more arrays implementing the same functionality for each edges inverse. Reverse edges are stored in (`r_begin` and `r_node_idx`).

Loops

For a detailed understanding of access patterns, it is important to keep track of loops. In order to analyze how often data is accessed, each loop iteration along with the loop's index variable needs to be stored. For many algorithms,

Listing 5 triangle-counting written in Green-Marl

```
Procedure triangle_counting(G: Graph): Long
{
    // undirected version
    Long T;
    Foreach(v: G.Nodes)
        Foreach(u: v.Nbrs) (u > v) {
            Foreach(w: v.Nbrs) (w > u) {
                If ((w.HasEdgeTo(u)))
                    T += 1;
            }
        }
    Return T;
}
```

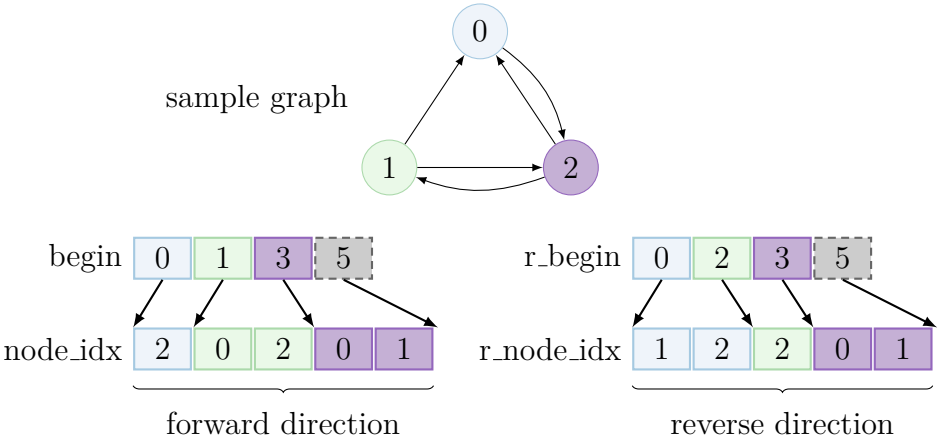


Figure 3.3: CSR representation for a small sample graph

the number of loop iterations is either constant, or depends on the program input, e.g. the number of nodes and edges in the currently loaded graph.

Keeping track of loops is crucial to determine memory access patterns. We do so using a stack data structure and, whenever the Green-Marl compiler generates a loop, we push a new entry on this stack and pop it again once the loop body finishes. The following code section describes which relevant functions of Green-Marl's compiler:

```
// Generate for-loop over Green-Marl data structures
gm_cpp_gen::generate_sent_foreach(ast_foreach* f);

// Generate constant size while-loops
gm_code_generator::generate_sent_while(ast_while *w)
```

`generate_sent_while` is used for example in PageRank to generate a while-loop that executes PageRank until the changes in the ranks are smaller than what is given by the programmer. For `generate_sent_foreach`, we support currently the following types of loops:

- `LOOP_NODES`: an iterator over all nodes in a graph
- `LOOP_NBS`: an iterator walking over all neighbors of a node
- `LOOP_CONSTANT`: constant number of iterations of a fixed-size loop

Other loop types offered in Green-Marl are not yet supported, since our workloads do not make use of them. However, they could be trivially pushed to our stack data structure as well.

A stack data structure is needed for nesting of loops. In some cases, two nested loops need to be combined in the function stack. For example: an iteration over all nodes (`LOOP_NODES`) followed by an iteration over their neighbors (`LOOP_NBS`) represents an iteration over all edges. Hence, whenever the stack of loop iterators contains a sequence of these two loop iterators, we thread this as an iteration over all edges of the graph (E). An example of this are lines 17 and 18 in Listing 4 and lines 5 and 6 in Listing 5.

We also track the index variable of each loop in a list. For each access to an array, we then check if the variable used to access the array is used as an index by looking it up in the list. If so, the access is an indexed access.

Memory access

Our goal is to capture array access patterns. Here, we are now discussing which parts of the Green-Marl compiler we had to modify to track memory access patterns. We will first discuss some helper functions we implemented to track memory access patterns, followed y a list of Green-Marl statements we needed to modify.

Helper functions to track memory accesses To facilitate tracking array accesses, we need to store information about each memory reference generated by the compiler internally. For that, we provide the following functions:

```
/** Track an access to one of the arrays */
void record_array_access(const char* array_name,
                         bool is_indexed, bool is_write);

/** Generate array access and track them.  */
static char* array_access_gen(const char* array_name, const char* index,
                 std::string original_array);

/** Generate and print array access; Internally calls array_access_gen. */
void array_access(gm_code_writer* Body, const char* array_name,
                  const char* index, std::string original_array);
```

Access to arrays Based on our description of Green-Marl's internally memory layout (Section 3.4.1), we are now describing our annotations of Green-Marl's high-level constructs. Note that the data layout might change in the future or even look different depending on the architecture. With our approach, this is not a problem, as access patterns are extracted with the internal data representation in mind. In the case of having several representations at hand, each combination of high-level operation and data representation would have to be annotated separately.

We track accessed on a per-array basis. We modified the Green-Marl compiler to record each access to each array along with the the type of array access. This is in contrast to what the high-level compiler would normally do: when compiling the high-level program down to its low-level implementation, this information is normally lost. The following functions generated code that performs a memory access:

```
/** Assignment */
void gm_cpp_gen::generate_sent_assign(ast_assign* a);
void gm_cpp_gen::generate_sent_reduce_argmin_assign(ast_assign *a);

/** LHS */
void gm_cpp_gen::generate_lhs_field(ast_field* f);
```

Memory accesses can also be generated from other functions, but the Green-Marl code generating the memory reference is still evaluated from function `generate_lhs_field`, which parses the given code and potentially generates memory accesses out of that.

LHS stands for left-hand side, which is a bit of a misnomer, since the same function is also used to evaluate the right-hand side of a statement. To distinguish read- from write-accesses, we have to keep track of whether we are evaluating the right- or left-hand side of a statement. We do this with variable `bool sk_lhs`. LHS statements are invoked from within `generate_sent_assign` and `generate_sent_reduce_argmin_assign`. The first is used to generate simple assignments, while the latter is used for generating low-level code for `min=`, as used in hop-distance (Listing 4, line 20).

```
/** Code generator for build-in functions. */
void gm_cpplib::generate_expr_builtin(ast_expr_builtin* e,
                                      gm_code_writer& Body);
```

Green-Marl's `generate_expr_buildin` is where most of Green-Marl's graph-specific operations are implemented. The following is a list that describes the access patterns as detected by our modifications to the Green-Marl compiler.

- `GM_BLTIN_NODE_HAS_EDGE_TO`: Determines if a node has an edge to another node. This functionality is used in triangle-counting.
  - `node_idx`: read-only, non-indexed

  - `begin`: read-only, non-indexed

- `GM_BLTIN_NODE_DEGREE`: Determines the node degree, i.e. the number of outgoing edges of a node. For that, `begin` has to be accessed twice and the difference between the indices calculated. This gives the number of entries in the `node_idx` array associated with a source node id, which equals the number of outgoing edges of that node.

  - `begin`: twice, read-only, non-indexed

- `generate_foreach_header` as `LOOP_NBS`: Generates a for-loop that runs over neighbors. The iterator variable is of `EDGE_T`. This is used twice in triangle-counting, and once each in PageRank and hop-distance. Since it iterates over the neighbors, array `begin` is accessed twice to determine start and end index in `node_idx`.

  - `begin`, read-only, indexed
  - `begin`, read-only, non-indexed

- `generate_foreach_header` as `LOOP_NODES`: For iterating over all nodes, no memory has to be accessed. However, the loop has to be tracked and its type added to the list of active loop iterations, so that array accesses from within the loop can be recorded with the appropriate number.

We did not have to instrument the following Green-Marl operators, since they have not been used in any of the benchmark we were executing.

- `GM_BLTIN_NODE_IN_DEGREE`: This is the same as `GM_BLTIN_NODE_DEGREE` except for accessing `r_node_idx` instead of `node_idx`.
- `GM_BLTIN_NODE_IS_NBR_FROM`
- `GM_BLTIN_NODE_RAND_NBR`
- `GMTYPE_NODE_ITERATOR`

Declaration of a variable As described before in Section 3.4.1, Green-Marl stores the graph in a total of four arrays. These are always generated with the same name and type. Their size depends on the input graph, but that is already known from the header at the beginning of the file storing the graph.

Additionally, Green-Marl might need additional per-node or per-edge arrays for some workloads, for example for ranks in PageRank. This is why we need to know about each variable-declaration in the system and hence intercept `generate_sent_vardecl`, which generates a variable declaration:

```
/** Variable declaration - need to encapsulate state. */
void gm_cpp_gen::generate_sent_vardecl(ast_vardecl* v);

/** Generate an algorithm's main function. */
void gm_cpp_gen::generate_proc_decl(ast_procdef* proc, bool is_body_file);
```

In order to avoid redundant work in initializing arrays, we also detects how arrays should be initialized. In many cases, the initial state of the array is irrelevant (for example: the double-buffered second copy of PageRank's ranks array).

Furthermore, we instrument `generate_proc_decl`, the function that generates an algorithm's main method. For each variable, we need to know if values are passed from outside this algorithm, or if the array has to pass it back to the caller after execution of the main function. This helps to avoid unnecessary additional copy operations.

Detailed access patterns

In this section, we will show the result of extracted memory patterns for PageRank, a well-known algorithm to determine the importance of each node in a graph as well as triangle-counting and hop-distance. The output shown here is automatically generated by our modifications to the Green-Marl high-level compiler.

We now describe for each workload the arrays used in it and list some of their properties in Table 3.2. All of these properties are automatically extracted by our modifications to the Green-Marl high-level compiler. The table shows:

node Whether an array is a Green-Marl per-node array.

edge Whether an array is a Green-Marl per-edge array.

used Whether Green-Marl's static arrays storing the graph are accessed by this workload: in some cases, Green-Marl allocates an array, but does not actually use it.

ro Whether it is read-only.

std Whether it is one of the Green-Marl standard arrays storing the graph itself: these are filled by the Green-Marl runtime with the graph loaded from disk and have to be copied from there.

dyn Whether arrays are dynamically created from within the algorithm's main function, in which case they do not explicitly have to be initialized. If they need to be initialized to a specific value, it has to be specified explicitly as part of the Green-Marl high-level code. These accesses are then already covered by our modifications to the compiler and such arrays consequently do not have to be tracked otherwise.

indexed Whether the array is always accessed via an index variable.

array	node	edge	used	ro	std	dyn	indexed
Green-Marl arrays for PageRank							
`begin`	X		X	X	X		
`node_idx`		X					
`pg_rank`	X		X				
`pg_rank_nxt`	X		X			X	X
`r_begin`	X		X	X	X		
`r_node_idx`		X	X	X	X		
Green-Marl arrays for hop-distance							
`begin`	X		X	X	X		
`dist`	X		X				X
`dist_nxt`	X		X			X	
`node_idx`		X	X	X	X		
`r_begin`	X						
`r_node_idx`		X					
`updated`	X		X			X	X
`updated_nxt`	X		X			X	
Green-Marl arrays for triangle-counting							
`begin`	X		X	X	X		
`node_idx`		X	X	X	X		
`r_begin`	X						
`r_node_idx`		X					

Table 3.2: Extracted array properties for PageRank, hop-distance and triangle-counting

Table 3.3 shows the access cost of all used arrays in our evaluation. N is the number of nodes and E the number of edges in the graph. k is an workload-specific constant, often representing the number of iterations of the main loop until the algorithm converges. This constant is normally small compared to the size of the graph and often equally multiplying the otherwise workload-specific number of accesses. In terms of prioritizing arrays based on their relevance it can then be ignored in such cases.

For example, the number of reads in Green-Marl's PageRank rank array is: $kE * kN$, where E is the number of edges and N is the number of nodes. While, we derives these formulas and have them readily available, so far we found the more coarse-grained results as shown in Table 3.2 sufficient in the context of this work.

Note that for simplicity, Table 3.3 does not show a breakdown of array accesses to its type. For example, it does not show how many indexed array accesses are going to be executed for a certain workload. If hat information was required, it could be trivially implemented in our modifications to the

array	#reads	#writes
Green-Marl arrays for PageRank		
`pg_rank`	$kE + kN$	$N + kN$
`r_begin`	$kN + kN$	0
`r_node_idx`	kE	0
`begin`	$kE + kE$	0
`pg_rank_nxt`	kN	kN
Green-Marl arrays for hop-distance		
`dist`	$N + kE$	$N + kN$
`updated`	$N + kN + kN$	$N + kN$
`dist_nxt`	$kE + kE + kN$	$N + kE$
`updated_nxt`	kN	$N + kE + kN$
`begin`	$kN + kN$	0
`node_idx`	kE	0
Green-Marl arrays for triangle-counting		
`begin`	$N + N + E + E + EE$	0
`node_idx`	$E + EE + EE$	0

Table 3.3: Array cost functions as automatically extracted by our modifications to the Green-Marl compiler. Here, we exclude arrays that are are not used by Green-Marl.

Green-Marl compiler, since all information required to track that information is already part of the code.

Correctness

We tested the correctness of our modifications indirectly via execution of PageRank, hop-distance, triangle-counting on two graphs. For each array, we dump the CRC checksum of each array before and after execution of the algorithm.

The checksum would change if:

- the prediction of read-only access would be wrong: in that case, a write-access would not work as expected with replication, as only a thread's local replica would be updated with others being unchanged. This would show up in a different output of the algorithm and a change in the CRC code for each array.

- if one of the memory accesses would still be direct instead of redirected via our additional memory abstraction — whenever we changed the code to used the memory abstraction, we also annotated the memory access with its array pattern. From that, we conclude that we have be annotating all of Green-Marl's memory accesses correctly as shown by the results in Table 3.3.

Examples of other languages

While not implemented in other languages, we would like to analyze some other examples of where memory access patterns could be extracted.

Databases

In databases management systems (DBMS) or systems like LINQ [MBB06], there is typically only a limited number of database queries. In that case, queries can be analyzed to figure out memory access patterns.

Here, we want to give an intuition on how well this works. For example:

```
SELECT * FROM students WHERE age>30
```

From that, we can see that we have an indexed access on the students relation, which makes it an ideal fit for a partitioned working set.

Machine learning

OptiML [SLB+11] is a machine learning domain-specific language. Due to the complexity of the toolchain, we did not consider it for updating the high-level compiler for automatic annotation of the program. However, here, we give an intuition about the feasibility of our approach also in the context of machine learning.

Common workload types in machine learning are images, training sets, which are internally represented as vectors, matrices and indexed graphs, but also views implemented as sub-matrices.

Computation then often boils down to dot products between matrices, linear algebra executed on vectors and matrices. Frequent operations are map, count, filter. However, higher-level languages such as OptiML also support rich operations such as `histogram(Image)`, which in its current implementation writes to a shared data structure.

Operator fusing OptiML employs Operator fusing. For example, it knows that a vector + vector operation is equivalent to a `ZipWith` operation, i.e. a simultaneous loop over two vectors.

Conclusion

In this chapter, we showed how detailed memory access patterns can be automatically extracted from high-level programs in the field of domain specific languages. Combined with the machine model introduced in Chapter 2, memory allocation of parallel programs can be tuned automatically without programmers having to understand the characteristics of complex multicore

hardware and their implications on algorithm performance. We show our approach to smart memory allocation in Chapter 4.

Limitations Our access patterns are generated by a static code analysis. Consequently, our results do not consider dynamic aspects such as load caused by other processes running on the system. For that, online approaches like Carrefour are a better match. However, static analysis is still beneficial to find a good initial placement.

4
Memory management

Introduction

Memory allocation in NUMA multicore machines is increasingly complex. Good placement of and access to program data is crucial for application performance, and, if not carefully done, can significantly harm scalability [ASK+07, DFF+13].

For example, Figure 4.1 visualizes a single node memory allocation: the entire working set of the application is allocated on just one NUMA node. Concurrent access to that memory leads to two problems: (*i*) only one memory controller is used, leading to a waste of additional memory bandwidth from all others, and (*ii*) contention on the interconnect close to where memory is allocated.

Techniques such as replication and partitioning of data across NUMA nodes exists and can retain scalability (e.g. [BBD+09, ASK+07]). However as we show later, many programmers struggle to develop software applying these techniques. The problem is that it is unclear which techniques to apply in which situation. What is worse is that with rapidly evolving and diversifying hardware, programmers must repeatedly make manual changes to their software to keep up with new hardware performance properties. Beyond simple placement of data, ideas and techniques from traditional distributed systems like replication and partitioning can help to improve memory management [SSGA11, ASK+07]. The challenge is to decide when to apply which of these strategies. The choice depends on machine and program characteristics.

One solution to automatically and dynamically decide which techniques to use is based on online monitoring of program performance, for example as in Carrefour [DFF+13]. However, monitoring may be expensive. Firstly, due to missing hardware support if pages must be unmapped to trigger a fault when data is accessed. Secondly, our results suggest that they are insuffi-

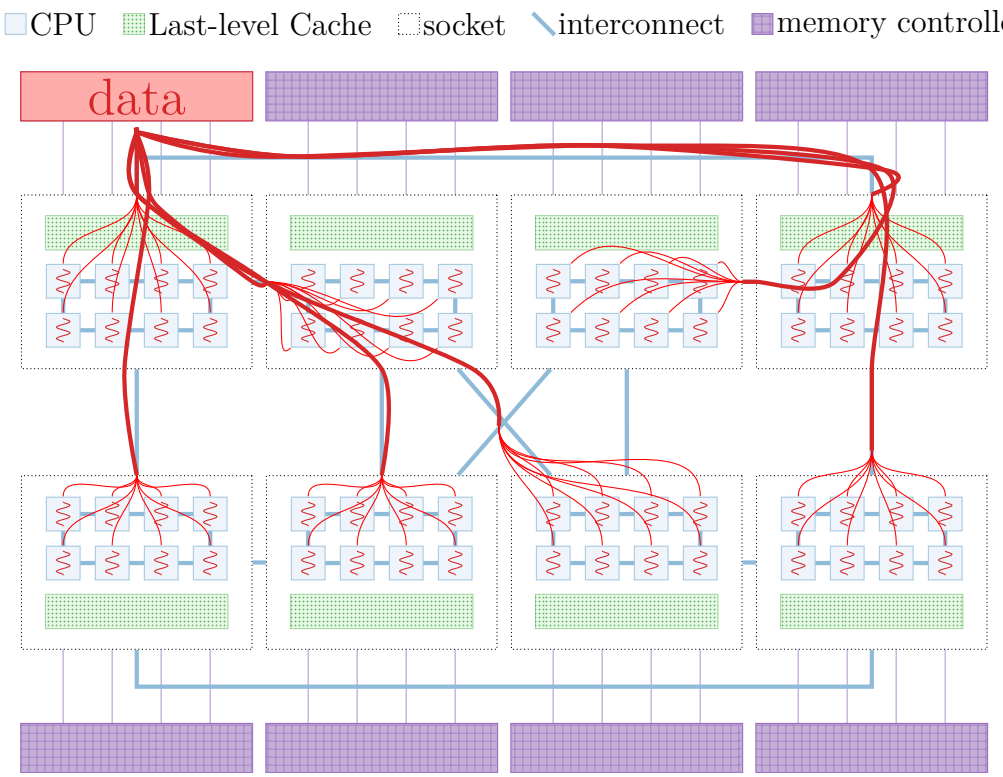


Figure 4.1: Contention with single node allocation on multicores

ciently precise if statistically measured based on sampling using performance counters. Both approaches are limited to a relatively small number of optimizations. For example, it is hard to incrementally activate large pages or switch to using DMA hardware for data copies based on monitoring or event counters due to a lack of proper programming interfaces. Worse, online approaches have to guess programmer's intentions based on these statistically observed measurements, and the wrong choice of optimizations might be inefficient or even reduce the performance (see Section 4.6.1 for an example).

In this chapter, we present Shoal, a system that abstracts memory access and provides a rich programming interface that accepts hints on memory access patterns at runtime. These hints can either be manually specified by programmers or automatically derived from high-level descriptions of parallel programs such as domain specific languages as described in Chapter 3.

Shoal includes a machine-aware runtime that selects the implementation for this memory abstraction dynamically during buffer allocation based on the hints for a concrete combination of machine and workload. If available, Shoal is able to exploit not only NUMA properties but also hardware features such as large pages and DMA copy engines.

Our contributions in this chapter are the following:

- a memory abstraction based on arrays that decouples data access from

the rest of the program together with an interface for programs to specify memory access patterns when allocating memory,

- a runtime that picks and configures one of several highly tuned array implementations based on access patterns and machine characteristics and can exploit machine specific hardware features.

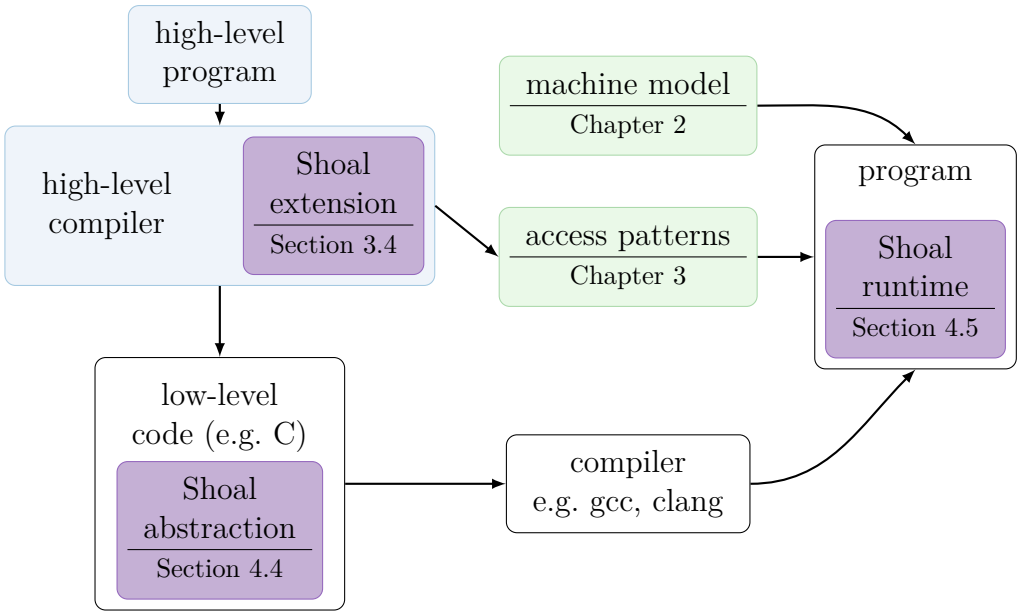


Figure 4.2: Shoal system overview

We visualize this in Figure 4.2. The high-level program, high-level compiler and details about how we extract memory access patterns have already been discussed in Chapter 3. Chapter 2 describes the machine model that is fed into the Shoal runtime.

In this chapter, we first give an overview over the state-of-the-art in Section 4.2 and related work (Section 4.3). We then describe the remaining components depicted in Figure 4.2: first, our memory abstraction and the programming interface of Shoal in Section 4.4 followed by the runtime in Section 4.5, the remaining components depicted in Figure 4.2.

As a result, Shoal allows programmers to write programs that achieve good performance without having to understand machine characteristics and needing to constantly rewrite applications in order to keep up with hardware changes. We demonstrate Shoal using the Green-Marl graph DSL, and the Streamcluster low-level C++ program from the PARSEC benchmark suite and present our results in Section 4.6.

Background

Here, we argue that programmers take little care of where memory is allocated and how it is accessed. In cases like the popular Streamcluster benchmark (evaluated in Section 4.6.1) and applications from the NAS benchmark suite [DFF+13], memory is allocated using a low-level `malloc`-call, which provides no guarantees about where memory is allocated or other details such as the page size that is going to be used.

For example, Linux currently employs a first-touch memory allocation strategy. Memory is not allocated directly when calling `malloc`, but mapped only when the corresponding memory is first accessed by a thread. The resulting page fault causes Linux to back the faulting page from the NUMA node of the faulting core.

A surprising consequence of this choice is that on Linux the implementation of the initialization phase of a program is often critical to its memory performance, even through programmers rarely consider initialization as a candidate for heavy optimization, since it almost never seems to matter for the total execution time of the program. To understand why, consider `memset`, a widely used approach for initializing the elements of an array. Most programmers will spend little time evaluating alternatives, since the time spent in the initialization phase is usually negligible. An example is as follows:

```
// Initialization (sequential)
void *ptr = malloc(ARRSIZE);
memset(ptr, 0, ARRSIZE);

// Work (parallel, highly optimized)
execute_work_in_parallel();
```

The scalability of a program written this way can be limited. `memset` executes on a single core and so all memory, if not accessed before the call to `memset` is allocated on the NUMA node of that core (Figure 4.1). For memory-bound parallel programs, one memory controller will be saturated quickly while others remain idle since all threads (up to 64 hardware contexts on the machines we use for evaluation) request memory from the same controller. Furthermore, the interconnect close to this memory controller will be more susceptible to congestion.

There are two problems here:

- Memory is not allocated (or mapped) *when* the interface suggests. Instead of allocating memory inside `malloc` itself, this happens lazily later in the execution.
- The choice of where to allocate memory is made in a subsystem that has no knowledge of the intended access patterns of this memory. In this case, the decision is made in the OS kernel.

These problems can be addressed by tuning algorithms to specific operating systems. For example, on Linux we could initialize memory using a parallel for loop:

```
// Initialization (parallel)
void *ptr = malloc(ARRSIZE);
#pragma omp parallel for
for (int i=0; i<ARRSIZE; i++) {
  init(i);
}

// Work (parallel, highly optimized)
execute_work_in_parallel();
```

This will be faster and in some cases retain scalability in current versions of the Linux operating system. It is for example implemented in Green-Marl. Leveraging the first-touch strategy this strategy aims to spread out memory across all memory controllers approximately equally, which balances the load on them and reduces contention on individual interconnect links.

However, there are several problems with it: First, above code might experience bad interactions with the page size if all threads are accessing memory on the same page concurrently, and then moving on to the next page simultaneously. A better strategy would be to access each page only once, but this requires explicit knowledge of the page size allocated by the operating system. Furthermore, it also requires correct setup of OpenMP's CPU affinity to ensure that all cores participate in this parallel initialization phase in order to spread memory equally on all memory controllers. Another drawback of this strategy is the loss of portability and scalability when the OS kernel's internal memory allocation policies change. Finally, it does not work well for non-uniformly accessed memory, as not every section of memory receives equally many accesses and the random distribution prevents the scheduler from considering access locality when allocating threads. Consequently, distributing memory as described above is a good starting point for optimizations, but puts unnecessary work on programmers and only work if programmers have a basic understanding of hardware. This examples visualizes that memory allocation is non-trivial and motivates our goal to assist programmers by choosing the memory allocation strategy automatically

Related work

Memory optimizations

Modern machines are becoming inherently complex: Baumann *et al.* argued that computers are already a distributed systems [BPS+09]. They proposed a multikernel approach [BBD+09] which avoids sharing of state among OS

nodes by replication and applying techniques from distributed systems. Similarly, Wentzlaff *et al.* [WA09] apply partitioning to OS services.

SMMP OS [ASK+07] observes that locality of data is as important as eliminating locks and reducing lock contention. They propose to selectively partition, distribute and replicate data. In contrast to Shoal using them for bulk data, SMMP OS applies these to object-oriented data structures as provided by K42 [KAR+06]. They are applied selectively depending on where distributed objects are accessed from rather than inferring access patterns a priori. Furthermore, converting lock based algorithms to their distributed object-based counterparts can induce an additional burden to programmers. Lastly, since objects are visible to programmers, the decomposition of algorithms can make it harder for programmers to understand the overall functionality of a program.

Techniques from distributed systems are beneficial not only on an OS level, but also for applications. Multimed [SSGA11] replicates database instances within a single multicore machine. Salomie *et al.* showed that congestion of memory controllers and interconnects impact the overall performance.

Cache coherence protocols like MOESI [Adv13] allow cache-lines to be in a shared state which is a form of hardware-level replication. This is only effective with workloads having good locality and a small working set and is less effective for workloads experiencing random accesses such as graph workloads. Research systems, such as Stanford FLASH, have provided software control over this form of replication [SHV+98].

Abstractions

PGAS languages such as UPC [UPC13], co-array FORTRAN [NR98], X10 [CGS+05], Chapel [CCZ07], and Fortress [Ora15b] provide an abstraction of shared arrays which can be implemented across a distributed system. Code iterating over an array can execute on the node holding the portion of the array being accessed.

In high-performance computing, array abstractions [NHL96] have been used to simplify programming while still providing good performance and scalability. They often support high-level operations on arrays e.g. matrix multiplications or atomic operators.

OpLog [BWKMZ14] provides a data structure for update-intensive (but rarely read) data that is machine aware and scales well. It logs updates in a per-core log. The key idea is to defer updates until the global state is read. Similarly, we also introduce a new data structure that is machine aware, but we provide several implementations that can be selected automatically based on application demands.

Wang *et al.* [WSP16] also proposes an abstraction, ParSec, for data structures in the context of real-time operating systems. They focus on scalable concurrent operations when accessing the data structure via a given interface.

Interfaces

The Solaris operating system provides a `madvise` [Ora15a] operation to let an application give hints on future accesses to a memory region which results in a distributed or local allocation to the calling thread. Linux also offers an interface for fine grain memory allocation, `libnuma` [Sil15a]. Shoal extends these approaches with a wider range of possible usage patterns, adds an easier to use higher-level interface on top of that, and infers appropriate settings to use automatically.

Access to large and huge MMU pages is provided by services and libraries like `libhugetlbfs` [ALM15]. The latter, however, requires static setup of a large page pool, among other issues.

Online tuning of memory allocation

Carrefour [DFF+13, GLF+15] attempts to reduce the contention on interconnect and memory controllers by online monitoring of memory accesses and auto-tuning NUMA-aware memory allocation. Carrefour focuses on three techniques: memory collocation, replication and interleaving. Collocating data means that memory is migrated close to threads that access it frequently to maximize local accesses. For the same purpose, replication stores several copies of the data on multiple nodes. Finally, interleaving causes memory to be allocated such that it is equally distributes across nodes. Since implemented as a kernel module, this approach can be applied to any application without modifications to the program code, but is less fine-grained and more intrusive to set up. While Shoal derives a program's semantics from annotations or DSL compiler analysis, online tuning approaches need to guess programmers' intentions in retrospect. Furthermore, optimizations are applied reactively rather than proactively, causing additional overhead at runtime for migrating and replicating pages.

AutoNUMA [Cor13] also follows an online tuning approach similar to Carrefour. AutoNUMA iteratively clears the present bit of pages in the page table, causing a page-fault on the next access. The page-fault handler records access information and remaps the page. While conceptually similar, this strategy is more expensive than Carrefour's use of performance counter hardware [DFF+13].

However, in comparison to Carrefour, it is less efficient in acquiring memory access statistics as a result

Systems such as SGI's Origin 2000 [Sil15b] use page-level migration and replication of data. Hardware monitors access to pages including which processors tend to access the page, and whether these are reads or writes. Based on that, pages are replicated or migrated towards a frequently accessing CPU.

Li [LGP04] attempts to find the best algorithm for a specific task depending on machine characteristics and workload based on empirical search. In contrast, we decide a priori based on additional information extracted from

high-level languages or given by manual annotations.

Franchetti *et al.* [FVP06] automatically tune FFT programs to multi-core machines. They argue that programming such machines is increasingly complicated, which increases the burden for programmers and makes a case for automatic tuning. Similarly, Atune-IL [SPT09] auto-tunes applications, including the number of threads. It does so by exploring all possible parameters, but tries to reduce the search space.

However, tuning data placement and parallelism individually is not optimal, because threads might then not be scheduled close to the data they are accessing. Hence, affinity of threads and data need to be enforced in order to improve the performance of OpenMP programs [TaMS+08].

Abstraction and programming interface

Our work is based on the idea of automatically and transparently tuning memory placement and access. For that, we first introduce two simple memory abstraction for both bulk in Section 4.4.1 and metadata in Section 4.4.2. We optimize memory allocation only for bulk data and implement it for arrays. We decided to provide a second abstraction for smaller metadata, that cannot be efficiently handled by our bulk data interface: an example are counters. We wanted to encapsulate them as well, so that, given a proper backend, a Shoal program could also be executed correctly on a machine without shared-memory (e.g. a rack). However, in the scope of this thesis, we focus almost entirely on the bulk data abstraction and note that an abstraction for metadata might be useful for more distributed settings in the future. Our abstractions for both bulk and metadata allow us to exchange the underlying implementation without having to modify the high-level program.

In this section, we describe the programming interface and memory abstraction Shoal provides to programmers in the low-level code (Figure 4.2). The generated code – C/C++ for Green-Marl– uses our abstraction to allocate and access memory. At compile time the concrete mapping of array implementation is not made yet, but kept open until the program is executed. The selection then happens at runtime based on hardware specifications. The low-level code is described in Section 4.4.

Shoal encapsulates the entire program state: this includes smaller metadata such as variables, but mostly focuses on larger bulk data such as arrays. If all the state of a program is encapsulated, a program could be executed in a shared memory as well as distributed environment without changing its source code. Furthermore, in the case of failures, both bulk and metadata could be replicated to provide fault-tolerance. We did not yet investigate neither of these in detail, but carefully designed our memory abstraction to support them in the future.

We first give an overview of our abstraction for bulk data and later on

briefly discuss how the remaining state can be encapsulated in a separate data structure.

Abstraction for bulk data

Shoal abstracts bulk data as arrays. Arrays are widely used and well understood by programmers. They are often used to implement other data structures on top of them, for example CSR graph representations used in Green-Marl (Section 3.4.1), sparse matrices in machine learning workloads or columns of database tables. We found arrays sufficient for the workloads we have been looking at in the context of this research, but we expect to add more data types in the future. All of our array implementations implement the same interface. This allows Shoal to select an implementation transparently to the programmer.

Shoal represents its arrays internally as C/C++ class templates. It leverages polymorphism to maintain the various types of arrays while still keeping the code clean and easy to read. Access to arrays can always be executed using methods `set` and `get` provided by the instance of the array's class.

Listing 6 Interface of Shoal

```
// allocate an array
template<class T>
shl_array<T>* shl__malloc_array(size_t size, bool ro,
                                bool indexed);

// get/set element at position i, return size
T get(size_t i);
void set(size_t i, T v);
size_t get_size(void);

// return accessor to array
T* get_array();

// initialize and copy elements between arrays
void copy_from_array(shl_array<T> *src); // from Shoal array
void copy_from(T* src); // from C array
void init_from_value(T value);

// synchronize replicas
void shl__repl_sync(void* src, void **dest,
                    size_t num_dest, size_t size);

// calculate the CRC checksum
unsigned long get_crc(void);
```

Listing 6 illustrates Shoal's programming interface, which decouples com-

putation and memory access allowing transparent selection of different array implementations. Now, we are discussing first how memory is allocated, then operations for accessing data and finally two implementations of our abstraction: one entirely in software using an additional indirection and a second one implemented on top of virtual memory hardware.

Array allocation

`shl__malloc_array` allocates Shoal arrays and selects the best implementation for the machine it is running on based on memory access pattern hints given as arguments.

Shoal always maps all pages of an array to guarantee memory allocation and avoid non-determinism. This provide strong guarantees on where memory is allocated. Shoal also implements a consistent allocation policy across different OSes. This is normally not the case, as operating systems often implicitly allocate memory based on a specific internal allocation strategy. Indeed, these policies even change between different version of the same OS.

Linux, for instance, implements a first touch allocation policy, which causes confusion about where and when memory will actually be allocated. Libraries such as libnuma provide an interface which gives more control, but this lacks support for large and huge pages. Barrelfish [BBD+09] gives the user the ability to manage its own address space via self-paging [Han99]: an application requests memory explicitly from a specific NUMA node and maps it as it wishes.

These systems provide different trade-offs between complexity, portability, and maintainability of application code and efficient use of the memory system: an explicit, flexible interface imposes an additional burden to programmers. We believe that programmers should not have to deal with this complexity while still benefiting from performance gains when choosing a good allocation strategy. Furthermore, we want to avoid manual tuning to adapt programs to new machines.

Data operations

Reads and writes to arrays are performed with `get()` and `set()`, but we also provide optimized high-level array operations for initializing and copying arrays. These provide relaxed guarantees, that allows the Shoal runtime to apply optimizations. For example, the order in which elements are initialized or copied is not specified, allowing these operations to be parallelized and offloaded to DMA engines in an asynchronous fashion.

Writes to replicas can be realized by writing to the master copy and propagating the changes to all replicas using `shl__repl_sync()`. This allows to re-initialize replicated arrays, for example to reuse otherwise read-only buffers in streaming applications.

`shl__repl_sync()` is not strictly required. The same functionality could be implemented using `set()` and `get()`. However, in contrast to these generic low-level access functions, `shl__repl_sync()` allows the use of DMA hardware to accelerate copying data.

Implementation with software indirection

Here, we discuss the implementation of our memory abstraction. For partitioning and distribution of memory, Shoal allocates a contiguous piece of virtual memory and guarantees that it is backed by physical memory on appropriate memory controllers. In that case, software does not require any modifications, as accesses to virtual memory are forwarded to the intended memory controller automatically by virtual memory hardware.

However, for replication, several copies of the same data are allocated in the machine. Virtualization hardware treats these replicas as distinct data, which means that software has to choose which of the replicas to use for each memory access to replicated data by accessing the data via its distinct virtual address.

Since Shoal chooses to replicates, partitions or distributes memory and dynamically enables the used of superpages and DMA hardware, the virtual address of data is unknown at compile time. Replication for example stores the same data multiple times. Each thread then has to lookup which replica to access. In Shoal, this is implemented with an additional indirection.

For efficiency, we introduces the concept of *accessors*. An accessor is a partly cached lookup that is valid only within the thread that has been initializing it via `get_array()`. Depending on the array type used, accesses are directly executed on memory using the accessor instead of access via `set()` and `get()`. We realize this with C/C++ macros. Listing 7 code shows an example.

Additionally, Shoal provides a selection of high-level functions to initialize memory and copy elements between Shoal arrays.

Replication: implementation using paging hardware

This work was executed in collaboration with Reto Achermann.

In contrast to the previous section, we are now discussing an alternative approach for accessing replicated data: the use of replicated page tables to provide distinct address spaces for each thread. This is similar to Carrefour's on-demand page replication [DFF+13].

This is different from what traditional operating systems provide. In Linux, for example, all threads of a process share its page table. We implement this is Barrelfish, whose flexible memory management based on capabilities allows Shoal to enable hardware-level NUMA replication via page table manipulation. We achieve this by duplicating the root page table (i.e. PML4 for `x86_64`) instead of sharing it as is the case for traditional threads.

Listing 7 Example program for accessing a Shoal array

```
#define shl__ARR-NAME__wr(i, v) shl__ARR-NAME[i] = v
#define shl__ARR-NAME__rd(i)    shl__ARR-NAME[i]

/* Initialize arrays */
shl_array<double>* shl__ARR-NAME__array = \
    shl__malloc_array<double>(size, ro, indexed);

/* Execute algorithm - in parallel */
#pragma omp parallel {

  /* Initialize thread state on each thread of OpenMP's pool */
  double* shl__ARR-NAME = shl__ARR-NAME__array->get_array();

#pragma omp for
  /* Execute code */
  for (int i=0; i<size; i++) {
    shl__ARR-NAME__wr(i, 3.141); // write to array
    assert (shl__ARR-NAME__rd(i)==3.141); // read from array
  }
}
```

Figure 4.3 shows a scheme on the resulting page table hierarchy for two threads. Once the root page table is duplicated for each thread, we can selectively share or replicate physical page mappings across threads or thread groups by assigning PML4 slots into shared and non-shared address ranges. This task is performed by our modified Shoal library and Green-Marl runtime, without changing application code. In Barrelfish, this is possible entirely in user-space. No modifications to the kernel are needed. A similar approach was used by Corey [BWCC+08] to reduce the overhead in walking the page tables on multiple cores.

When looking up memory, a thread than finds its root page table via register `CR3`. For memory that is shared between threads, the root page table points to a shared second-level page table (PDPT), which simplified maintaining consistency of the address space across threads. Only for memory that is replicated, Shoal uses a thread local page table allowing to configure replication for each thread individually.

This cannot be done in conventional Linux because page tables are always shared there – replicating data in that case can only be achieved by modifying the kernel or using software indirection for memory accesses as shown in the previous section.

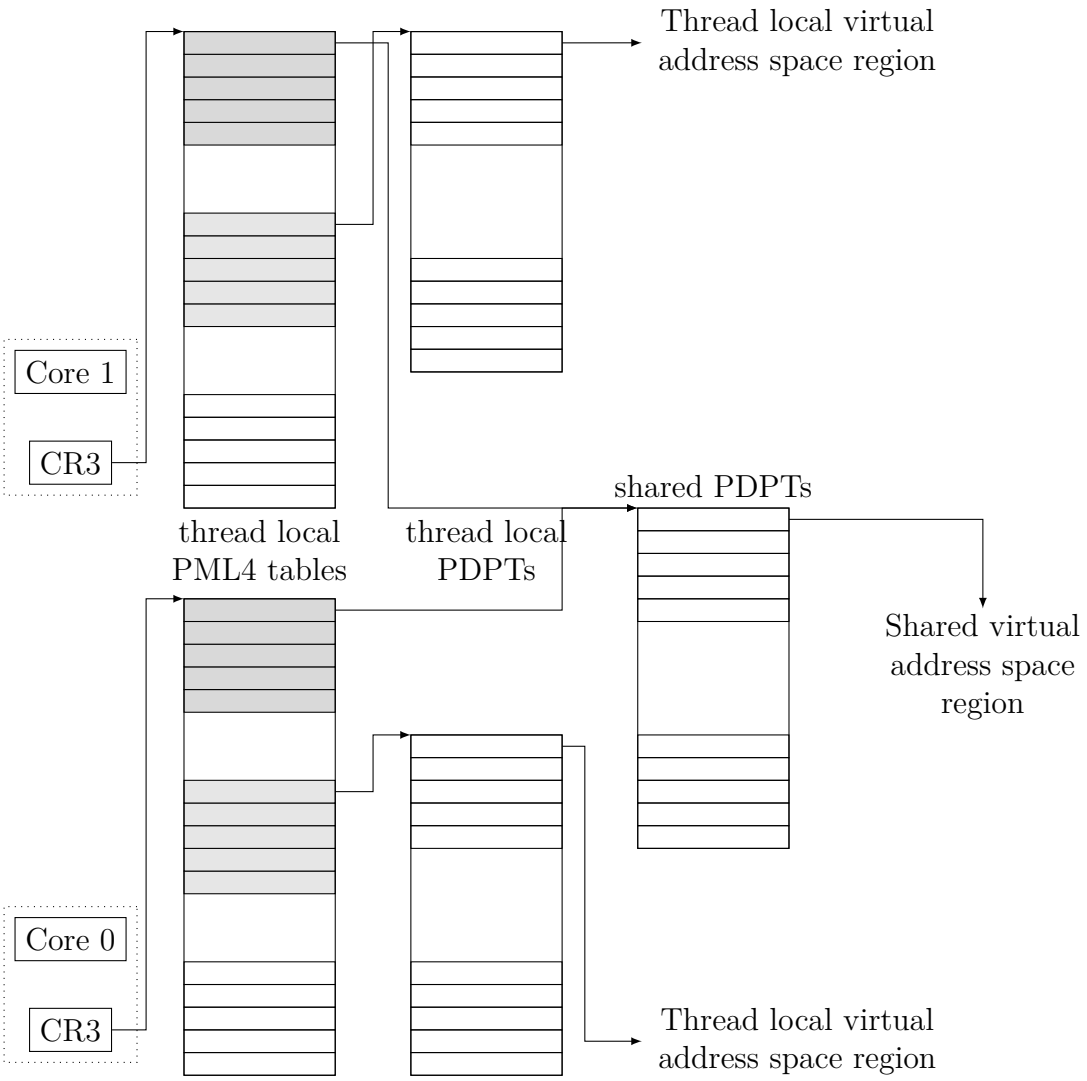


Figure 4.3: Page table layout of two threads with hardware supported replication on `x86_64`.

Abstraction for meta data

In addition to the memory abstraction for bulk data introduced in the previous section, we also implemented an abstraction for smaller meta data similar to structs. For this, Shoal provides a memory abstraction called *frames*.

The idea behind frames is to encapsulate the remaining, meta data presented in typical parallel applications. Such data cannot be efficiently represented with the bulk interface introduced before. We support per-thread and global frames. An example for meta data are counters in applications like triangle counting for graphs. There, one counter per thread and a global one are needed for keeping track of the number of triangles found so far.

Frames complement the data abstraction for bulk data. With the com-

bination of the two, the entire state of a program is abstracted by Shoal. With that, Shoal could run on machines without cache-coherence or shared-memory, where access to frames could be mediated according to where frames are allocated and how they are shared between execution contexts.

Thread initialization

For Shoal, threads have to be pinned to cores. The mapping of thread to partition or replica depends on the core it is running on, so migrating a thread would involve updating these mappings.

For simplicity, we simply pin Shoal threads and initialize these mappings once at initialization. For OpenMP, we update the mappings once at the beginning of a `pragma omp parallel` to ensure that each thread in the thread pool updates its mapping table.

Each Shoal array is represented the to the programmer as an instance of a C/C++ class. In order to get access to the payload itself, threads execute method `get_array()`, which returns a typed pointer to the array the class is maintaining.

This has several advantages. First, the code is modular, clean and easy to read. However, on the fast path, we use a C/C++ pointer for direct memory access.

Shoal runtime

The Shoal runtime is implemented as a library that takes care of selecting array implementations based on extracted access patterns, and hardware specification of the machine. The program to be used with Shoal is linked against this library.

For each array, Shoal has to choose which array implementation to use. This section discusses the trade-offs in doing so and visualizes Shoal's decision trees.

Memory placement

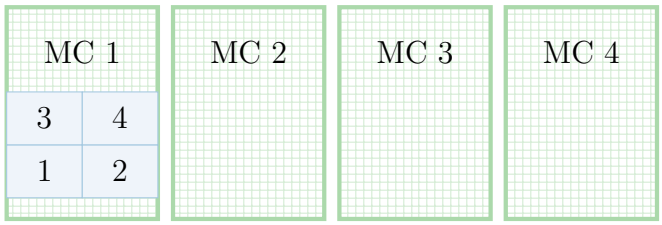
The performance of parallel applications on multicores critically depends on maximizing memory throughput. To achieve that, memory accesses should be equally distributed across all memory controllers such that the maximum accumulated bandwidth can be reached. Additionally, the load on the interconnect should be kept low, so that the memory bandwidth is not limited by the interconnect connecting cores when requesting memory from memory controllers. This is best achieved by localizing memory accesses. If that is not possible, as with random accesses, an equal distribution of memory helps to spread out traffic on as many interconnect links as possible.

Data placement strategies

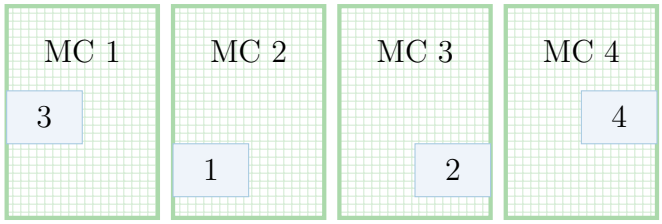
In this section, we describe strategies for allocating memory supported in Shoal. In addition to choosing the allocation strategy, Shoal also tunes them internally based on machine characteristics.

Unless specified otherwise, Shoal guarantees an eager allocation of program memory. Consequently, all allocation strategies introduced here result in a more deterministic program execution compared to the Linux' lazy memory allocation, where programmers have no direct influence on where memory is allocated.

Single node allocation All memory is allocated on the same node. While simple to implement, single node allocation does not scale well and is therefore rarely used for parallel programs.



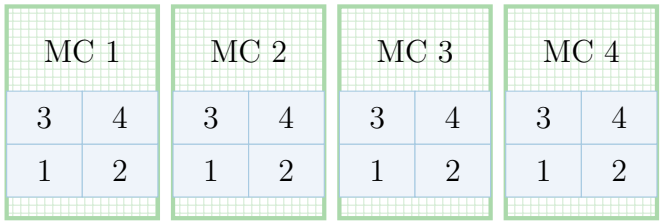
Distributed allocation Distributed arrays allocate data approximately equally, but randomly, across NUMA nodes. The distribution is implementation specific.



Distribution reduces pressure on memory controllers, but can lead to high latency or congestion if many accesses are remote. The performance of distributed arrays can be non-deterministic, as data is scattered semi-randomly and might vary between program executions.

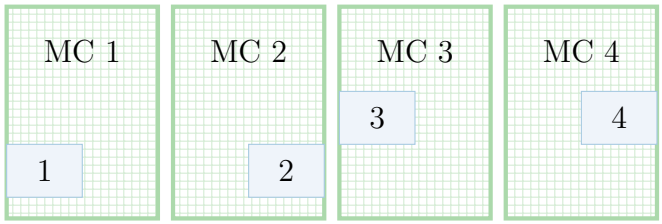
As we show later in Section 4.6.2, the use of data distribution often achieves good scalability without requiring detailed information about hardware characteristics or application requirements, which makes it relatively easy to apply.

Replication Replication stores several copies of the program's working set, typically one on each node. This ensures that memory accesses can be handled locally instead of having to fetch them from remote storage locations, which helps to distribute load and reduce communication costs.



Replication increases a program's memory footprint since several copies of the working set have to be stored on the machine. Care has to be taken when choosing whether or not to apply replication, as increasing the memory footprint to an extent where data has to be spilled to disk has a negative effect on program performance. More importantly, updates to replicated data can be expensive as typically some form of consistency has to be maintained in the presence of concurrent updates. Consequently, to decide if replication can be applied efficiently, information about the application and the machine is required. This includes the working set size, the machine's available memory and its hierarchy, and the read/write ratio of accesses to the memory segment to be allocated.

Partitioning Partitioning is a form of distribution where data is spread out in the machine such that work units can be executed close to where their data is allocated.



The efficiency of applying partitioning depends on the workload: if many random accesses are executed, partitioning can not guarantee local accesses, in which case the performance is similar to a random data distribution as described earlier. If accesses have a large spacial locality, partitioning behaves similar to replication, in which case accesses can be guaranteed to be handled locally. In contrast to replication, partitioned arrays store each data item only once and therefore do not increase a programs memory footprint. However, it imposes an additional challenge for scheduling, since the working set for each thread must be known and the jobs scheduled accordingly.

Choice of array implementation

Based on these observations, we try to adhere to the following principles when selecting array implementations:

- maximize local access to minimize interconnect traffic,

- load-balance memory for the remaining data on all available controllers to avoid contention as a consequence of traffic being limited to only a small number of interconnect links.

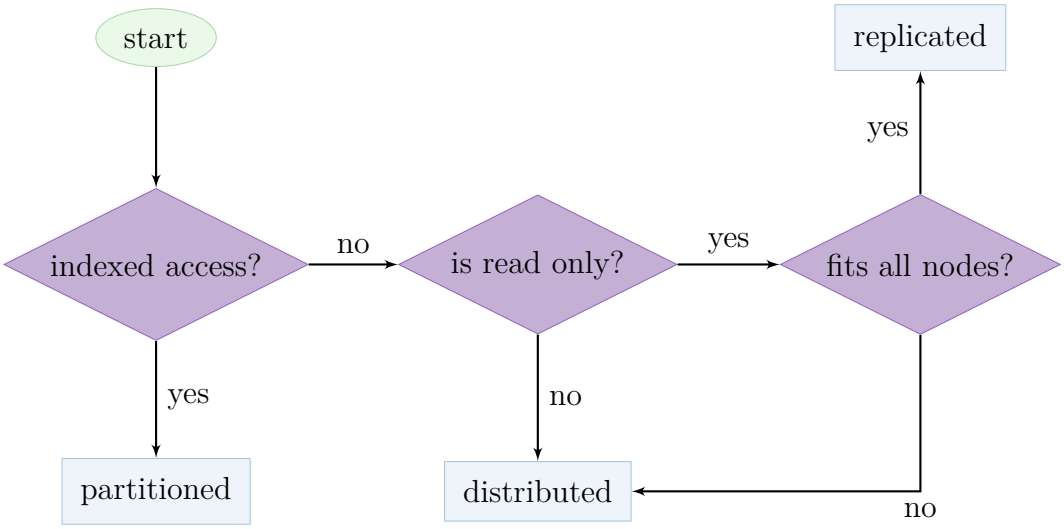


Figure 4.4: Array Selection

We show our policy for selecting array implementations in Figure 4.4. We have described the access patterns used as an input to the decision tree in Chapter 3, the patterns themselves in Section 3.3.

1. We use partitioning if the array is only accessed via an index.
2. We enable replication if the array is read-only and one copy of the data fits into every NUMA node of the host machine. If not all, but some of the arrays can be replicated given the size of memory, an array can be chosen for replication based on the number of total accesses to it. Section 3.4.4 explains how we extract these automatically from high-level programs.
3. We otherwise use a uniform distribution among all available memory controllers.

We only replicate read-only arrays, as we found that the cost for maintaining consistency dominates the performance benefits in current NUMA machines (Section 4.6.4) – however, we plan to revisit this for more complex NUMA hierarchies.

Large and huge pages

If available, we use 2 MB large pages (Figure 4.5). As our results show, this is not always optimal since the impact of using large pages is hard to anticipate.

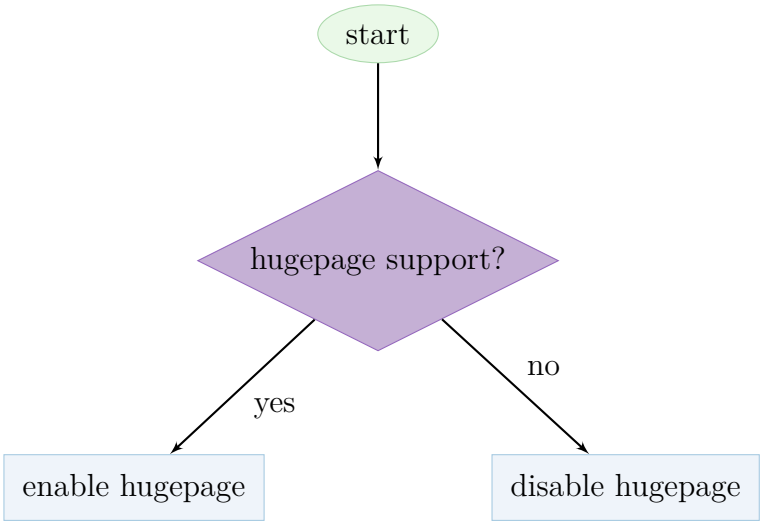


Figure 4.5: Huge page

On average, however, enabling 2 MB large pages seems to be a good choice. On operating systems that allow the simultaneous use of both large 2 MB and huge 1 GB pages, Shoal supports the use of both page sizes. However, currently programmers have to choose the page size manually. Our algorithm always uses 2 MB pages for all arrays if the hardware supports them.

Considerations for further optimizations

Many current multicore machines have independent TLBs for different page sizes. The total coverage of all these TLBs can only be leveraged if all of them are used. This suggests that it might be useful to keep some arrays on normal pages. Consequently, smaller arrays can always be backed by 4k pages. Using large or huge pages would waste memory and increase fragmentation of memory, since the size of the memory allocation has to be a multiple of the page size.

For large arrays with random access, the TLB coverage, even when using large or huge pages, might still not be sufficient to prevent TLB misses. To avoid internal fragmentation and because superpages are a sparse resource, it might then be better to use normal 4k pages even though the array is large. Furthermore, the use of bigger pages increases the granularity at which NUMA optimizations such as distribution and partitioning can be applied [GLD+14]. Physical pages are the unit of memory allocation, and hence NUMA optimizations have to be executed on a size that is a multiple of the page size. For large, and even more so for huge 1 GB pages, the granularity is then often too coarse and the negative impact on NUMA optimizations is bigger than benefits from reducing the number of TLB misses.

One approach would be to use large pages for mostly sequential read-only array access. These can be replicated, and due to spacial locality benefit from

large pages.

Currently, however, our experiments show that the choice of whether or not to use large pages for mappings of arrays does not have a big impact on performance.

Hardware support for copying data

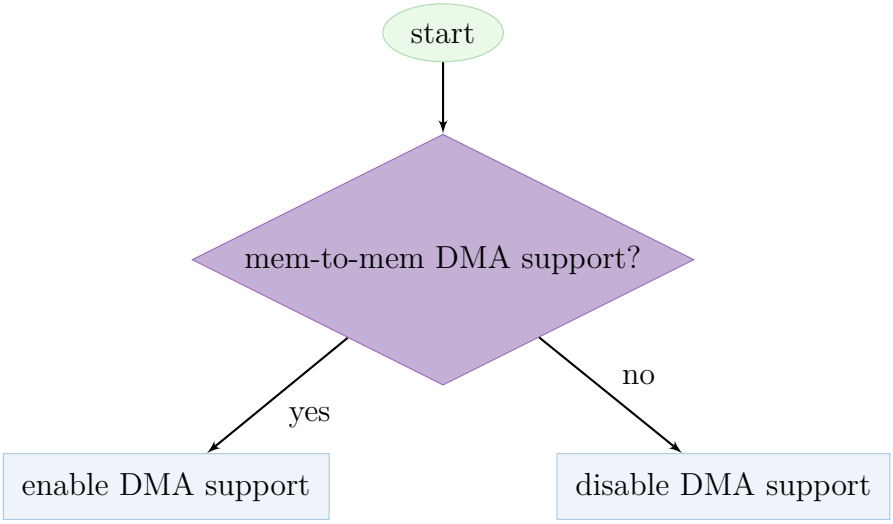


Figure 4.6: DMA hardware

If hardware support for memory to memory copy operations is available on a machine, we use it for functions `copy_from_array`, `init_from_value` and `copy_from`. We visualize this in Figure 4.6.

In the future, we plan to distinguish between synchronous and asynchronous copy operations. In the synchronous case, CPUs have to wait for completion and it makes sense for them to participate in copying data. We evaluate the opportunity for hybrid copy operations in Section 4.6.3.

Otherwise, if data can be copied asynchronously, it is possible to offload the memory copy operation completely to hardware.

Scheduling

For replication and partitioning, Shoal has to map threads to replicas. To prevent threads from migrating to other cores while running a parallel section, we pin threads on Linux by setting their core affinities. Barrelfish has more explicit control over thread placement, which allows us to directly create threads on the the core of choice.

Given a concrete data distribution, data can be partitioned and scheduling can be optimized to execute work units close to where data is accessed. Currently, Shoal is not fully integrated with the OpenMP runtime and we use a static OpenMP schedule for partitioning to ensure that work units

are executed close to the partitions they are working on. This works well for balanced workloads, but can lead to significant slowdown compared to dynamic work distribution approaches if the cost of executing work units is non-uniform.

In the future, we plan to design and integrate our own OpenMP runtime to provide us fine-grained control of scheduling without losing performance for unbalanced workloads. An alternative approach would schedule work units on partitions using OpenMP 4.0's `team`-statement.

OS-specific backends

Shoal supports two operating systems: Linux and Barrelfish [Bar15]. To improve portability, we separate high-level array implementations from low-level, OS-dependent functions which mediate access to the memory allocation facilities or DMA devices.

Evaluation

We now show that programs scale and perform significantly better with Shoal than with a naïve memory allocation. We also show a comparison of our array implementations and analyze Shoal's initialization cost, and finally investigate the benefits of using a DMA engine for array copy.

Throughout this thesis, we use various different machines for evaluation. For reference, we keep a details about this machines in Section A. We use two workloads: Green-Marl and PARSEC Streamcluster:

Green-Marl

We first evaluate our work in Green-Marl, a domain specific language for graph analytic workloads [HCSO12]. The Green-Marl suite already provides a variety of graph algorithms. From these, we have selected three graph algorithms to demonstrate the performance characteristics of Shoal:

- *PageRank* [PBMW99] iteratively calculates the importance of each node in the graph as a sum of the rank of all incoming neighbors divided by the number of outgoing edges they have.
- *hop-distance* calculates the distance of every node from the root using Bellman-Ford.
- *triangle-counting* counts the number of triangles in the input graph. This is implemented as a triple loop: for all nodes in the graph, it looks at all combinations of nodes reachable from it and checks if there is an edge connecting them.

For our evaluation we use two graphs:

- *Twitter graph* [KLPM10] having 41M nodes and 1468M edges. The total working set size is 2.459 GB with Green-Marl configured to 64 bit node and edge types. This is excluding some additional space for arrays that Green-Marl internally allocates, but never uses. The Twitter graph is available for download from `http://an.kaist.ac.kr/traces/WWW2010.html`.

- *LiveJournal graph* [BHKL06] having 4M nodes and 69M edges with a total working set of 392 MB in Green-Marl. We generally use the larger Twitter graph for evaluation, but where not able to run triangle-counting on it for our machines. Hence we were falling back on the smaller LiveJournal graph in that case. LiveJournal is available for download from `http://snap.stanford.edu/data/soc-LiveJournal1.html`.

Streamcluster

PARSEC's Streamcluster [BKSL08] solves the online clustering problem. Input data is given as an array of multi-dimensional points. We manually modified it to use Shoal for memory allocation and accesses.

In contrast to Green-Marl, Streamcluster is implemented in C and hence there is no automatic method of extracting access patterns. We modified Streamcluster to use Shoal's array abstraction to demonstrate that using Shoal directly by programmers can improve scalability with little efforts for manual annotation.

To make Streamcluster work with Shoal, we had to (*i*) abstract access to arrays using Shoal's get and set methods, (*ii*) initialize each thread using `shl__thread_init()` and change the array allocation to use `shl__malloc_array()` instead of `malloc()`. Since Streamcluster is a streaming application, arrays for input coordinates are reused for each chunk of new streaming data, but are otherwise read-only. We use (*iii*) `shl__repl_sync()` to synchronize the master copy of the array to it's replicas once after a new chunk has been read.

Scalability

In parallel workloads, scalability is one of the key concerns. In this section we show the benefits of using Shoal compared to unmodified versions of the workloads and that allocating memory based on access patterns, if available, is favorable compared online methods.

Green-Marl

We evaluated the scalability of three Green-Marl workloads. On each of them, we compare Shoal (on Linux and Barrelfish) with the original Green-Marl implementation ( ) and Carrefour ( ). Figures 4.7 and 4.8

show the scalability of Shoal (in the best configuration each) on A IL 4x4x2 and I SB 4x8x2 respectively.

Shoal clearly outperforms the original implementation by up to 2x. It also performs better than Carrefour's online approach. The latter is a consequence of Shoal knowing the access patterns of each array, while Carrefour has to guess programmer's intentions in retrospect based on a narrow set of statistical data recorded online. Furthermore, our results reveal that a statistical analysis can harm the performance in some cases. In hop-distance, for example, Carrefour seems to fail to detect access patterns correctly.

Note that all configurations including Green-Marl's default OpenMP implementation achieve good scalability for PageRank and hop-distance. Shoal further improves performance by up to around 2×.

We also executed the same measurements on Shoal's Barrelfish implementation (▭) to show Shoal's portability. Our intention is not to claim that either operating system is faster than the other. Instead, we show that our platform-independent abstraction backed by highly tuned platform-specific backends allows Shoal-programs to be executed efficiently on various operating systems without programmers having to change their implementations. Barrelfish, for example, does not implement a first-touch memory allocation strategy. Consequently, despite being efficient on Linux, Green-Marl's low-level optimizations to force allocations to be distributed would not have any effect on Barrelfish.

Note that neither of the results presented in this section includes the graph loading time nor Shoal's initialization. How to load the graph e.g. from disk is a complicated matter all by itself and there has been research in that area to focus on just that [TKKN16]. We investigate the latter in detail in Section 4.6.2. Furthermore, on Barrelfish, only static OpenMP schedules are supported due to implementation limitations. This negatively impacts the performance for triangle-counting. However, Shoal still performs better than the original implementation, which uses dynamic OpenMP schedules.

PARSEC – Streamcluster

We now compare the original Streamcluster implementation with Carrefour and Shoal. Our results in Figure 4.9 confirm the bad scalability of Streamcluster due to the use of `memset()` after allocating arrays [DFF+13, GLD+14]. This causes all memory to be allocated on a single NUMA node, which leads to congestion of the interconnect and an imbalanced used of memory controllers. Due to its smart memory allocation, Shoal achieves an 4x improvement over the original implementation. For that, we replicate the array containing input coordinates and use huge pages to back the memory.

Shoal's approach to smart memory allocation outperforms Carrefour's online method. We want to emphasize here, that we focus on optimizing the most frequently used array. Further optimizations might be possible when tuning more of Streamcluster's internal data structures.

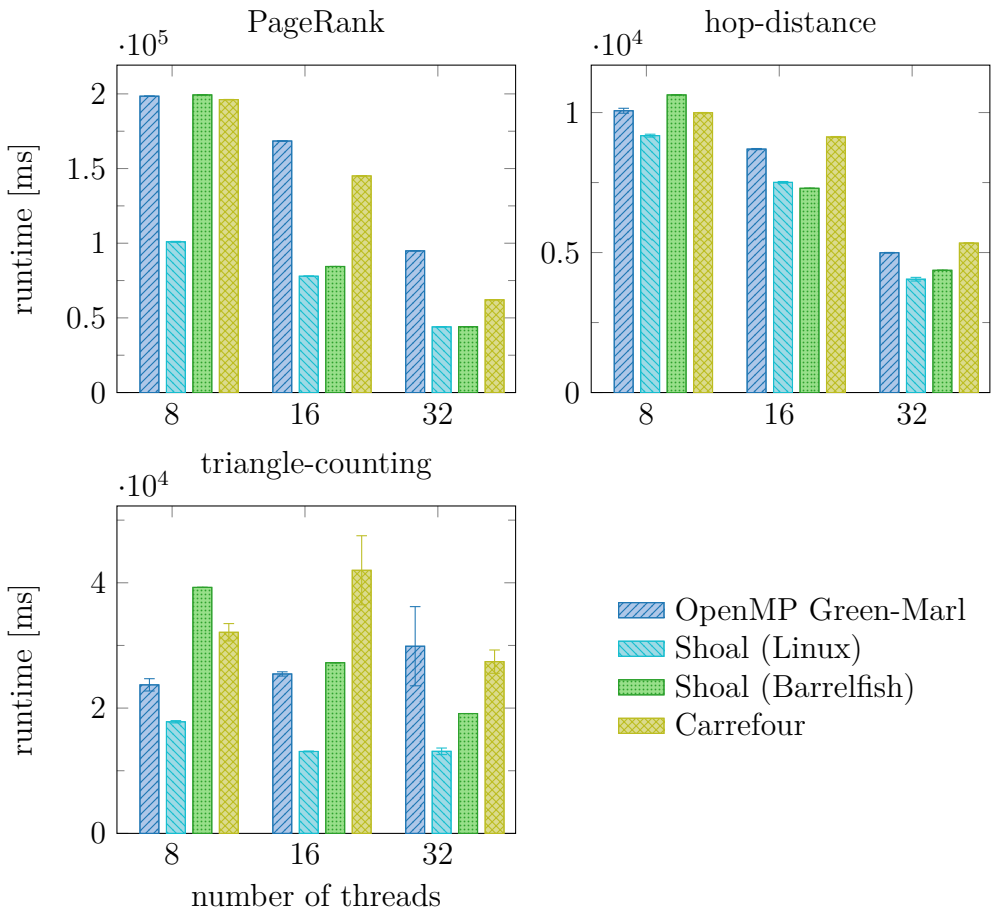


Figure 4.7: Scalability on A IL 4x4x2. Workload: Twitter (LiveJournal for triangle-counting). For Shoal, we chose the best configuration each. We omit results for 64 threads: using SMT threads has similar performance to using only the 32 physical cores.

Comparison of array implementations

We now conduct a detailed analysis of Shoal's different array implementations using all physical cores of our machine. In this section we show, that Shoal achieves better performance than the original Green-Marl implementation regardless of which array configuration we use.

Figure 4.10 shows our results normalized to the original Green-Marl implementation and Figure 4.11 shows the breakdown into initialization and computation times. The measurements were executed on the A IL 4x4x2 using all 32 physical cores. Following, we give explanations for each configuration and relate them to our performance counter observations (Figure 4.12).

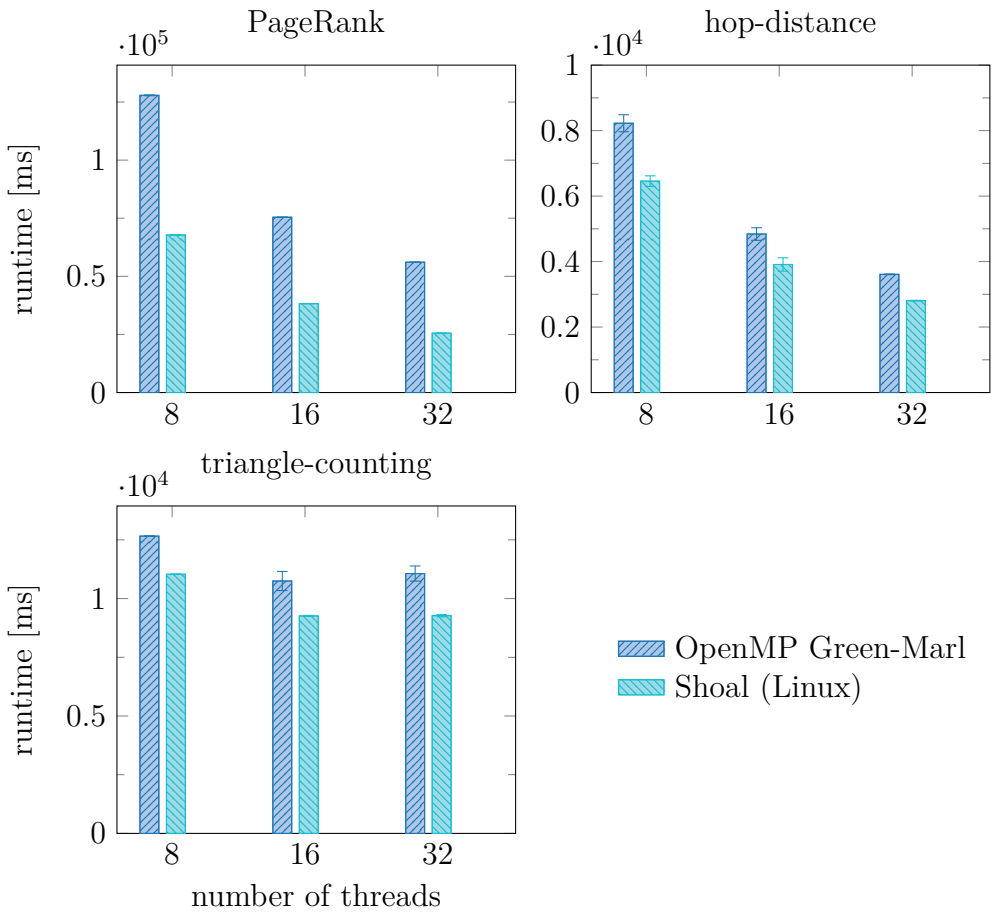


Figure 4.8: Scalability on I SB 4x8x2. Workload: Twitter (LiveJournal for triangle-counting). For Shoal, we chose the best configuration each. We omit results for 64 threads: using SMT threads has similar performance to using only the 32 physical cores.

Distribution (▨)

The original Green-Marl implementation already initializes memory for storing the graphs with a OpenMP loop to distribute memory in the machine. However, this is not done for dynamically allocated arrays (e.g. `rank_next` in PageRank). With Shoal, *all* arrays are ensured to be distributed among the nodes, resulting in a more even distribution of memory and a better performance across all workloads.

As in Green-Marl, Shoal's Linux implementation for initialization of distributed arrays relies on an OpenMP loop to allocate memory evenly across all NUMA nodes. It is hence executed in parallel, which results in small initialization cost compared to other array types. We show this in Figure 4.11.

Our claims are supported by measurements capturing each memory controller's read- and write throughput: compared to the original implementation shown in Figure 4.12 (*i*) where all reads and writes are executed on

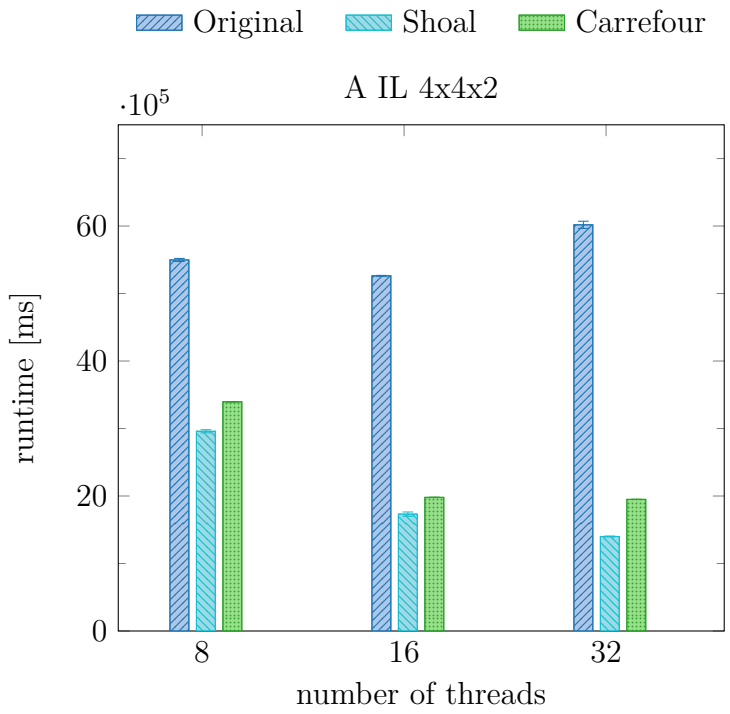


Figure 4.9: Scalability of PARSEC streamcluster on A IL 4x4x2.

socket 0, enabling distribution in Figure 4.12 (*ii*) results in an evenly distributed load across all memory controllers. However, in both cases, the memory controllers are not saturated. Memory throughput suffers from the lower bandwidth of the interconnect links, i.e. 9.6GB/s for QPI. With random distribution of memory, only $1/n$ of all memory accesses are expected to be local for n memory controllers.

Distribution + replication (▭)

In contrast to distribution, we apply replication only to read-only data. In our workloads, the graph itself is not altered by the program and hence replicated among the nodes. This results in a increased fraction of locally served memory accesses and consequently reduces interconnect traffic. Furthermore, memory accesses are evenly distributed among all memory controllers as shown in Figure 4.12 (*iv*). Note that enabling replication without distribution allocates non-read-only arrays as single-node arrays resulting in an unbalanced memory access for that part of the working set, see Figure 4.12 (*iii*). Initialization cost for replicated arrays are higher than distributed arrays because more memory needs to be allocated. Furthermore, we enforce NUMA-aware allocation by touching each replica on its designated node, which induces a high cost for thread migration. Finally, copying the content of the master array to the other replicas causes some additional copy-overhead when initializing replicated arrays (Figure 4.11).

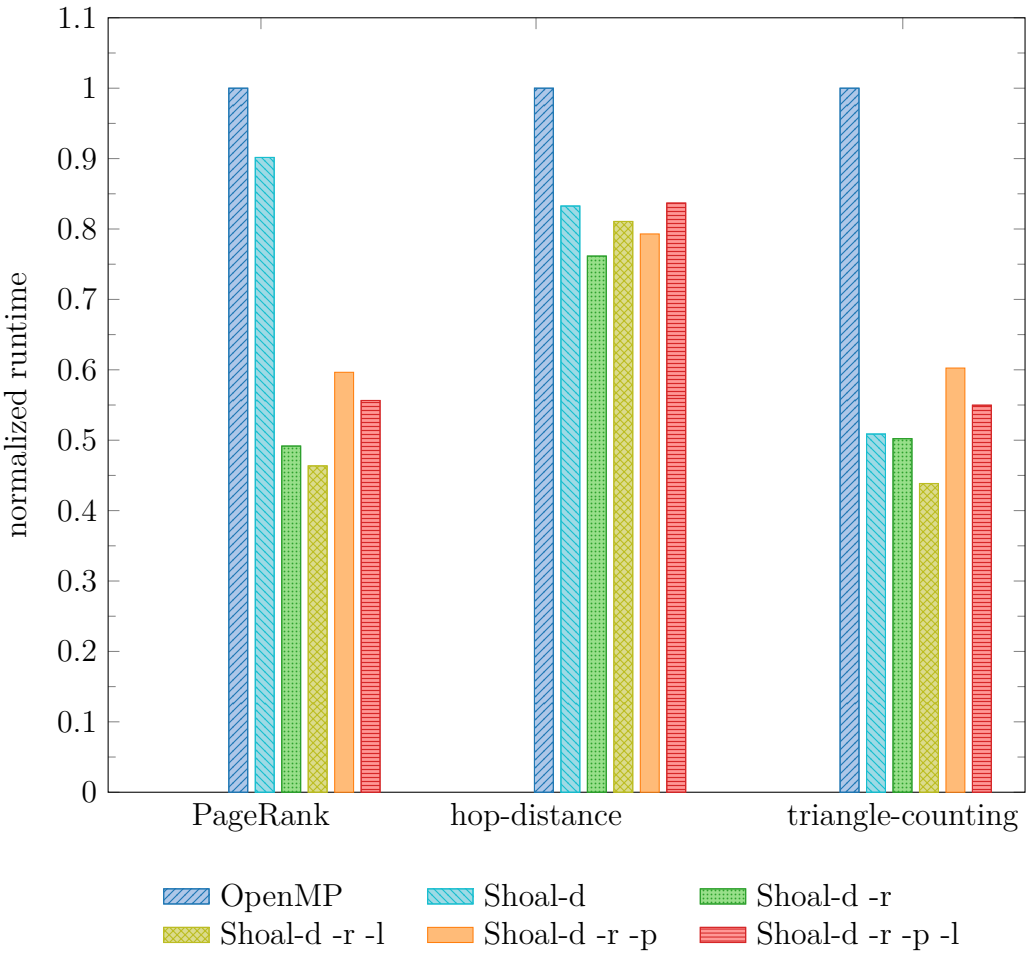


Figure 4.10: Comparison of various combinations of array implementations on A IL 4x4x2: distribution (-d), replication (-r), partitioning (-p), large page (-l) (runtime normalized to stock-Green-Marl)

Partitioning (and)

Since it stores multiple copies of the same data, replication increases the memory footprint of the application. Partitioning aims to avoids this while still preserving locality of replication. For that, the working set is partitioned such that data is stored close to the threads accessing it.

Our current implementation requires a static OpenMP schedule for partitioning to ensure scheduling of work units close to the right partitions. However, static schedules potentially lead to an imbalance of work among the execution units as workloads may be skewed (e.g. in triangle-counting).

Even though the same amount of memory has to be allocated as with distributed arrays, its initialization is more complex: using Linux' first touch policy, Shoal ensures that memory is touched on the correct node by migrating a thread to where memory should be allocated and touching each page from there. This results in similar initialization time as with replication, without the additional time to copy data multiple times.

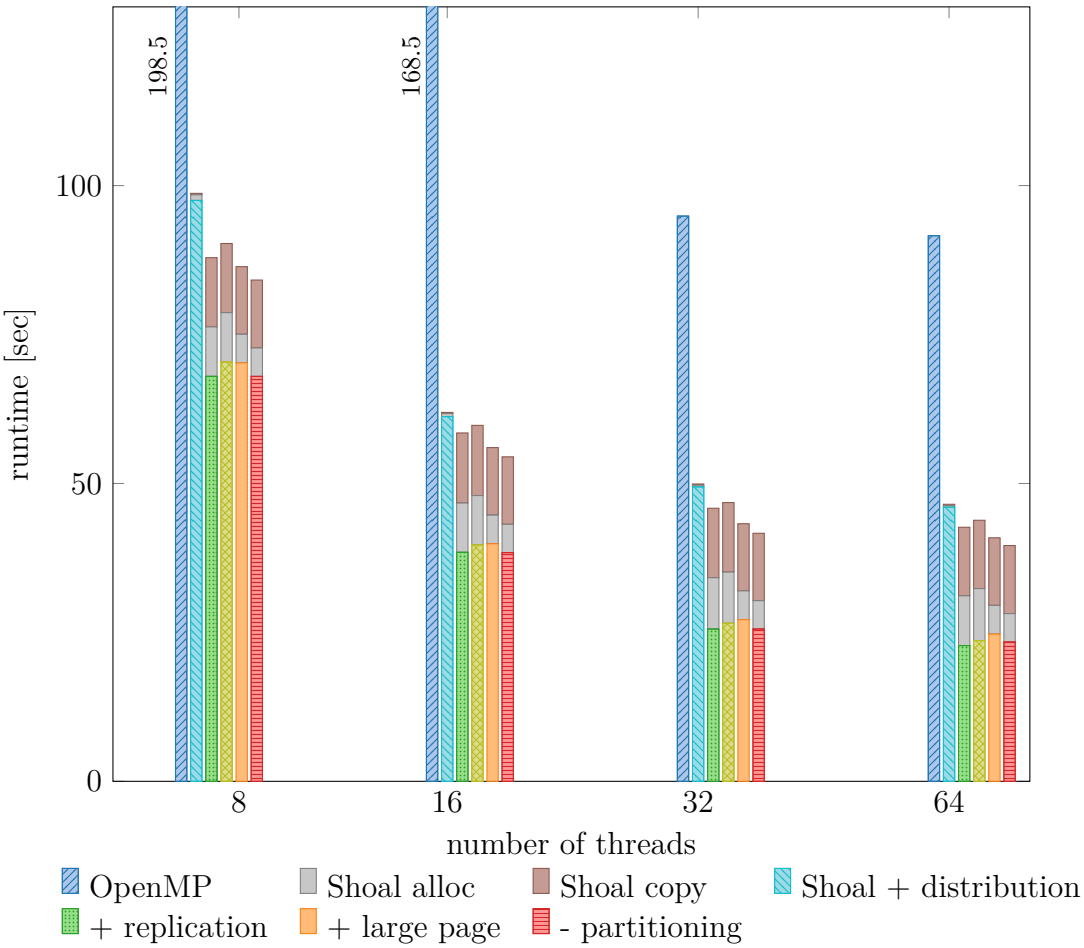


Figure 4.11: Shoal initialization and runtime on A IL 4x4x2 for various array configurations using PageRank with Twitter workload

Large Pages (and)

Modern CPUs support various page sizes and have a distinct TLB for each page size. A miss in the TLB enforces the CPU to do a full page table walk, which drastically increases memory access time. Shoal supports large pages for its arrays. Enabling large pages for PageRank and triangle-counting results in a slightly better performance, while hop-distance runtime increases slightly. This often happens for two reasons: Firstly, it reduces the granularity at which distribution and partitioning can be applied, and secondly, it causes memory to be aligned to a multiple of the same address potentially increasing cache and TLB misses. Gaud *et al.* [GLD+14] concluded similar findings in their experiments with large pages.

Enabling large pages reduces the total number of pages used and, as each page has to be touched only once, the number of required first touches in the allocation process. This results in a decrease of the allocation time (Figure 4.11).

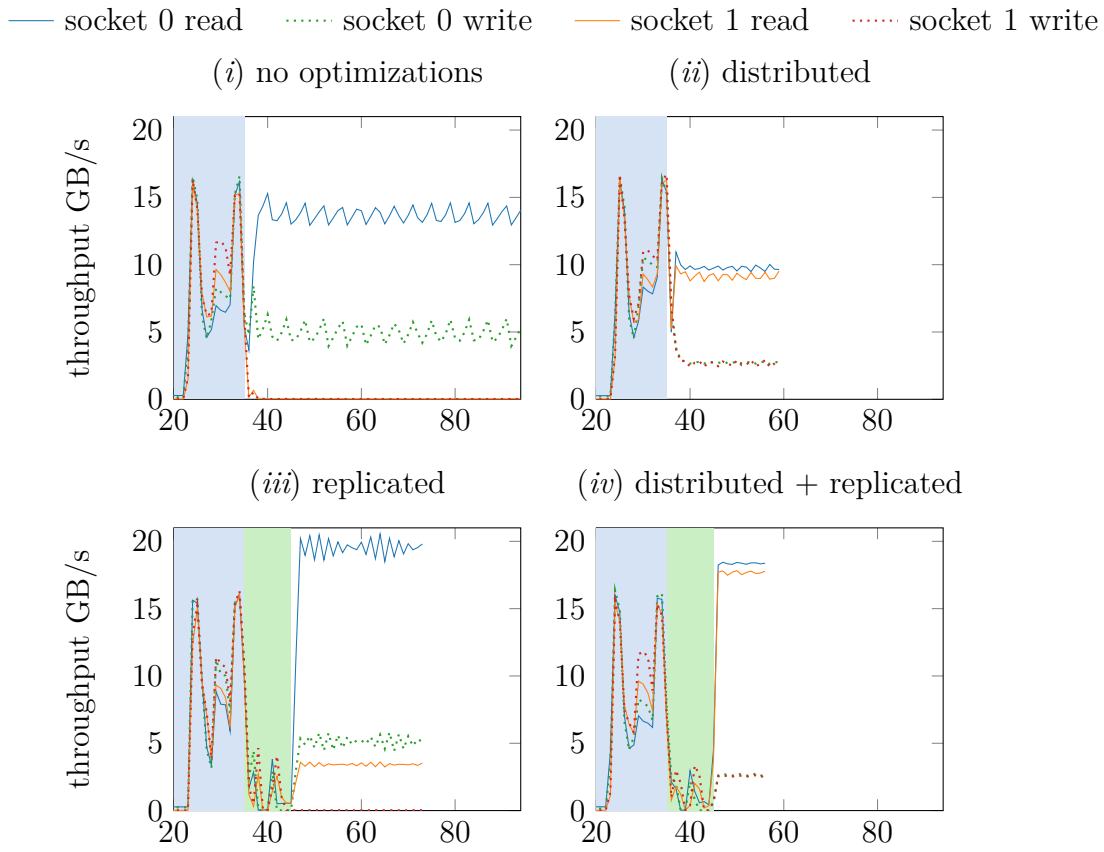


Figure 4.12: Memory throughput for sockets 0 and 1 on I SB 4x8x2. In the first 35 seconds, the graph is loaded from disk. For replication, we show the replica initialization cost. Note: Sockets 2 and 3 are comparable to socket 1 and left out for readability.

Conclusion

We conclude that despite the additional overhead of allocation and initialization, the total runtime with Shoal is still reduced. However, we want to emphasize here, that we do not consider initialization time as a main target of optimization as typically time spent for computation dominates the program execution. For example, in the case of Green-Marl, the cost of loading the graph is considerable higher than Shoal's initialization, which only has to be executed once at program start and could hence be executed in parallel with loading the graph.

Nevertheless, allocation could be improved by (*i*) maintaining a cache of pre-allocated pages on each node, or (*ii*) applying a smarter page mapping strategy (e.g. mbind).

Use of DMA engines

This work was executed in collaboration with Reto Achermann.

Modern CPUs have integrated DMA engines, which provide a rich set of memory operations. For instance, recent Intel server CPUs provide integrated CrystalBeach 3 DMA engines [Int14]. We evaluate the use of DMA engines for initialization and copy operations on I IB 2x10x2 (our A IL 4x4x2 and I SB 4x8x2 do not have DMA engines). We run these experiments on Barrelfish, since user-level support for DMA engines is already integrated and requires no additional setup.

We now compare the raw copy performance of DMA controllers to CPU `memcpy()`. Asynchronous memory operations offered by DMA controllers can free the CPU from the burden of copying data and provide cycles to do actual work. Shoal's high-level interface (Section 4.4.1) allows the runtime to automatically enable the use of memory copy hardware if available for operations `copy_from_array`, `copy_from` and `init_from_value`. Additionally, Shoal offers an interface to start an asynchronous memory copy and to check for completion of the operation.

In our PageRank workload, the ranks are copied between two arrays in every iteration. With our high-level array abstraction, we can use DMA engines to improve the copy time. However, our measurements show, that depending on the array configuration only 1-5% of the entire runtime is spent copying and hence optimizing that part does not have a notable effect on PageRank's overall runtime. hop-distance behaves similarly. Our approach is still meaningful for workloads allowing asynchronously copy of data, which would be a more obvious candidate for optimizations based on DMA engines, since CPUs could execute other work in the meantime rather than simply waiting for the DMA engine's completion.

We now benchmark filling an array with constant data, which is a frequent operation in parallel programs. Note that the same approach is suitable for copying data between arrays as well. If memory can be copied asynchronously, while CPUs execute other work, the use of DMA engines is clearly beneficial. Beyond that, we now look into minimizing the time for synchronous memory copy. In that case, program semantics require the copy operation to finish before further work can be executed.

For best in the synchronous case, we copy a certain ratio of the array using DMA engines asynchronously while using parallel OpenMP loops to copy the remaining elements and synchronize at the end. Figure 2.4 shows this for a varying ratio of how much of the array is copied using DMA engines vs. parallel OpenMP loops affects performance. For the parallel copy, we show the result for using all the physical threads and 40 SMT-threads respectively. As expected with the use of SMT threads, memory access latency is hidden and the use of a DMA engine improves performance only slightly (about 10%). This is presumably because these 40 hyper-threads are sufficient to saturate all memory controllers on that machine. With SMT disabled, the memory latency cannot be hidden as only have as many software execution contexts are available for executing copy operations. In that case, our results show clearly, that using DMA engines and CPU copy simultaneously reduces the

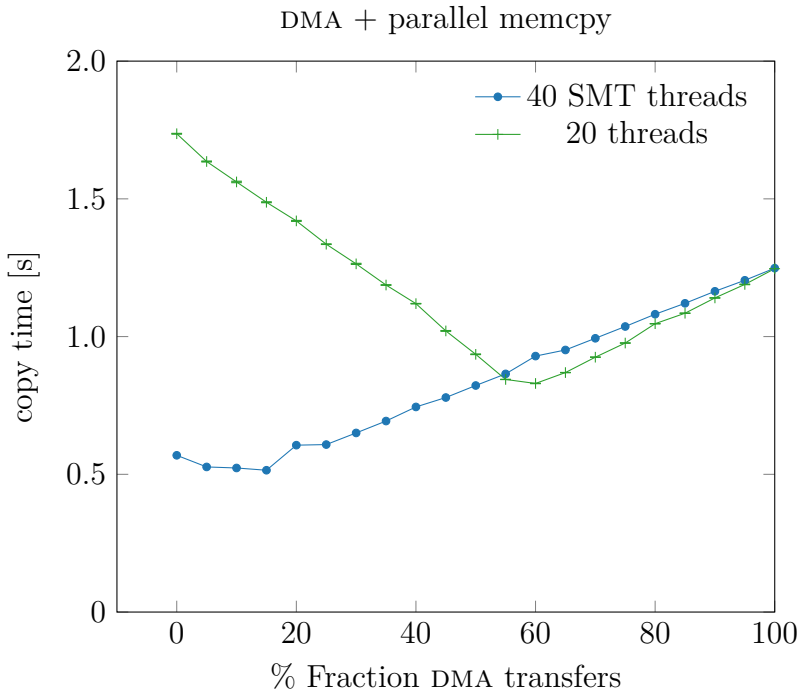


Figure 4.13: Initialization cost for copying data into Shoal arrays using the Twitter working set with replication on Barrelfish

time for copying arrays by 2x.

Writeable replication

Efficiently maintaining consistency of replicated data is difficult; updates must be propagated to all replicas. This can be achieved by issuing writes to all replicas or by applying techniques such as double-buffering and asynchronous copies. Both relax consistency guarantees, but are often strong enough for use with OpenMP loops, where concurrent writes and reads in the same loop iteration would cause non-determinism.

In this section, we show that replicating non-read-only data does not deliver much benefit on current NUMA machines for already otherwise optimized workloads: the additional cost of maintaining meta-information such as write-sets and propagating updates to all replicas outweighs the potential performance gain of replication.

Here, we are looking at hop-distance (Listing 4), having two non-read-only arrays that cannot trivially be replicated since updates have to be consistently applied to replicas. These are `dist_nxt` for remembering the current minimum hop distance from the root and `updated_nxt` to track if each vertex's distance changed in the current iteration of the algorithm. We vary the implementation of these two arrays and apply optimizations to all other arrays as before.

As possible implementations for `dist_nxt` and `updated_nxt`, we compare writeable replication with single-node allocation and distribution (Table 4.1). Replication not necessarily achieves better performance compared to distributed arrays as the load on the interconnect in the latter case is already relatively low. This is even true if the cost of maintaining consistency is excluded ("wr-rep w/o copy op"). As expected, the cost of maintaining consistency grows with the number of replicas as the time required to propagate updates increases linearly with the number of replicas. For example, if four replicas ("wr-rep, 4 reps") are used instead of two replicas, the runtime of the program significantly increases independently on the NUMA node chosen for replication: the runtime when using nodes 0 and 1 for the replicas is similar to using nodes 0 and 4, for example.

dist configuration	runtime [ms]	stderr	notes
single-node	214.0	11.0	
distributed	203.0	0.9	
wr-rep w/o copy op	202.9	7.2	
wr-rep, 2 reps	248.8	7.0	nodes: 0,n-1
wr-rep, 2 reps	249.9	6.8	nodes: 0,1
wr-rep, 2 reps	254.6	9.9	nodes: 0,4
wr-rep, 4 reps	333.6	5.9	nodes: 0,n-1

Table 4.1: Writeable replicas on A IL 4x4x2. Workload: hop-distance with distribution, replication and huge page configuration

We believe that writeable replication will be useful (and needed) in heterogeneous systems, where memory non-uniformity is more drastic (e.g. more NUMA nodes, slower links). In that case, replication of data in local memory is crucial for performance even in the presence of updates. Writeable replication could also have an application for more complex workloads (e.g. a smaller fraction of read-only data), where the simple mechanisms we presented in this paper cannot be applied. Furthermore, if data was to replicated for fault-tolerance rather than performance, a mechanism for updates would be needed. However, in the context of this thesis, we found replication of read-only data sufficient to achieve good performance and could not further benefit from replicating non-read-only data compared to otherwise optimized workloads.

Hardware support for replication

This work was executed in collaboration with Reto Achermann.

Given support from the operating system (such as with self-paging [Han99], as supported by Barrelfish), memory mapping hardware can be leverage to provide distribution and replication of memory regions in hardware without modifying the program's source code. We describe our approach for that in Section 4.4.1.

We compare the performance of hardware replication with the software-based approach as used in the rest of this Chapter. In contrast to the software based approach, that relies on the Green-Marl compiler and is implemented by obtaining the correct pointer to the local replica inside OpenMP's parallel constructs, the hardware-based implements replication solely by modifications to the hardware's page tables. No modifications of software are needed, as the choice of replica is realized by changing a thread's page table.

configuration	runtime [s]	stderr
No replication	37.7	0.002
Software replication	28.1	0.001
Hardware replication	27.8	0.008

Table 4.2: Effects of replication in PageRank executing the Twitter workload on I IB 2x10x2.

Our results in Table 4.2 clearly confirm the benefits of replication: the runtime decreases about 25%. We expect the differences to be emphasized on larger machines with a more complex NUMA topology.

Shoal's hardware assisted replication performs slightly, but statistically significant, better than software based replication However, using hardware replication, programmers can benefit from replication without changing their programs.

Mixed page sizes

We now investigate the interaction of page size and NUMA allocation. Previous work [GLD+14] has shown that while large pages can be beneficial, they can also hurt performance. The choice of page size becomes more complicated when there are more than two options to choose from (e.g., 4 kB, 2 MB, 1 GB for `x86_64`). Furthermore, modern multicore machines have a distinct TLB for each page size, which suggests that using various page sizes increases overall TLB coverage.

Use of large or huge pages has interesting interactions with the NUMA techniques described above, because it changes the granularity at which these techniques can be applied to data structures that are contiguous in virtual memory. The granularity of NUMA distribution, for example, is the page size. Hence, the smaller the page size the more slack the run-time has to distribute data across NUMA nodes. Bigger page sizes also make memory allocation more restrictive. The starting address when allocating memory must be a multiple of the page size. Bigger page sizes can increase fragmentation and increases the chance of conflicts in caches and TLB.

If supported by the operating system, Shoal allows arbitrary combinations of page sizes for different arrays. Furthermore, no complex setup of page

allocations and kernel configurations are required.

We now show the effects of the page size on application performance using Green-Marl's PageRank A IL 4x4x2 and note that AMD's SMT threads (CMT) are disabled in our experiments.

page size	array configuration		
	T=1	T=32 (dist)	T=32 (repl + dist)
4 kB	597.91	51.32	34.43
2 MB	414.80	58.09	28.87
1 GB	395.64	265.94	128.77

Table 4.3: PageRank runtime [seconds] depending on page size and PageRank configuration (repl = replication, dist = distribution, T is the number of threads). Highlighted are best numbers for each array configuration. Standard error is very small.

Table 4.3 shows our evaluation of PageRank for two configurations: First, we execute it single-threaded (T=1). In this case replication does not make sense as all memory accesses will be local, and distribution is unnecessary as a single thread cannot saturate a memory controller even if the entire working set is stored on it – indeed, an increase in remote memory access and interconnect traffic would reduce performance. In this case, for a single application running in isolation, the use of bigger pages always improves performance.

Next, we run PageRank on all cores and explore the impact of replication and distribution on the choice of page sizes. Our measurements show that 1 GB pages clearly harm performance, as distribution is impossible or too coarse grained. With approximately 90% of the working set replicated we counteract this effect. However, the remaining 10% still cannot be distributed efficiently, which leads to worse performance when huge pages are used.

From our results, it is clear that choosing the page size is highly dynamic and depends on workload and application characteristics. It is impractical to statically configure a system with pools (as in Linux) in a way that all programs can request their pages as desired, as the requirements are not known beforehand. Also, memory allocated to pools is not available for allocations with different page sizes.

In contrast, Shoal's simpler interface allows arbitrary use of page sizes and replication by the application without requiring *a priori* configuration of the OS.

Conclusion and future work

In this chapter, we presented Shoal, a library that provides an array abstraction and rich memory allocation functions that allow automatic tuning of

data placement and access depending on workload and machine characteristics. Tuning is based on memory access patterns. These are either (*i*) given by manual annotation, or, ideally, (*ii*) by modifying compilers of high-level languages to extract that information automatically. We have shown that we can use this additional information to automatically choose array implementations that increase performance on today's NUMA systems. We report an up 2x improvement for Green-Marl, a high-level graph analytics workload, without changing the Green-Marl input program. We found our memory abstraction as well as the simple policy for selecting the array implementation sufficient for current workloads and machines, but believe that future machines can benefit from a more fine-grained selection of array implementations.

Limitations

Access distribution of the workload Memory access depend fundamentally on the workload. This is important for example if used with partitioning: it is hard to partition the workload such that memory controllers are accessed equally. Graph workloads representing social graphs, for example, often follow a power-law distribution: there are few well very well connected nodes than most others in the graph. Consequently, the computation for many algorithms is unbalanced, e.g. computing the rank for a highly connected node takes longer. Consequently, more memory accesses are executed from a thread executing that iteration of the parallel loop.

Shoal currently requires a static schedule with a partitioned data set to guarantee that work is executed close to data. A better approach would be to integrate Shoal tightly with the the scheduler, such that it supports work stealing. In that case, dynamic analysis of partitioned arrays would be required to migrate data closer to its workers.

For example, our memory distribution scheme splits up memory equally by size, not by the number of access patterns as workload characteristics are not yet known when loading the graph. A better approach [GLF+15] would be to distribute memory according to the number of memory accesses, i.e. allocate smaller pieces of frequently accessed memory as opposed to larger ones for rarely accessed ones.

Static selection Shoal decides statically how to allocate memory. This is suboptimal in the presence of other load in the system. For example, if one of the memory controllers already suffers from a large number of memory accesses from other applications, an optimal memory allocator should allocate less memory on that NUMA node.

This would either require a global resource management, where each application allocates memory using Shoal, or an online approach, where the performance is measured online and migration and replication of pages executed dynamically based on results from analyzing these statistic results.

Note that both approaches can be combined: static analysis of the program code gives a good initial placement, while online analysis refines the allocation based on that.

Replication with updates Applying updates to replicated data can be tricky. Coordinate between replicas is required if a consistent view of updates is requested from applications. Our current findings suggest that software-replication does not pay off on multicores for an already otherwise tuned workload.

There is a trade-off between performance of updates and the consistency level needed by applications: updates become cheaper when consistency guarantees are relaxed. We expect that replication of non-read-only data will become more attractive for some applications if the runtime would be able to take into account the consistency requirement of applications.

Furthermore, the benefits of replication increase with a bigger discrepancy between local and remote accesses.

5 Synchronization

Introduction

It is well known that the frequency of processor cores has reached the physical limitations in terms of heat dissipation, which sets and end to Dennard Scaling. While Moore's law [Moo06] still holds, it now translates into a growing number of cores instead of higher core frequencies [Gee05] leading to an increase of on-chip parallelism.

To leverage hardware-given parallelism, software needs to be parallelized as well [BC11]. To derive a meaningful global view of a program's state from concurrently executing contexts, program threads then need to be synchronized. Examples of synchronization primitives are barriers and agreement protocols to ensure consistency of distributed state. Due to their importance for many parallel programs, the efficiency of these protocols is often crucial for the overall performance of an application.

In this chapter, we address the problem of efficiently communicating between the cores on a modern multicore machine. We show how near-optimal tree topologies for point-to-point messaging can be derived automatically from online measurements and other hardware information encapsulated in our machine model (Chapter 2).

The problem is hard because modern machines have complex interconnect networks. If tuned carefully, significant latency improvements for group operations can result from carefully choosing a group-communication topology and scheduling of messages. Unlike classical distributed systems, the extremely low message propagation time within a machine means that small changes to messaging patterns and ordering have large effects on coordination latency. Moreover, as we show in Section 5.5.1, different multicores show radically different optimal topologies, and no single tree topology is good for all machines.

In response, we build efficient machine-aware group communication prim-

itives on top of individual peer-to-peer communication channels automatically from information encoded in our machine model. We implement this in Smelt, a software library which builds efficient multicast trees at runtime. Based on that, we show how Smelt serves as a fundamental building block for higher-level operations such as atomic broadcast and consensus.

Smelt provides significant performance gains. Despite other modern barrier operations being highly tuned monolithic implementations, Smelt barriers can be constructed easily on top of Smelt's machine-aware multicast tree without further hardware-specific tuning. Even so, they provide as much as 3× better performance than a state-of-the-art shared-memory dissemination barrier, and is up to 4× faster than an MCS-based barrier.

In this chapter, we first motivate our work and give some background information in Section 5.2 followed by an overview of Smelt's design and implementation in Section 5.3 and Section 5.4.

Motivation and background

We first observe that many parallel programs are increasingly built on message-passing rather than shared-memory synchronization, even within a single cache-coherent machine (Section 5.2.1). Applications normally require group communication semantics, which have to be built on top of individual peer-to-peer connection links between cores. How communication is arranged, i.e. which core sends a message to which other core, is described in the network topology. In addition to the topology, the order in which messages should be sent has to be chosen for each core's outgoing links.

We show that the efficiency of high level applications depends on correctly laying out and scheduling communication on the machine, and that this task is hard because each system has complex message-passing characteristics. Even worse, a good choice is different since machines show very different performance characteristics.

This provides the motivation for a library to automatically create efficient communication operations on a single machine.

The move to message passing

Modern high-end servers are large NUMA multiprocessors with complex memory systems. They are employing cache-coherence protocols to provide a consistent view of memory to all cores. Accessing shared data structures (e.g. in shared-memory barriers or locks) thus entails a sequence of interconnect messages to ensure all caches see all updates and that they do so in the same order [DGT13]. This makes write-sharing expensive: cache lines must be moved over the interconnect incurring latencies of 100s to 1000s of cycles. Atomic instructions like compare-and-swap introduce further overhead since

they require global hardware-based mutual exclusion on some resource (such as a memory controller).

This has led to software carefully laying out data in memory and minimizing sharing using techniques like replication of state, in areas as diverse as high-performance computing [RH15], databases [SSGA11], and operating systems [BPS+09]. Systems like multikernels eschew shared-memory almost entirely, updating state through communication based on message-passing.

Another example is Google's Go language, which favors a message-passing-based programming model over using shared state. This allows simpler programs as concurrently executing threads do not have to be synchronized as with traditional shared-memory programs requiring mutual exclusion, locks and barriers. Furthermore, lightweight threads operating exclusively on thread-local state are easier to parallelize. No synchronization primitives are needed beyond messages explicitly send on channels.

Furthermore, while contemporary machines are mostly coherent, future hardware might not provide global shared *non*-coherent memory [FKMM15]. Here, efficient message-passing is not merely a performance optimization – it is required functionality. The same is true today for programs than span clusters of machines. A single paradigm facilitates a range of deployments. In addition, future multicore machines will likely expose hardware failures to programmers [FKMM15]. If software should be able to continue executing in the presence of failures, application state has to be partitioned and replicated rather than shared. Updates to non-shared state are naturally implemented with a message-passing paradigm.

Ironically, most NUMA message-passing mechanisms today use cache-coherence. With few exceptions [BEA+08], multicore machines provide no message-passing hardware. Explicit point-to-point message channels are implemented above shared-memory such that a cache-line can be transferred between caches with a minimum number of interconnect transactions. Examples are URPC [BALL91], UMP [BBD+09] and FastForward [GMV08]; in these cases, only two threads (sender and receiver) access shared cache lines.

Communication in multicores

While cache-coherence protocols aim to increase multicore programmability by hiding complex interactions when multiple threads access shared-memory, this complexity of the memory hierarchy and coherence protocol makes it hard to reason about the performance of communication patterns, or how to design near-optimal ones.

The protocols and caches also vary widely between machines. Many enhancements to the basic MESI protocol exist to enhance performance with high core counts [HMN09, MHS14], and interconnects like QPI or HyperTransport have different optimizations (e.g., directory caching) to reduce remote cache access latency.

Worse, thread interaction causes performance variabilities that prevent

accurate estimation of communication latency. For example, when one thread polls and another writes the same line, the order in which they access the line impacts observed latency.

Prior work characterized [MHSN15] and modeled [RH15] coherence-based communication, optimizing for group operations. However, these models require fine-grained benchmarking of the specific architectures, for example including specifics of the cache-coherence protocol. They are also more generic in terms of target applications: Molka *et al.* [MHSN15] benchmark communication costs also for bigger payloads, which is unnecessary for purely message-passing based synchronization. As a results, the are providing more accurate models but are less portable algorithms and harder to apply without already having a good understanding of the hardware's characteristics.

Smelt is a much more general approach: we abstract coherence details and base our machine model on benchmark measurements, simplifiying tree construction while still adapting to the underlying hardware. We show that sufficient hardware details for message-passing-based group communication can be obtained from a few simple micro-benchmarks which are easy and fast to execute on new machines without needing to understand intricate low-level hardware details.

Challenges for group communication

In practice, message-passing is a building block for higher-level distributed operations like atomic broadcasts, reductions, barriers or agreement. Group communication operations are either applied to all cores or any arbitrary subset of them depending on the application's requirements.

When messages have to to be sent to many core, they must be sent on multiple point-to-point connections. The problem described above is therefore critical, particularly in complex memory hierarchies. A large coherent machine like an HP Integrity Superdome 2 Server has hundreds of hardware contexts on up to 32 sockets, with three levels of caching, many local memory controllers, and a complex interconnection topology.

For executing a broadcast to all cores, Baumann *et al.* [BBD+09] show how a careful tree-based approach to the broadcasting problem outperforms and outscales both sequential sends as well as using shared memory between all recipients. What matters for the performance of a tree is:

- the *topology* of the tree indicating the which node sends a message to which other node, and
- the *order* in which messages are send to each node's children.

As shown before in Section 2.5, unlike classical distributed systems, message propagation time in a single machine is negligible compared to the (software) send- and receive time as perceived by the sender and receiver. The

store operation underlying each send operation initially takes around 100 cycles on most machines as writes can be buffered by the cache's write buffer allowing the hardware to executed them in parallel. However, buffers are relatively small with only a couple of slots on current machines, so that send operations on individual point-to-point channels quickly exceed that limit. Once all of these slots fill up, write become more expensive as they are blocking until space in the write buffer becomes available again. The cost then is 500 to 1000 cycles.

Loads on the receiver wait until the cache line transfer is done, which is between 300 and 1000 cycles depending on the machine. For broadcasts, each core receives the message only once and succeeding operations depend on the content of the message to be received forcing the processor to stall until data becomes available from remote caches. Since sequential software execution time therefore dominates, it is beneficial to involve other cores quickly in the broadcast and exploit the inherent parallelism available. Figure 5.1 visualizes this for an 8-socket machine.

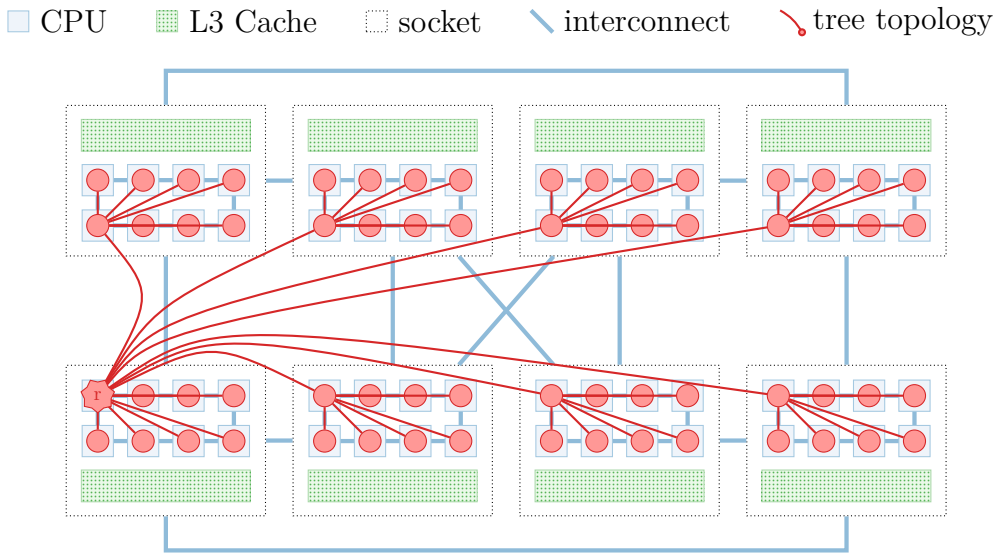


Figure 5.1: Multicore with message-passing tree topology

The cost of sending and receiving messages between cores is a subtle machine characteristic typically not known to programmers, and depends both on the separation of cores in the machine hierarchy and on more complex subtleties of the cache-coherence protocol. Very little of this information is provided by hardware itself (e.g. in the form of registers with hardware features) and vendor specifications are often incomplete or vague. Worse hardware *diversity* is increasing as much as complexity.

Despite this, prior work has built machine-optimized broadcast trees (e.g. Fibonacci trees [BCH92]), and MPI libraries provide shared-memory optimizations for group or collective operations which try to account for NUMA hierarchies [LHS13] using shared-memory communication channels [GS08].

Our results in Section 5.5 show that these trees are sometimes good, but there is no clear winner across all our machines we used for evaluation. Hence, if the goal is to achieve good performance across a wide set of machines, it is non-trivial which topology to choose for a concrete machine. Instead, a careful evaluation of all of these topology is required on each machine to find the best fit for it.

Common tree topologies

As discussed in Section 2.5, send an receive costs dominate message transmission time. As a consequence, the cost of a broadcast would increase linearly with the number of recipients if messages were to be sent from a single sender. Instead, a tree topology can be used. In that case, other cores can be involved in the broadcast. This helps to parallelize the sends leading to a logarithmic increase of broadcast cost with the number of recipients.

We now introduce the tree topologies we evaluate in this paper and visualize them for A BC 8x4x1 (cf. Section A). The first three are hardware-oblivious, constructed without accounting for underlying topology. As we will show later in our evaluation (Section 5.5.1), the choice of the tree topology matters for performance. Unfortunately, there is no clear winner across all machines: while one topology might work well on some machines, it might not be a good match for others.

Note that many of the tree topologies presented here could be improved with some of the strategies we are showing later. This would effectively mean to partly apply our ideas to a static tree topology to make them more machine aware, but would waste the full potential of the machine-aware optimizations we present by enforcing a given tree topology. Note also that the choice of tree only defines the topology on which to send messages, but not the order in which to send in each node.

The choice of the tree topology is orthogonal to the choice of the schedule expressing in which order to send message in each node. In order for them to be competitive, we use our pairwise send and receive costs to provide a competitive schedule also for these common tree topologies.

Binary tree

Among the most simple tree topologies are binary trees (Figure 5.2). Here, each node n connects to nodes $2n+1$ and $2n+2$, which then also get involved in the broadcast so that messages are sent in parallel. Consequently, binary trees are a simple solution to spread out send operations.

However, such trees often introduce redundant cross-NUMA links, causing unnecessary redundant high-latency links that negatively affect overall latency. Performance is suboptimal since the low fanout (two) means that nodes become idle even when they could further participate in the protocol.

Furthermore, binary trees are balanced trees. Both children of a node have a sub-tree of the same size (except for leaf nodes). This wastes opportunities for parallelism, as sub-trees that receive messages first will be idle while others that have been receiving messages later are still sending messages.

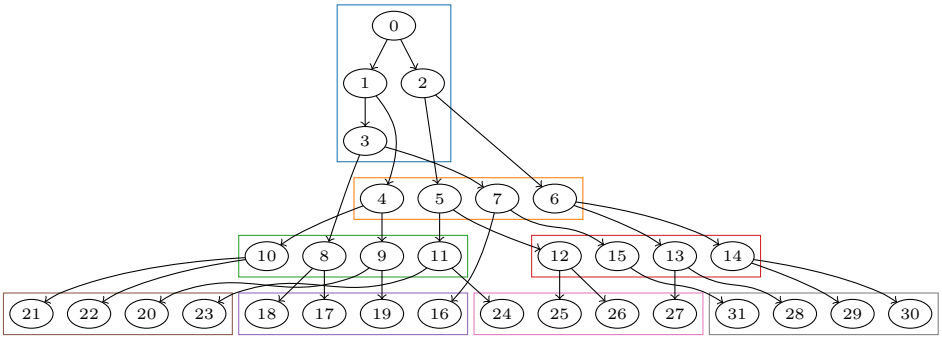


Figure 5.2: Example of a binary tree topology for A BC 8x4x1

Fibonacci tree

Fibonacci Trees [Knu98] (Figure 5.3) are widely used unbalanced trees: left-hand subtrees are larger than right-hand ones. In contrast to binary trees, this imbalance causes more work to be executed in sub-trees that receive messages earlier. However, like binary trees they have low fanout and can exhibit redundant cross-NUMA transfers.

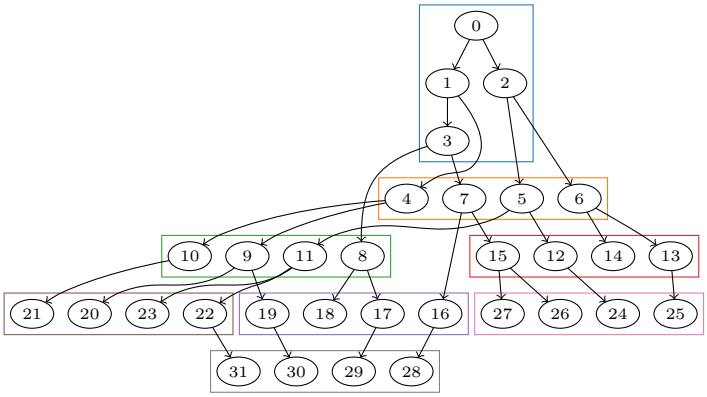


Figure 5.3: Example of a Fibonacci tree topology for A BC 8x4x1

Sequential tree

Sequential sends (Figure 5.4) are also widely used: a root sends a message to each other node sequentially (star-topology). There is no parallelism since one node does all the work. Given the high cost of the send operation, this typically scales poorly for broadcasts on multicore machines.

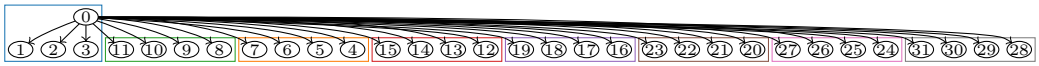


Figure 5.4: Example of sequentially sending messages from one core on A BC 8x4x1

Minimum spanning tree (MST)

In contrast to previous tree topology, the Minimum Spanning Tree (Figure 5.5) considers the link cost when building the tree. It does not enforce a fixed topology. The resulting tree could be anything including a single path or a star-topology with a single sender, which both are executing the send operations purely sequentially.

Our Minimum Spanning Trees use Prim's algorithm [Pri57] by adding edges in ascending order of cost until the graph is connected. This minimizes expensive cross-NUMA transfers, but does not optimize fanout and hence send parallelism.

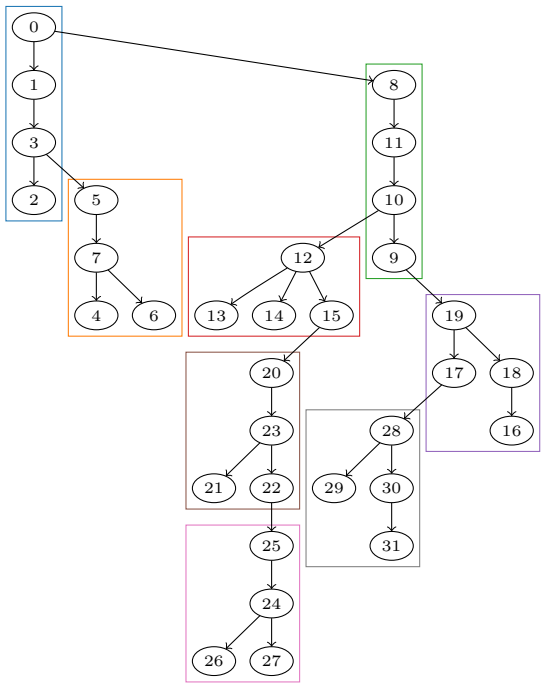


Figure 5.5: Example of a MST tree topology for A BC 8x4x1

Cluster

Cluster (Figure 5.6) trees are built hierarchically, as in HMPI [LHS13]. A binary tree is built between NUMA nodes, and messages are sent sequentially within a node.

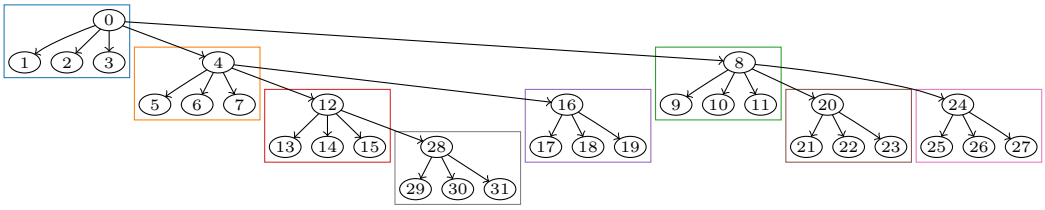


Figure 5.6: Example of a Cluster tree topology for A BC 8x4x1

Bad trees

As a worst-case tree example, we built a topology we call "bad tree" (Figure 5.7) by running an MST algorithm on the inverse edge costs, which maximizes redundant cross-NUMA links. We will use this later to show that the topology matters, and choosing a sub-optimal tree can be as bad as sequentially sending messages on some machines.

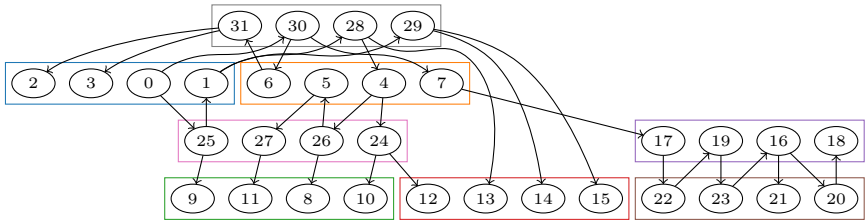


Figure 5.7: Example of our "bad" tree topology for A BC 8x4x1

In some cases, a bad tree constructed like this can still achieve better performance than sequentially sending messages, despite the high send-cost on each link, simply because of parallelizing send operations.

The Smelt approach

Our library, Smelt, simplifies programming multicore machines by providing a machine-aware atomic broadcast tree, that smartly adopts to the concrete machine it is running on. The tree topology for it is generated automatically from a machine model, which unifies information given by hardware itself with a set of fine-grained micro-benchmarks to overcome insufficient information about pairwise communication cost between cores.

We show good performance on all machines we are considering for our evaluation compared to other approaches with a fixed tree topology.

We also see Smelt as a building block for implementing higher-level protocols on top of it. We show how to build an efficient barrier in only two lines of code that performs comparably or better than state-of-the-art shared-memory barriers. Furthermore, we evaluate an agreement protocol, which can be used to provides consistent updates on replicated data. We build a key-value store on top of that, which provides good performance from repli-

cated data. Furthermore, our key-value store is fault-tolerant to crash-stop failures of replicas.

Design

Smelt combines various tools to a smart runtime system: (*i*) a machine model with micro-benchmarks to capture hardware characteristics, (*ii*) a Tree Generator for building optimized broadcast trees and (*iii*) a runtime library providing necessary abstractions and higher-level building blocks to the programmer. An overview of Smelt's runtime system is visualized in Figure 5.8.

In Chapter 2, we describe our machine model consisting of static information encoding the memory hierarchy of the machine as well as its cache configuration. We enrich this information with fine-grained micro-benchmarks that accurately capture the cost for sending and receiving messages for each pair of cores.

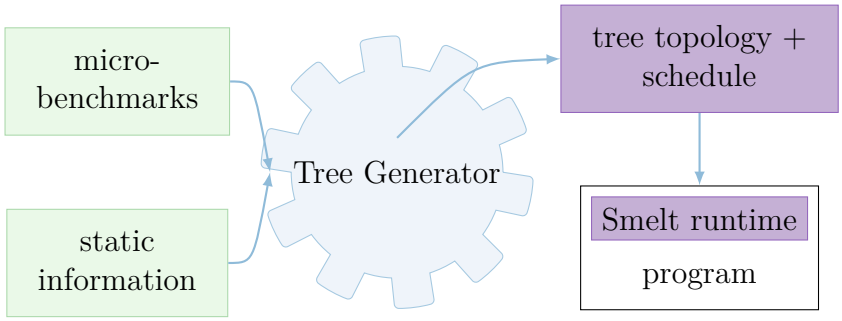


Figure 5.8: Overview of Smelt's design

In this chapter, we describe how our Tree Generator automatically creates a machine-aware group-communication topology from that model. We use tree-based structures for group communication. As two fundamental building blocks, Smelt offers two highly tuned machine-aware communication primitives: broadcasts and reductions. These can then be used to build higher-level algorithms on top of that. In the context of this thesis, we will show how to use them to implement a barrier and agreement protocols. For the latter, we show a simple two-phase commit protocol and 1Paxos, a Paxos-variant for multicore machines.

Broadcasts Smelt's broadcast primitives guarantee reliability and ensure that all nodes receive messages in the same order (atomic broadcast). We assume that multicores today are fail-stop as a whole and hence either run reliable or the entire machine fails. This includes the interconnect: messages are sent reliably and in order. What remains is for software to provide buffer management to guarantee that senders to not overwrite messages in queues before receivers read them.

Future machines will likely have partial failures that are exposed to and have to be handled by programmers [FKMM15]. Smelt does not currently provide reliable channels over unreliable hardware: however, we believe that our machine model could naturally be extend to a system with a combination of reliable and unreliable connection channels. As a result, a good message-passing configuration of the topology would be composed out of a mixture of different message-passing backend implementations, each of them tuned to a subset of the machine.

Smelt's broadcasts start at a defined root and therefore all messages are sent through the root which acts as a sequentializer. An alternative approach would be to use multiple trees with different roots. In that case, however, each core has to poll several memory locations for messages associated with multiple endpoints from different trees. This increases the receive overhead on each core, but also the latency, as each endpoint is only checked in a round-robin fashion. The sequentializer together with the provided FIFO property of the edge links implements the atomic broadcast property.

Reductions Collective operations such as reductions do in-network processing on each node from payload received from all children and pass the new value to the parent node. We use the same tree as in the broadcasts whereas the final value can be obtained at the root.

Finding a broadcast on multicores In order to find a topology for executing broadcasts, we take as an input the graph representation of the machine as described in Section 2.4.2.2. This represents low-level information about the communication capabilities of a concrete multicore.

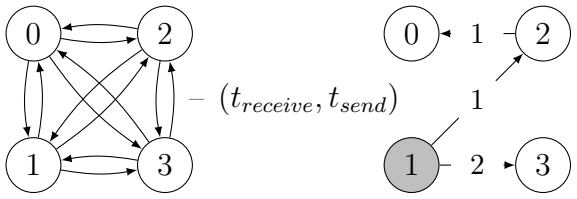


Figure 5.9: Example of fully connected input graph for four CPUs and one possible resulting broadcast tree topology τ with root 1 and edge send order. The root of the broadcast tree v_r is visualized with a dark background. The order in which messages are sent to children is given as an edge priority.

As shown in Figure 5.9, the desired output consists of three parts:

- a root v_{root},
- a tree $T = (V, E')$ with $E' \subseteq E$, where T is a spanning tree of G and

- the schedule $O(v_s, v_d)$, a function representing the order in which a source vertex v_s sends on its outgoing edge to v_e. o maps to a single integer value out of range $1 \dots n$ for a node v_s with outdegree n.

The combination of tree topology T, schedule O and the root node v_{root} has to be chosen such that the latency $lat(T)$ is minimal.

$$lat(T, o) = \max_{v \in V} \ lat(T, o, v)$$

The latency $lat(T, o)$ is given recursively from (i) send time $t_{send}(v_p, v_i)$, the cost of sending a message from vertex v_p to v_i, and (ii) $lat(T, o, v)$, the time until node v receives a message T starting in node v.

For a node v_k with parent v_p and a schedule o with $o(v_p, v_k) == k$, the recursive calculation is calculated as follows:

$$\begin{aligned} lat(T, v_{root}) &= 0 \\ lat(T, v_k) &= lat(T, v_p) + \sum_{i=1}^{k} t_{send}(v_p, v_i) + t_{receive}(v_p, v_k) \end{aligned}$$

Send operations are implemented as memory store instructions. From the perspective of software, only one instruction is executed by each send, making it impossible to further parallelize it. Hardware provides write buffers caching write operations until they are applied to the caches. It therefore hides the cost of the cache-coherence protocol to some extend allowing some small number of send operations to be executed concurrently. However, write buffers are small an quickly filled up, forcing consecutive send operations to wait for previous ones to finish. Consequently, send operations are to a large extent executed sequentially, which is why v_k's latency is delayed by the send cost of neighboring siblings v_i of its parent's node v_p, that have a lower send order: $o(v_p, v_i) < o(v_p, v_k)$.

Simulation

Our model allows to simulate and predict the performance and therefore reason about the performance of algorithms on a wide set of machines. We depict it in Figure 5.10.

The simulator is an event based program. Available events are send, receive and propagate. Events are raised by algorithms that define the sequence in which these events occur and define each processor's internal state. For example, for reductions, the internal state is the number of messages already received from children before the message can be forwarded to the node's parent. As soon as enough receive events have been raised on that node, it will schedule a send to its parent.

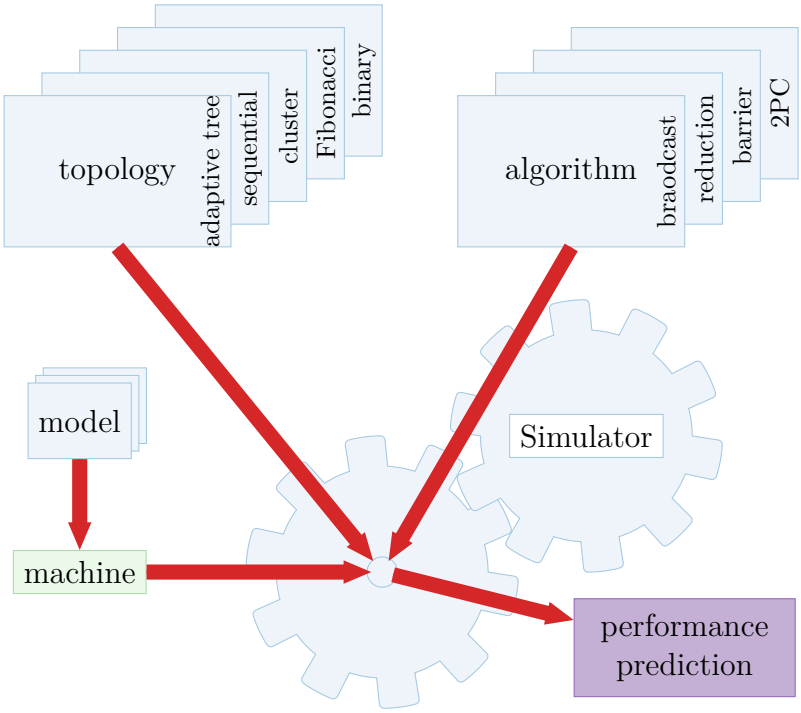


Figure 5.10: Overview of the Simulator

Children and parents are encoded in the topology that is executed and the timing information is read from the pairwise data in the machine model as we have described in Section 2.4.2.2. The Simulator ensures that receives and sends are executed sequentially, as it would be executed by hardware.

In its core, the Simulator uses a virtual time emulating cycles and schedules events as given ordered by a priority queue. It predicts the execution time of algorithms for a concrete set of machines based on the pairwise measurements as included in the their models.

As we show in our evaluation in Section 5.5.3, our predictions, while not perfect, are in line with what has been reported by the community Ours approach is quick to evaluate due to a simple model, even across a wide set of machines and allows to rapidly test new algorithms for a large arsenal of multicore machines, which allows adequate predictions about how an algorithm will perform in a real setup of an average machine.

Adaptive tree: a machine-aware broadcast tree

When building our tree, we rely on the performance characteristics described in Chapter 2, and make use of the fact that the Tree Generator can defer a global view on the system state for message send and receive times. For example, it knows which messages are in transit and which nodes are idle. The Tree Generator operates as an event-based simulation using our pairwise measurements (Section 2.4.2.2). Whenever an active core is idle, it uses the

model to choose an inactive core to send the next message to.

The desired output of the Tree Generator is (*i*) the root of the tree, (*ii*) a spanning tree connecting all nodes, and (*iii*) a schedule that describes the send order in each node.

Note that, while we generally discuss broadcasts to all cores of a multicore machine, the algorithm presented here can directly be applied to generating multicast trees as well. The machine model contains the performance characteristics of the entire machine including any arbitrary subset of cores and hence, can be directly used for generating multicast trees with the exactly same algorithm. We show the performance of Shoal's multicast in Section 5.5.4.

Step 1: Base algorithm

We now first describe our basic offline algorithm to generate a broadcast tree, which we call *adaptive tree*. This algorithm is the foundation for optimizations presented later. It builds a basic machine-aware tree in a single iteration. In contrast to optimizations described later, the basic version of the algorithm does not reconsider an edge already chosen for the broadcast for replacement by other edges.

The task is to design a good heuristic to find near-optimal solutions. For example, messages can first be sent on expensive links to minimize the cost of the link that dominates the broadcast execution time. An alternative strategy would be to send on cheap links first to increase the level of parallelism early up. However, we found that for current multicore machines, it is more important to send on expensive links first: local communication is comparably fast, so messages can be distributed locally and efficiently once received on a NUMA node. This is likely a consequence of the communication cost still being non-uniform on many machines (e.g. Figure 2.5) and even on machines with a uniform remote communication cost (e.g. Figure 2.6), it is still important to execute remote communication first, as they are significantly slower than local one.

Another trade-off is between minimizing expensive cross-NUMA links and avoiding nodes being idle. Our approach to this problem is to initially avoid redundant cross-NUMA links, but to iteratively add them in the optimization phase in case idle nodes have enough slack to send more messages and the message would be received earlier via an additional NUMA link rather than a belated local communication given the often higher cost of remote communication.

Our heuristics are as follows:

1. We select the root to be the core having the lowest average send cost to every other node in the system.

2. Remote cores first: We prioritize long paths as they dominate the latency for broadcasts. We select the cheapest core from the most ex-

pensive remote NUMA node.

3. Avoid expensive communication links: We send the message to a remote NUMA node only if no other core on that node has received the message. We can do this because our Tree Generator has global knowledge on the messages in flight. This minimizes cross-NUMA node communication.

4. Local communication last: Send messages to cores on local nodes last, since this is relatively cheap.

5. Parallelism: We try to involve all nodes in the broadcast as much as possible. The challenge here is to find the optimal fan-out of the tree in each node. The result often resembles an imbalanced tree so that cores that received a copy of the message early have a larger sub-tree than later ones.

6. No redundancy: We never send messages to the same core twice. The Tree Generator knows which messages are in flight and will not schedule another send operation to the same core. We will relax this heuristic slightly with incremental improvements of the tree as described in the next section.

We describe the tree generation in detail in Algorithm 1. At any point during the generation run, a core in the Tree Generator can be in either of two states: (*i*) *active* meaning that it has received a message and is able to forward message to other nodes, or (*ii*) *inactive* otherwise. Inactive nodes are waiting to receive a message from their parent. The set of active cores is denoted as A_{cores}. NUMA nodes are active if at least one of its cores is active (A_{nodes}).

When generating the tree, we heavily rely on our machine model, in particular the pairwise send- and receive cost introduced in Section 2.4.2.2. t_{send} and $t_{receive}$ directly refer to these measurements. Function `SEND(v)` sends a message to node v. Functions `PICK_MOST_EXPENSIVE()` and `PICK_CHEAPEST()` select the most expensive or cheapest out of the given set of cores respectively. We use the first one to find the most expensive NUMA node using the sum of send and receive time as a metric for the overall cost of communication with that core. `PICK_CHEAPEST()` works similarly, except that we select only on the send cost to minimize the effort required on the sending core. Function `NODE_OF(c)` returns on all cores that are on the same node than c.

The output of the algorithm is given from function `SEND`. The call adds the corresponding edge to the output graph T and remembers the ordering as returned by o.

We build the trees once at program initialization, although our framework supports acquiring and replacing the tree online.

We observed that the tree obtained from this algorithm is multi-level tree for most machines. Message delivery is first executed across NUMA nodes

Algorithm 1 Smelt: adaptive tree generator

```
C_all                                                    ▷ Set of cores
A_nodes ← NODE_OF(c_root)                                ▷ Active nodes
A_cores ← c_root                                         ▷ Active cores

function PICK_MOST_EXPENSIVE(C, i)   ▷ Pick most expensive neighbor
    return arg max_{x∈C} (t_send(i, x) + t_receive(i, x))

function PICK_CHEAPEST(C, i)                  ▷ Pick cheapest neighbor
    return arg min_{x∈C} (t_send(i, x))

function NODE_IDLE(i)                  ▷ Executed for active idle node i
    C_inactive ← C_all ∩ (A_cores ∪ CORES_OF(A_nodes) )
    c_next ← PICK_MOST_EXPENSIVE(C_inactive, i)
    if SAME_NODE(i, c_next) then                                 ▷ Local
        SEND(c_next)
        A_cores ← A_cores ∪ c_next                       ▷ Mark core active
    else
        C_eligible ← NODE_OF(c_next)                     ▷ All cores on node
        c_next ← PICK_CHEAPEST(C_eligible, i)
        SEND(c_next)                                     ▷ Send remotely
        A_nodes ← A_nodes ∪ NODE_OF(c_next)
        A_cores ← A_cores ∪ r

function ADAPTIVE_TREE                ▷ executed by the Tree Generator
    while C_all ∩ A_cores ≠ ∅ do
        if c_self ∈ A_cores then
            NODE_IDLE(c_self)
        else
            WAIT_MESSAGE
```

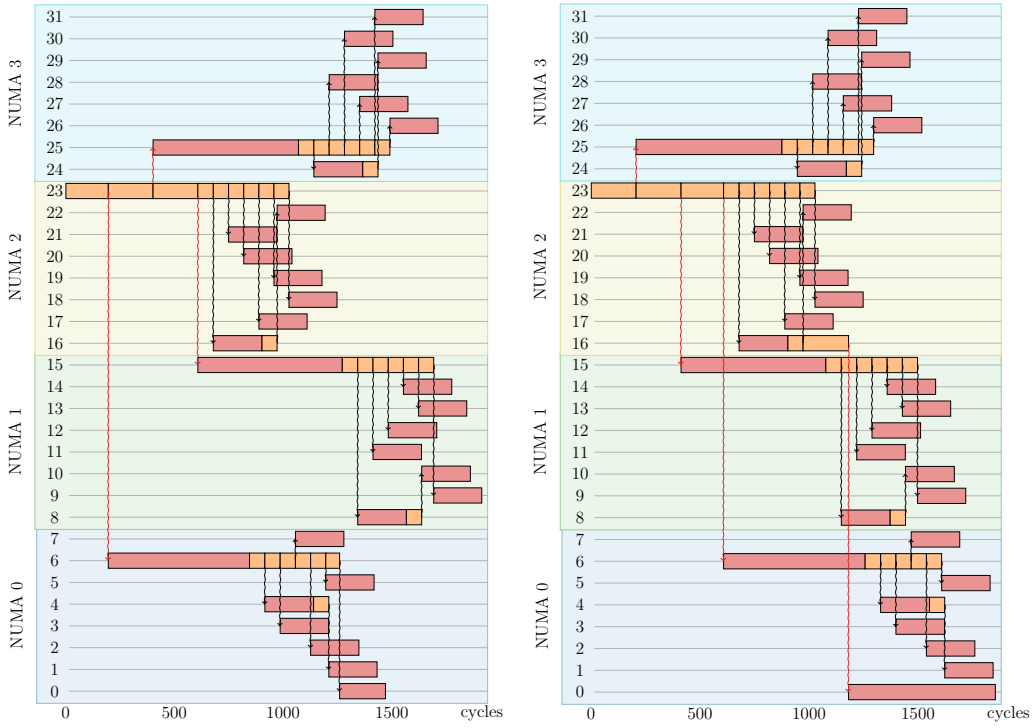
and then further distributed within each node. The Tree Generator creates a multi-level hierarchy in either of these steps only if the send operation is relatively expensive compared to receives. Otherwise, it will sequentially send messages. For example, with a NUMA node, sequentially sending messages is often faster than a multi-level sub-tree.

In our evaluation (§5.5.1), we show that a tree generated with our Tree Generator performs comparably or better than the best static tree topology on a wide range of machines. While the algorithm itself might have to be improved in the future to cope with changes in hardware development, the approach of using micro benchmarks to capture fine-grained hardware details for building machine-aware broadcast trees should still be applicable. Programmers then automatically benefit from an improved version of

the algorithm constructing the tree, even in presence of completely new and fundamentally different hardware without having to change application program code. Consequently, we believe our generator to be useful for future increasingly heterogeneous multicores.

Note that our algorithm is designed for broadcasts trees, but we show in Section 4.6 that it also works well for reductions. However, our design and implementation are flexible enough to use different trees for reductions and broadcasts if necessary for future hardware.

Figure 5.11a shows an example of an adaptive tree as generated for I NL 4x8x2 (cf. Section A).



(a) Step 1: basic adaptive tree

(b) Step 2: applying optimizations

Figure 5.11: Adaptive tree for I NL 4x8x2. The y-axis lists core IDs. Red boxes represent receive operations and orange boxes sends. Arrows between boxes visualize the propagation time, which we assume to be zero. The right-hand side shows the tree as generated by the base algorithm and the left-hand side after applying additional optimizations.

Step 2: Optimizations

Smelt's initial strategy is to build a hierarchical tree, where only one expensive cross-NUMA link is taken per node. This gives a good initial tree, but

leaves room for further improvements. The following optimizations can be applied on top of the basic tree generated in step 1.

Reorder sends: most expensive subtree Smelt's basic algorithm as described before sends on expensive links first. This is a good initial strategy, but can be further improved after constructing the entire tree-topology. In order to minimize the latency of the broadcast, the time until a message reaches the last core has to be reduced. Sending on links that have the most expensive sub-tree intuitively achieves that.

Shuffling: adding further cross-NUMA links As soon as a NUMA node is active, i.e. has received a message or has a message being sent to it already, it will not be consider for further cross-NUMA transfers.

On larger machines, this can lead to an imbalance, where some threads already terminate the broadcast and become idle when they could still further participate in forwarding the message to minimize global latency of the broadcast tree. Let's consider a simple example:

In Figure 5.11a, this is for example visible on core 14, which only sends one local message. It does not consider sending further remote messages, since that would be redundant. However, looking at that topology, it seems natural to further involve that node in the broadcast.

Figure 5.12 shows a zoomed in version with only cores 0 and 14. Core 14 is finished early and does not send any more messages, since each other NUMA node already received a message or has one in flight and all its local nodes finished as well. Core 0 terminates considerably later. The time between core 0 and core 14 finishing is t_{slack} as indicated in the figure.

If an additional cross-NUMA link between core 14's and core 10's NUMA node would terminate faster despite adding another expensive cross-NUMA link, it is beneficial from a purely-latency perspective to allow core 14 to execute this additional NUMA link replacing the link that initially connected node 0 before this optimization.

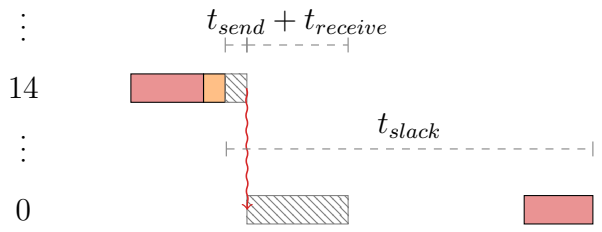


Figure 5.12: Optimization: add further cross-NUMA links

Smelt executes the following algorithm to decide if an additional cross-NUMA link can be beneficial for a concrete pair for cores, i.e. if for two cores v_s and v_e, having a link $v_s \rightarrow v_e$ would reduce the latency of the broadcast. This can be determined from the model. In each iterative step, we select nodes v_s

and v_e as the node that first becomes idle and the node that terminates last respectively. If $t_{send} + t_{receive} < t_{slack}$, Smelt adds an additional cross-NUMA link. Then we iteratively optimize until adding edge $v_s \rightarrow v_e$ does not further reduce the latency of the tree. If this is the case, the resulting tree from replacing previous edge $v_x \rightarrow v_e$ with $v_s \rightarrow v_e$ is always better according to the model. If slower, the algorithm would not have chosen to optimize it and terminated.

The result can be further improved by sorting the edges. Hence, after each "shuffle"-operation, we reorder the scheduling of sends on each outgoing connection of a core by the cost of the receiving core's sub-tree as described in the previous section.

Figure 5.11b visualizes this forI NL 4x8x2: compared to the base algorithm, our "shuffle" operation detects an opportunity for adding an additional cross-NUMA link. Furthermore, it re-orders several message schedules according to the cost of the neighbors subtree. For example, node 23 executes its cross-NUMA connections now in a different order.

Finding the optimal solution

Despite being a type of minimum spanning tree, traditional graph algorithms cannot be used to solve the MST problem in our context as they do not consider the edge priorities. In fact, finding a broadcast tree in the telephone model for an arbitrary graph is known to be NP-hard [SCH81]. A brute-force approach to the problem is not feasible as the search space grows rapidly. To obtain the best tree given a set of nodes n, we need to construct first all the possible trees with n nodes. This the Catalan Number [DZ80] of order $n-1$ (C_{n-1}). Moreover, there are $n!$ possible schedules per tree. Hence the number of possible configurations is shown in Equation 5.1.

$$Ntrees = n!C_{n-1} = \frac{(2n-2)!}{(n-1)!} \tag{5.1}$$

Assuming 1ms for evaluating the model of a single tree, this would take over 6 months for 10 cores. We ran a brute-force search for up to 8 cores (5 hours) for a subset of our machines, and compared the adaptive tree against the optimal solution under the model as well as on real hardware. The estimated runtime by our Tree Generator predicts that Shoal's adaptive tree has a relative error of 9% versus the optimal solution determine by the brute-force approach. If both versions are executed on hardware, the error is larger at around 13% on average.

Note that our Smelt algorithm has been designed for large multicore machines, and evaluating its optimality for configurations of only 8 cores does not show the full potential of our methodology. Unfortunately, due to the high cost of calculating the optimal tree with a brute-force approach, we were not able to extend this validation to bigger machines.

Implementation

The runtime library of Smelt is written in C/C++ and takes the generated tree configurations as input. It allows the programmer to easily implement machine optimized higher-level protocols by abstracting the channel setup and message-passing functionality. Currently, a Smelt application runs as a single, multi-threaded process. Our library is portable and we provide support for Linux and the Barrelfish OS [Bar15]. We now explain Smelt's interface and message-passing aspects.

Runtime

The Smelt runtime is an OS- and architecture-independent C++ library, but all experiments in this thesis have been conducted on Linux. The runtime allows the programmer to easily implement machine optimized higher-level protocols by abstracting the required channel setup and message-passing functionality. This runtime is structured in two layers. We first explain the transport layer and secondly the collective layer with its core concepts, interfaces and abstractions.

Transport layer

This layer provides standalone peer-to-peer message-passing functionality including send, receive and OS-independent abstraction for setup. Smelt's transport layers provide reliable, in-order delivery of messages. Smelt provides a synchronous as well as an asynchronous interface. Hardware already provides in-order delivery, since writes on memory are visible in the same order as they have been executed. In addition to that, flow-control has to be provided in order to guarantee that message queues are not overflowing on the receiver.

Our message-passing implementation is a Linux version of UMP originally implemented for Barrelfish [BBD+09], which is in turn inspired by URPC [BALL91]. UMP consists of a circular buffer of cache line-sized slot residing in shared-memory. Each message contains a header which includes the sequence number and epoch to identify new messages and to provide flow control. Each cacheline has one producer and one consumer to minimize the impact of the cache-coherence protocol. The cache lines holding messages are modified only by the sender. The receiver of a message is polling on an epoch bit, which is toggled by the sender once the entire message has been written. The receiver periodically updates a cacheline with the sequence number of the last received message. This line is checked by the sender to determine whether a slot can be reused. Compared to the original UMP implementation, we disabled sleeping and allocate a separate memory location for acknowledgments, rather than having them as part of the message.

Smelt abstracts message-passing implementations by a *queue-pair* and therefore allows to switch the transport backend transparently. Messages are abstracted using Smelt messages which encapsulate payload and length to be sent over the transport layer.

```
// UMP control word
union smlt_ump_ctrl {
    struct {

        smlt_ump_idx_t epoch;    // UMP epoch
        smlt_ump_idx_t last_ack; // UMP header
    } c;
    smlt_ump_ctrl_word_t raw;    // raw  field
};
// Smelt message
struct smlt_msg
{
    uint32_t words;
    uint32_t bufsize;
    smlt_msg_payload_t* data;
};
```

The interface for send and receive is then the following:

```
errval_t
smlt_queuepair_send(struct smlt_qp *queue_pair,
                    struct smlt_msg *message);
errval_t
smlt_queuepair_recv(struct smlt_qp *queue_pair,
                    struct smlt_msg *message);
```

Sending or receiving a message may block if the channel is full or empty respectively – whether or not such an operation will block can be queried.

The transport layer abstraction supports multiple backends that must adhere to the specification of a Smelt queuepair. We implement two different backends for shared-memory machines and believe it to be a good foundation for future extensions to inter-machine communication. In addition to our UMP-variant, we also integrate the FastForward [GMV08] backend, a cacheline optimized, lock-free queue into Smelt. Throughout the evaluation of this paper, we omit FastForward for brevity and focus on UMP. The modularity of Smelt will enable us to integrate message-passing backends over IP or RDMA protocols in the future.

Collective layer

The collective layer builds upon the transport layer and provides machine-aware, optimized group communication primitives based on broadcasts. Such

collective operations involve one or more Smelt-threads in the system and rely on Smelt's core concepts topologies and contexts.

We first introduce two fundamental concepts of Smelt: topologies and contexts. Based on that, we describe our basic collective operations: broadcasts, reductions and barriers.

Topology A Smelt-topology describes the communication structure for the collective operation. It defines which participants are part of this communication group, i.e. the multicast group. Normally, the topology is generated at program initialization time from the machine model as described in Section 5.3.2. For debugging purposes, we also allow to load the topology from a configuration file. This guarantees determinism.

Contexts Smelt takes the topology description as a blueprint for creating the required transport links. Smelt calls a fully instantiated topology a context. which are encapsulated in a context. A topology can be used to create multiple contexts. Collective operations require a valid context to identify the parent and child nodes. Each context has a distinct node which is the root of the tree.

Broadcast Smelt's broadcast implements an atomic broadcast, i.e. is reliable and ensures that if one node receives a message, all other nodes will also receive this message and that all nodes see messages in the same order.

```
errval_t smlt_broadcast(struct smlt_context *context,
                        struct smlt_msg *message);
```

Smelt's broadcast starts at a fixed per-context root through which all messages are sent: the root therefore acts as a sequentializer. The sequentializer, together with the FIFO property of the edge links and reliable transmission, implements the atomic broadcast property.

It is possible to have multiple contexts with different roots. However, in that case, each core has to poll several memory locations associated with multiple endpoints from different trees to check for new messages. This increases the receive overhead on each core. More importantly, it also increases the latency, as multiple channels have to be polled and the arrival of a message might only be detected after a full round of checking all other channels before the new message is eventually discovered.

Note that using the root node as sequentializer and proxy for broadcasts requires one more message to be sent and therefore adds one additional message transfer to the end-to-end latency of the broadcast. This proxy message cannot be overlapped with other computation and happens before parallel computation is possible by spreading out the tree. This will potentially be a problem on smaller machines, as a higher fraction of the total execution time of the broadcast is spend executing the proxy message.

Another problem with the sequentializer approach is that the sequentializer node has to potentially forward messages on behalf of every other core. To sense when other cores want to broadcast, the sequentializer has to pull a cache line for each potential sender core. Consequently, the root has to potentially poll $n - 1$ channels to detect if any other node wants to send a message. This is not feasible in many cases. To mitigate the latter, Smelt makes use of the fact that many applications know who the initiator of a broadcast will be (see 1Paxos in Section 5.5.8 as an example) and that the root can be chosen arbitrarily for others (such as barriers). The interface hence allows programmers to specify which core is initiating the broadcast, which allows the sequentializer to poll only for messages from this single sending node.

Implementing a sequentializer node also provides a bottleneck for high-throughput applications. However, as our work is mostly looking into low latency rather than high throughput applications, we found this not to be a problem for the programs we used in our evaluation.

Finally, due to the flexible concept of contexts, we think that Smelt's interface allows any of the described options to implement a broadcast. This is important since the choice is likely applications-specific.

Reduce A reduction is an operation, which blocks until a node receives an intermediate result from all its children. It then aggregates these values and forwards it to its own parent node. The procedure is repeated until the root of the tree calculates the final aggregate.

Reductions do in-network processing on each node from payload received from all children, and pass the new value to the parent node. The message flow for reductions is significantly different from broadcasts. For a given tree topology, messages are sent from children to their parents, i.e. a send operation for a broadcast becomes a receive operation if the reduction is executed on the same tree. Furthermore, sends from children can be executed in parallel while receive operations have to be processed sequentially on the parent. This is inverse to the execution flow of broadcasts.

Despite these differences, we use the same tree as for broadcasts for reductions as well. The final value can then be obtained from the root. While this does not generally provide the best possible solution for reductions, this approach keeps the design simple while still achieving good performance in our evaluation. However, we expect that in the future a separate tree for reductions and stored in its own context.

```
errval_t smlt_reduce(struct smlt_context *context,
                     struct smlt_msg *input,
                     struct smlt_msg *result,
                     smlt_reduce_fn_t operation);
```

The reduction takes an `operation` argument pointing to a function that

implements the reduce-operation. At the root, the `result` parameter contains the reduces value.

Barrier With the basic collective operations `reduce` and `broadcast`, we implement a barrier as shown in the code below. Note that we use the optimized zero-payload variants of `reduce` and `broadcast`.

```
void smlt_barrier(struct smlt_context *context) {
    smlt_reduce(context);
    smlt_broadcast(context);
}
```

Despite its simplicity, our `barrier` outperforms or is comparable to state-of-the-art implementations, as we show in our evaluation (Sections 5.5.5, 5.5.6, and 5.5.7). This demonstrates that a highly-tuned generic machine-aware broadcast as implemented by Smelt can be used for higher-level protocols that benefit automatically from Smelt's optimizations.

Two-phase commit We furthermore implement a simple non-fault-tolerant agreement protocol: two-phase commit. It is implemented as a broadcast followed by a reduction. If the request was accepted by every node, the initiator informs all other nodes with another final broadcast.

Evaluation

Section A in the appendix shows the characteristics of the machines we are using for the evaluation of Smelt.

Message passing tree topologies

Here, we evaluate the performance of atomic broadcasts, reductions, barriers and two-phase commit. We conduct these experiments using different tree topologies as described in Chapter 5.2.4 on a wide range of machines. We show that the choice of the tree topology matters and that there is no tree topology that works well across all protocols and machines, even for topologies that consider the NUMA hierarchy of the machine.

We measure how long it takes until every thread has completed the execution of the collective operation. We avoid relying on synchronized clocks and do this by introducing an additional message to signal completion: for atomic broadcast and two-phase commit a distinct leaf sends a message to the root. We measure the time until the root (i.e. the initiator of the operation) receives this message. We repeat this for all leaves and select the maximum time among them. For reductions, this is reversed and the root sends the message to the leaf. For barriers, we measure the cost on each core

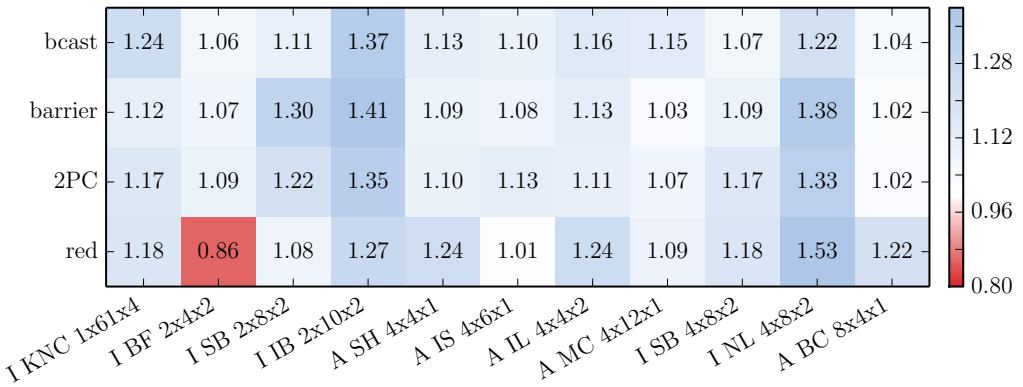


Figure 5.13: Speedup of Smelt compared to the best pre-generated tree topology on each machine. Machines are ordered by the number of sockets as indicated by the label. A white box and 1.0 means that Smelt performs the same as the best reference tree topology; a number > 1.0 and blue label means that Smelt is faster. (more details ↗)

and take the maximum. We repeat the experiment 10.000 times and collect 1.000 data points.

Comparison with the best pre-generated topology

The heat-map in Figure 5.13 shows the speedup of Smelt when executing an atomic broadcast ("bcast"), reduction ("red"), barrier and a two-phase commit ("2PC"). For each, we compared with the best pre-generated tree on each machines. For example, if the "cluster" topology is the best tree topology beside Smelt on a machine, we use that as a baseline. Machines are sorted by the number of sockets, i.e. machines on the right-hand side of the plot are larger. As expected, the benefit of using Smelt is larger on bigger machines This is because smaller machines are less sensitive to the tree topology.

Shoal not only matches the best tree topology on each machines, but also manages to achieve an average speedup of 1.16 over all machines peaking at a speedup of up to 1.24x compared to the best static tree on AMD (A SH 4x4x1) and up to 1.53x on Intel (I NL 4x8x2).

Smelt's performance for barriers and two-phase commits is similar to that of atomic broadcasts. However, the performance of reductions only achieves an average relative speedup of 1.07x. On I BF 2x4x2 and I IB 2x10x2 Smelt performs significantly worse than the best other topology on these machines (best configurations on both machine: cluster). This is a result of our tree being tuned for broadcasts rather than reductions. Tuning the outdegree of the tree as done for the atomic broadcast does not apply to the reduction, since send operation in the trees become receive operations in an reduction. While probably worth further investigation — particularly for potentially

more heterogeneous multicore machines in the future —, we found using the same tree for both broadcasts and reductions reasonably good on most machines and hence decided to use the same in both directions for simplicity.

In summary, for an atomic broadcast, the cluster tree topology manages to be the second best topology on 5 machines, the cluster tree topology on 3, and the Fibonacci tree as well as the MST at 1 of the machines each. The average speedup of Smelt over the best other topology across all machines is 23%.

To conclude, this experiment shows that even when the best pre-generated topology for a concrete machine is known, Smelt still manages to improve the performance.

Breakdown for selected machines

Figure 5.14 shows the detailed comparison of two 4-socket AMD machines (A BC 8x4x1 and A IS 4x6x1) and two 4-socket Intel machines (I NL 4x8x2 and I SB 4x8x2) when executing a broadcast, a reduction, a barrier and a two-phase commit on the static tree topologies introduce in Section 5.2.4 and Smelt.

Our results support the claim that there is no clear best pre-generated topology for all machines and the choice depends on the architecture as well as the workload. As expected, sequentially sending messages and the badtree topology result in a significant slowdown compared to all other topologies. The other hardware oblivious trees, binary and Fibonacci, perform comparable but suffer from using too many inter-socket messages. The cluster topology takes the NUMA hierarchy into consideration and performs well in most of the cases but not all of them (e.g. barriers on A BC 8x4x1 or atomic broadcast on I NL 4x8x2 and I SB 4x8x2). This is because of using a fixed outdegree in each nodes and the rather static topology in which the send order of the messages is not optimized. The cluster tree-topology is similar to what the adaptive tree generates: it builds a hierarchical tree, where communication is first executed between NUMA nodes followed by executing another tree locally within each NUMA node. In our case, both the cross-NUMA and intra-NUMA trees are binary trees. However, the cluster tree topology fails to select the outdegree of nodes ideally based on the difference between send and receive cost, which causes processors that are involved in the message exchange early up to idle rather than sending additional messages.

Despite using machine characteristics to select a minimum spanning tree with the minimal total cost in the case of sequential execution, the MST topology does not consider the protocol's communication patterns nor tries to maximize parallelism. While it achieves good performance on I NL 4x8x2, it works badly on others.

In contrast, Smelt's adaptive tree achieves good performance across all configurations due to the fact that it uses hardware information enriched with real measurements to capture fine-grained performance characteristics of the

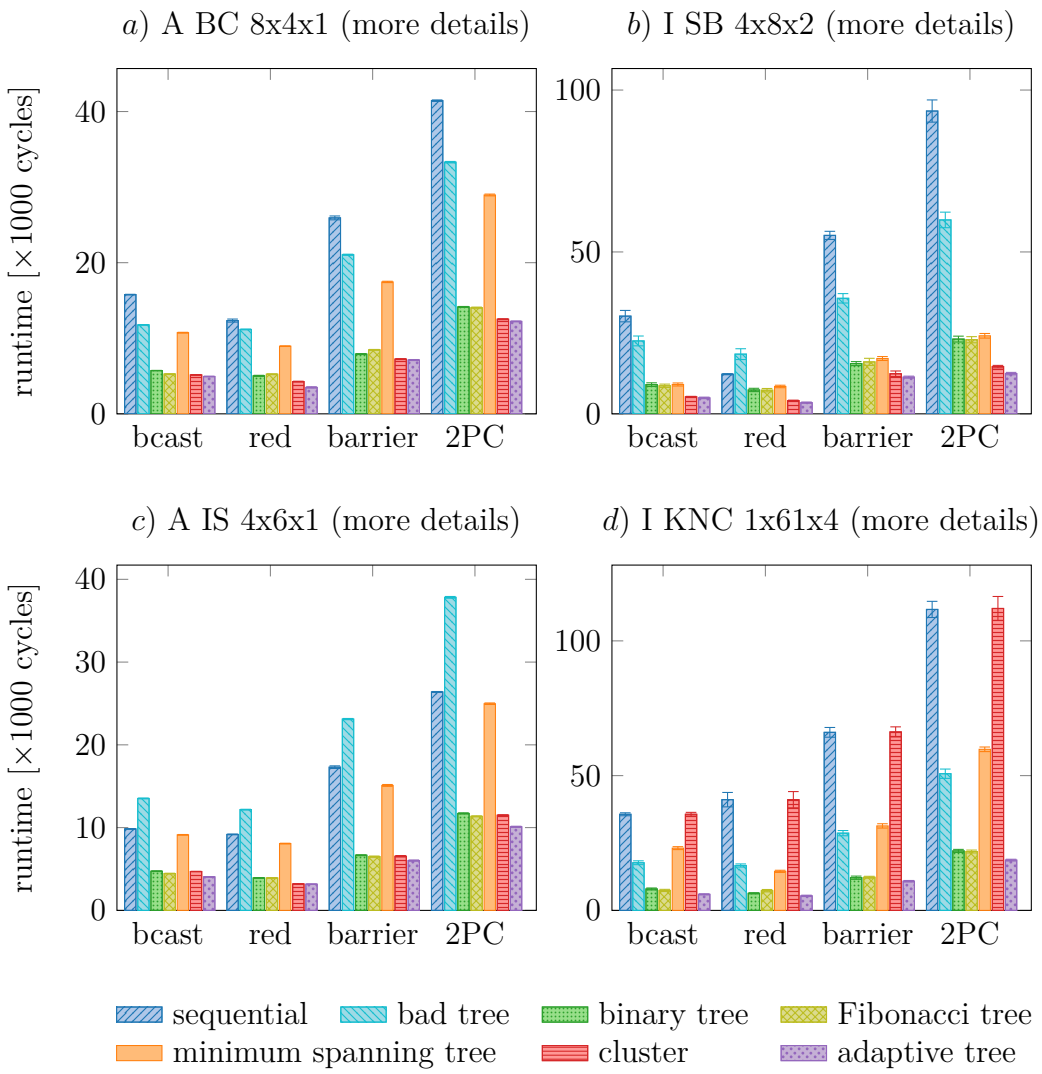


Figure 5.14: Performance of various tree topologies for broadcast (bcast), reduction (red), barrier and two-phase commit.

machine and adapt the message topology as well as scheduling accordingly. Our results show that *generating* a tree for a machine based on our machine model as described in Section 5.3.2 achieves good results without manual tuning and that the tree topology matters. We also show that there is no static topology that performs best on all machines even for topologies that are considering the NUMA hierarchy. In contrast, Smelt is able to adapt to a wide set of micro-architectures and machine configurations without manual tuning.

Evaluation of adaptive tree optimizations

In Section 5.3.2.2, we present optimizations for reordering and shuffling messages, which iteratively tune the basic algorithm such that idle cores send further cross-NUMA messages if the predicted execution of the broadcast leaves enough slack for it.

Note that according to the model, the optimizations always return a better tree. However, that might not necessarily lead to better performance on hardware as the machine model might not be precise enough to accurately choose whether further optimizations can be applied to a tree or not.

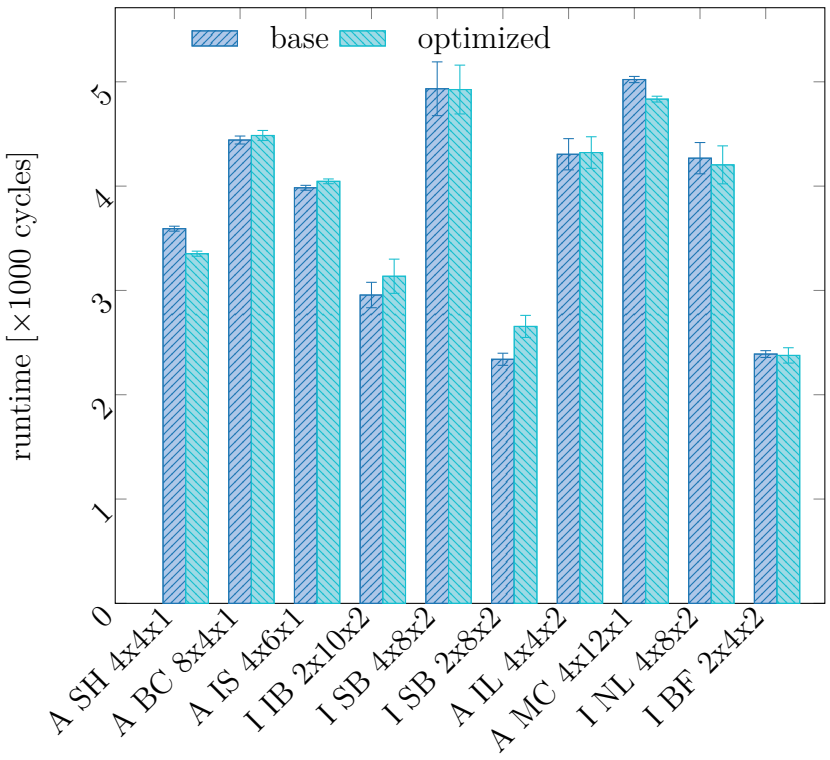


Figure 5.15: Comparison of the optimized version of the adaptive tree (Section 5.3.2.2) to the basic algorithm.

In Figure 5.15, we compare the performance of the adaptive tree in the basic version to a tree with optimizations enabled. Our results suggest that the optimized version of the adaptive tree does not provide significantly better performance for broadcasts, as trees in that case are fairly regular allowing the basic adaptive tree to build a very good tree topology already. Additional cross-NUMA links as added by the optimizations are unnecessary, as the more expensive links across nodes cannot deliver messages to cores earlier compared to locally sending messages even though local sends can often be started only later in time when one of the cores on each NUMA node already received the message.

However, our optimizations might provide a good starting point for incre-

mental changes to the tree, for example as a consequence of hardware failures or changes in the system's load that require reconfiguration.

Accuracy of the model

We now evaluate the accuracy of our model (Chapter 2) for atomic broadcasts. In Figure 5.16, we show our results in a heat map for all topologies on all machines. A value of 1.0 indicates that the Simulator's prediction is 100% accurate compared to execution of an atomic broadcast on real hardware, values < 1.0 indicate that the Simulator underestimates the cost of a broadcast and > 1.0 means an overestimate of the cost of executing the broadcast on real hardware.

Note that it was not the goal of our Simulator to accurately predict the performance of message passing algorithms on multicore hardware, but to provide a solid foundation for generating machine-aware broadcast trees without programmer's invention. Our focus was more on being able to generate the model automatically from micro benchmarks rather than accuracy. For higher accuracy, a more fine-grained modeling of the cache-coherence protocol as in [RH16] would likely be required.

Note that, as with the evaluation of tree topologies in Section 5.5.1, our measurements on hardware and consequently also the Simulator's prediction include the cost of one additional message sent from each leaf node of the tree in turn to notify the root of completion of the broadcast. From each leaf's measurement, we take the maximum. The Simulator considers this by simulating this additional message as well.

Figure 5.16 reveals that our prediction is less accurate for machines with SMT threading enabled. This is likely a consequence of the non-deterministic scheduling of both hyperthreads to physical contexts. Smelt is not optimized for SMT threads either: e.g. it indefinitely polls channels until a message can be found. While in the case of our pairwise measurements (Section 2.4.2.2) all but two threads are idling keeping the side effects of SMT threads minimal, here each threads in the system constantly polls its inward channels. The model's prediction has an average relative error of 20.6% compared to execution on real hardware for machines without SMT threads.

With all machines including SMT, the prediction is worse. On average the Simulator then has a 34% relative error compared to execution on real hardware. The adaptive tree and cluster topology have the highest average error in their prediction (41% and 38% respectively) and the bad tree is most accurate (38% error). On I SB 2x8x2, the prediction is worst at an average relative error of 83% and best on I SB 4x8x2 with an average error of 11% despite SMT threads.

While a more precise model would likely be beneficial, we found our machine model precise enough to build efficient trees for current multicore machines (Section 5.5.1).

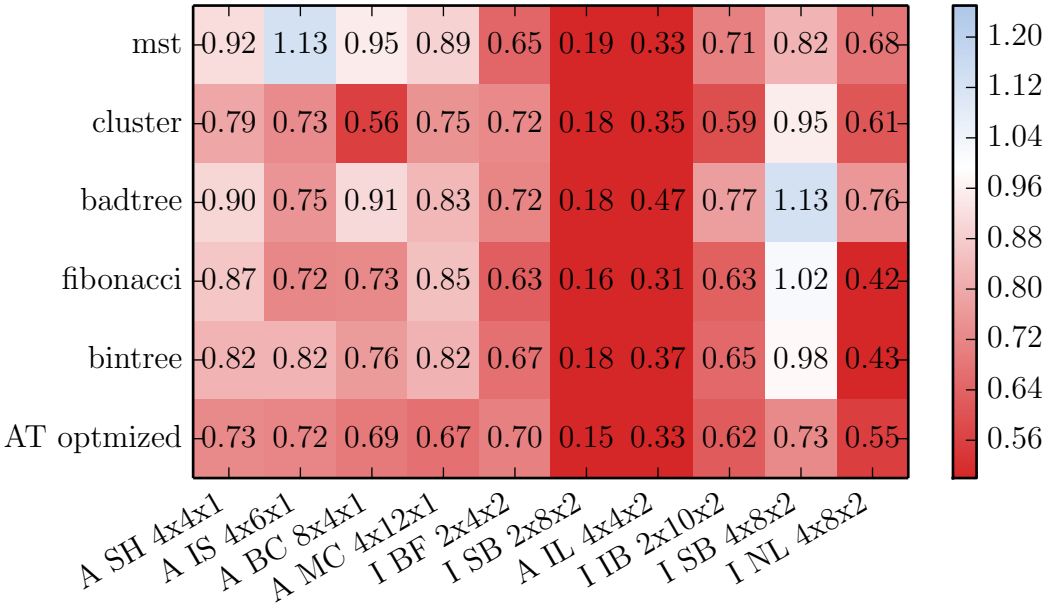


Figure 5.16: Evaluation of the Simulator's precision. We show the relative error of the Simulator compared to execution of hardware. "AT optimized" refers to the optimized version of the adaptive tree (Section 5.3.2.2). Machines with SMT enabled are listed first and otherwise sorted by the number of NUMA nodes.

Multicast

Certain workloads require collective communication within only a subset of the available cores. We evaluate this multicast-scenario by running the atomic broadcast benchmark from Section 5.5.1 and vary the number of threads from two up to the maximum number of hardware-threads on each machine. For this, we allocate threads round-robin from NUMA node one at a time, e.g. when showing four cores on a machine with four NUMA nodes, we would place one thread on each NUMA node. To visualize the effect of other thread allocation strategies, we fill NUMA nodes on I BF 2x4x2 before allocating threads from other nodes rather than allocating them from nodes one at a time.

Figure 5.17 shows the multicast scaling behavior of Shoal compared to the static tree configurations on two Intel (I IB 2x10x2 and I BF 2x4x2) and two AMD machines (A IL 4x4x2 and A IS 4x6x1). For a low number of cores it is often best to send messages sequentially or using an imbalanced Fibonacci tree: we observe this behavior in both Figure 5.17b and 5.17d. This is an especially useful configuration to implement consistent updates to data replicated on a per-NUMA basis as we will show later in Section 5.5.9. With that, all communication links are remote and the hierarchical cluster approach simply produces a binary tree between nodes. When more cores

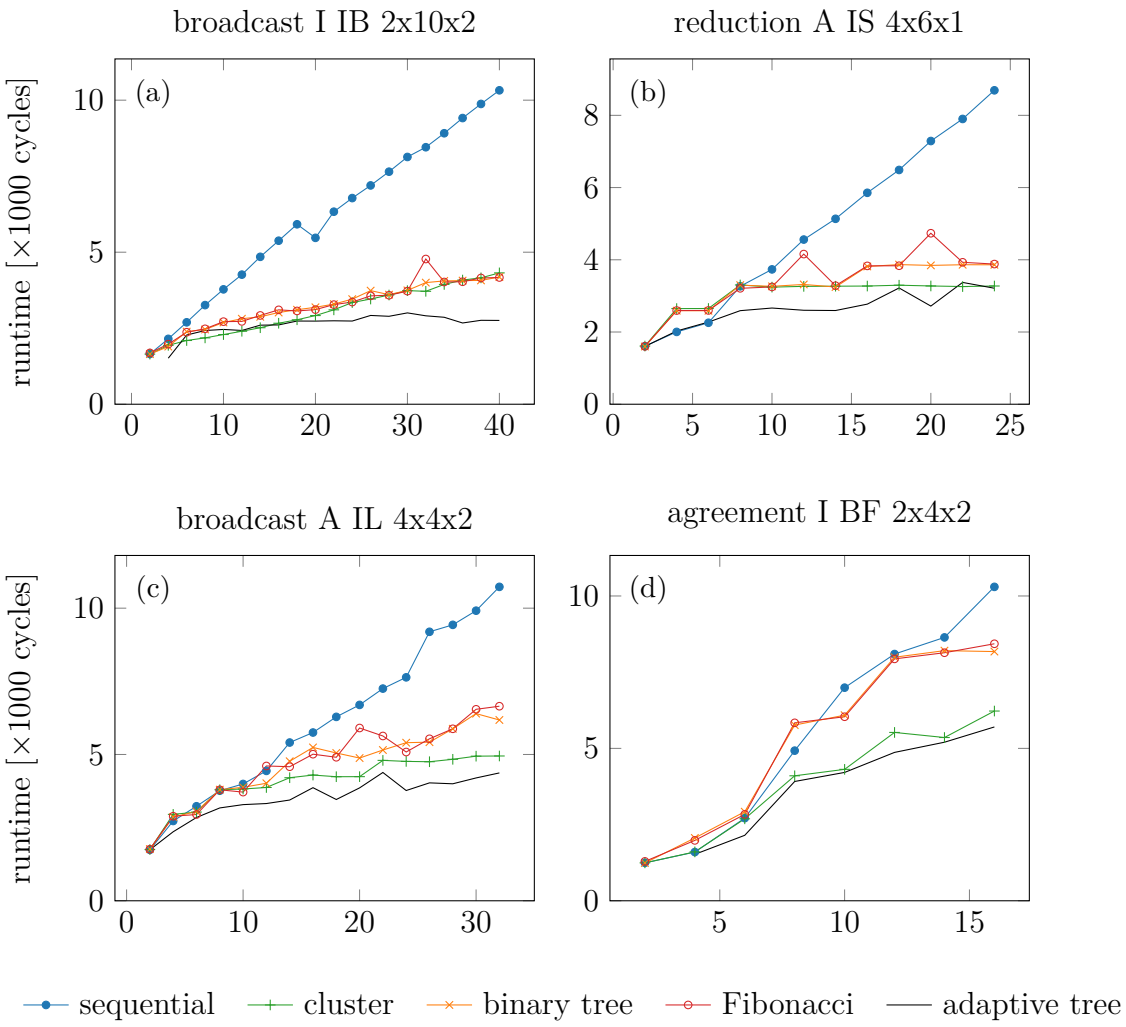


Figure 5.17: Comparison of Smelt with the three best static topologies when executing a multicast

are involved and the effects of local vs. remote communication becomes more obvious, the cluster topology in many cases performs best out of the static configurations. In summary, the choice of topology does not only depend on the machine, but also the participating cores for the multicast group.

Further, the performance benefit is often higher for Smelt in intermediate configurations: for example in Figure 5.17b, the maximum speedup over the best static topology is 1.25 when 12 cores as to compared to 1.02 for a reduction involving all 24 cores. The maximum negative speedup of Smelt is 0.97 on A IS 4x6x1 for 22 cores. The larger benefit when using Smelt for these partial multicast groups is likely due multicast configurations often being less regular than broadcast trees, in which case our iterative optimizations described in Section 5.3.2.2 show the biggest benefit.

In summary, our results show that Smelt scales well for multicasts and outperforms other topologies in many cases.

Comparison with MPI and OpenMP

We compare Smelt with two established communication standards: MPI and OpenMP. MPI (Message Passing Interface) [Mes09] is a widely used standard for message-based communication in the HPC community. MPI supports a wide range of collective operations, including broadcasts, reductions and barriers. Furthermore, MPI libraries provide several highly tuned channel implementations and optimizations for shared-memory systems.

OpenMP 4.0 [Ope08] is a standard for shared-memory parallel processing and is supported by most compilers, operating systems and programming languages. The OpenMP runtime library manages the execution of parallel constructs. If not specified otherwise by the `nowait` directive, threads are implicitly synchronized after a parallel block or explicitly by the barrier directive.

We compare the collectives of MPI (OpenMPI v1.10.2) and OpenMP (GOMP from GCC 4.9.2) with Smelt. For MPI we compare broadcasts, reductions, and barriers. OpenMP only provides reductions and barriers, although functionality similar to that of a broadcast could be implemented using OpenMP's `COPYPRIVATE` directive.

For each of them, we execute our experiments 3000 times and take the last 1000 measurements. The broadcasts are executed with a one byte payload and reductions have a single integer payload. In the beginning of each round, we synchronize all the threads with two additional barriers so that all the threads enter the collective operation at the same time and measure only the cost of the single barrier following them. This is different from the benchmark in Section 5.5.1, where we were able to send an additional notification message back to root to indicate completion since all systems were tree-based.

Figure 5.18 shows the results for some of the larger machines. Smelt outperforms MPI for all tested collective operations with speedups between 1.3x and 4.1x. OpenMP performs worse than MPI and has a high standard error. On some machines, barriers perform better than reductions, on others reductions are faster. Smelt has a speedup between 1.8x and 10.3x. OpenMP's performance breaks down on A IL 4x4x2 with barriers being more than 1000x slower than Smelt's, perhaps as a consequence of the complexities of its Bulldozer architecture.e

OpenMP in detail

Since OpenMP uses explicit and implicit barriers after each parallel construct, we extend the evaluation to demonstrate how Smelt can be used to improve the GOMP library [Fre]. GOMP's standard barriers are based on atomic instructions and the futex syscall [Dre11, FR02, Lin] on Linux. We replaced GOMP's barrier with Smelt and compared it against the vanilla version. As workload we took "syncbench" and "arraybench" from the EPCC

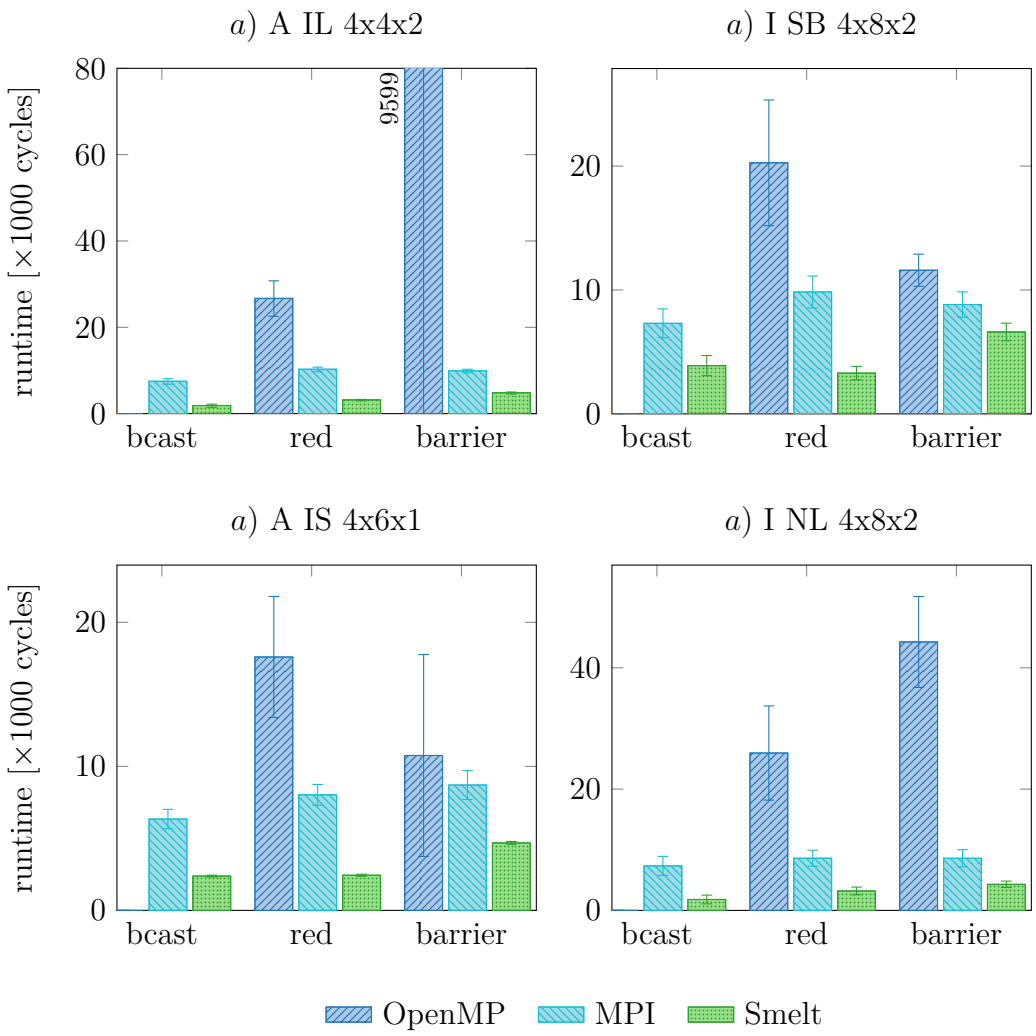


Figure 5.18: Comparison with MPI and OpenMP.

OpenMP micro-benchmarks suite [Mar15] using standard settings and 5000 outer repetitions. We ran the benchmark on all available threads.

The results of the benchmark are shown in Figure 5.19. Overall, Smelt performs significantly better or comparable to the original GOMP barriers. In the "BARRIER" micro-benchmark, we achieve up to 5.54x for A IL 4x4x2 and 2.23x for I SB 4x8x2. These results show that replacing the standard barriers in GOMP with Smelt reduces the overhead for synchronization significantly. Note also that especially on A IL 4x4x2, the standard error becomes significantly lower when Smelt is used. We account this to a more deterministic execution of the tree-based implementation compared to the use of shared data structures, where a suboptimal order of threads accessing it can lead to significant cache coherence traffic. Furthermore, OpenMP uses futexes, which can induce a large operating system overhead.

Due to OpenMP's popularity, many programs written on top of OpenMP

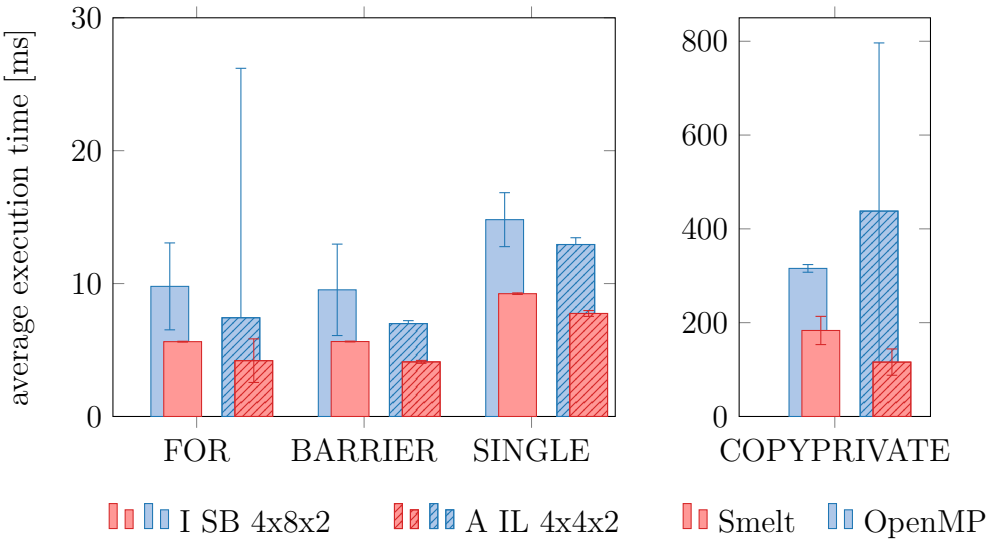


Figure 5.19: Results of FOR, BARRIER, SINGLE, and COPYPRIVATE for the the EPCC OpenMP benchmark suite. Smelt is consistently better.

can automatically benefit from Smelt's efficient barrier implementation without programmers having to tune their program source code.

Barriers micro-benchmarks

Barriers are important building blocks for thread synchronization in parallel programs. We compare our barrier implementation with the state-of-the-art MCS dissemination barrier [Amp] (parlibMCS ▨) and a 1-way dissemination barrier that uses atomic flags [RH15] (dissemination ▨). We show that our simple barrier implementation, based on broadcast and reduction, can compete with highly-tuned state-of-the-art shared-memory implementations.

This works because of our highly tuned and machine-aware broadcast tree, that does not only minimize the number of messages that have to be sent (as MCS and butterfly barriers do), but also considers link costs and maximizes parallelism by selecting the outdegree in a way that avoid nodes from being idle before the synchronization is terminated.

In this evaluation, we synchronize threads with each barriers in a tight for-loop of 100'000 iterations. This is yet another barrier benchmark and cannot be directly compared with the previous sections.

The results in Figure 5.20 show significant differences between machines. Whereas on A IL 4x4x2 both the parlibMCS and the dissemination barrier performs slightly better than Smelt, Smelt is up to 2.5x faster on I NL 4x8x2 and 3.0x times faster on I SB 4x8x2. With this evaluation we have shown how a competitive barrier can be implemented easily using Smelt's hardware aware collective operations.

Across all tested machines, Smelt improves performance of the MCS bar-

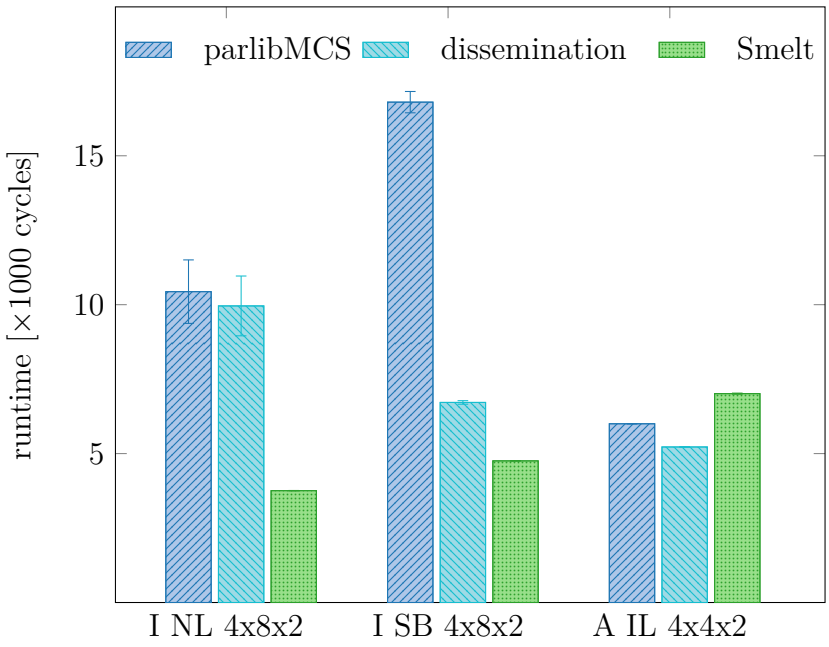


Figure 5.20: Detailed analysis of OpenMP

rier by 60% and the performance of the dissemination barrier by 5%.

Streamcluster

We now analyze the performance of Smelt barriers in a real application using PARSEC's Streamcluster [BKSL08] benchmark, which solves the online clustering problem. We choose this benchmark because it is synchronization-intensive. We evaluate the performance of Smelt's barriers using the optimized adaptive tree compared to PARSEC's default barriers, pthread barriers, and parlibMCS dissemination barrier [Amp]. As extensively discussed in Chapter 4, a good memory allocation matters for good performance of parallel programs. Hence, we evaluate both, the Streamcluster version optimized with Shoal (Chapter 4) as well as its default native memory allocation strategy.

Our results in Table 5.1 shows our results for some of the bigger machines. We confirm that optimizing both memory accesses and synchronization primitives matter for achieving good performance of parallel programs. Furthermore, our results suggest that our simple barrier implementation based on a generic broadcast tree performs better than shared-memory barrier implementations and is competitive with a state-of-the-art dissemination barrier.

In Figure 5.21, we show our performance in Streamcluster on all evaluated machines. Here, we compare the performance with MCS. Smelt achieves comparable performance on most the machines, despite using a simple barrier implementation based on a generic, but machine-aware tree that has not been particularly optimized for barriers.

		pthread	parsec	parlib	Smelt
A BC 8x4x1	no Shoal	211.4 (0.9)	208.2 (2.8)	170.5 (1.4)	161.7 (1.7)
	Shoal	139.8 (0.8)	140.2 (0.2)	112.6 (0.3)	111.6 (0.2)
A IL 4x4x2	no Shoal	122.9 (4.3)	114.4 (5.7)	64.6 (4.6)	66.2 (6.4)
	Shoal	55.1 (1.1)	54.9 (1.0)	38.6 (0.7)	38.7 (0.3)
I SB 4x8x2	no Shoal	316.7 (2.4)	315.2 (2.5)	125.0 (0.9)	124.5 (0.1)
	Shoal	84.1 (0.3)	82.8 (0.8)	28.4 (0.1)	27.9 (0.1)
I NL 4x8x2	no Shoal	193.6 (7.6)	193.7 (5.9)	74.4 (0.9)	60.9 (0.3)
	Shoal	166.5 (1.7)	166.5 (1.8)	54.3 (0.4)	53.4 (0.2)

Table 5.1: Execution time [seconds] of Streamcluster when executing a native workload. Standard error in brackets.

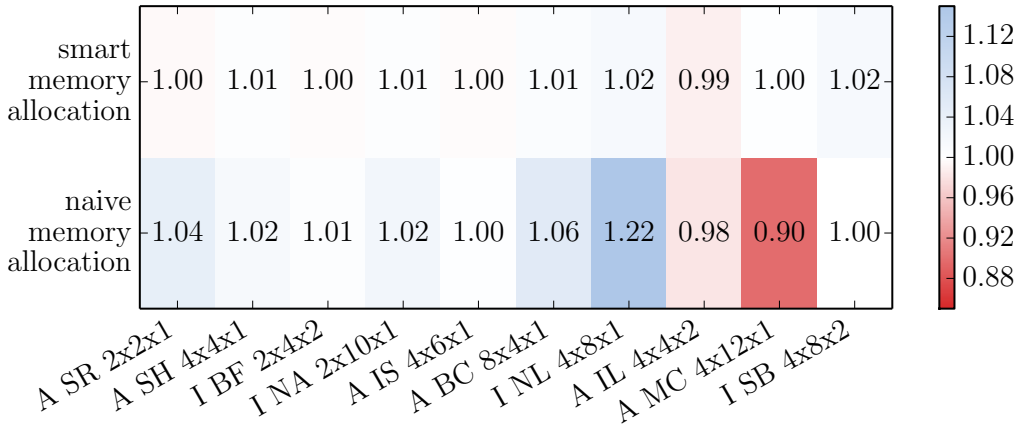


Figure 5.21: Performance of Streamcluster with a state-of-the art Dissemination barrier compared to Smelt's barrier evaluated on a wide range of machines.

Performance of Streamcluster closely follows our observation from Section 5.5.6: for example, the performance improvement on I NL 4x8x2 is biggest, but slightly lower on A IL 4x4x2. On A MC 4x12x1, Smelt has the biggest slowdown of all evaluated machines at around 10%.

When enabling Shoal in addition to Smelt, the choice of the barrier implementation has a lower impact on overall performance. Shoal likely already reduces traffic on the interconnect enough to make it less sensitive to additional traffic from synchronization primitives.

Agreement

We implemented the 1Paxos [DGY14] agreement protocol using Smelt. 1Paxos is a Paxos-variant optimized for multicore environments. Normal operation

is shown in Figure 5.22: a proposer (P) sends a request to the leader (P/L), which forwards the request to the acceptor. Then, a single broadcast from the acceptor to the replicas is needed. We optimized this broadcast using Smelt. Upon receiving the broadcast, the leader responds to the client.

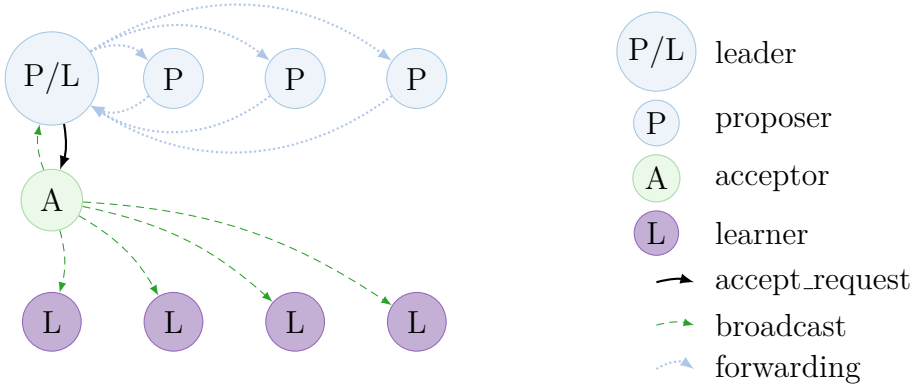


Figure 5.22: 1Paxos in the failure-free case

We vary the number of replicas from 8 to 28 and use 4 cores as load generators which was sufficient to issue enough requests to saturate the system. The measurements are averages of three runs of 20 seconds each. Figures 5.23 present the performance of the agreement protocol and compares it to a simple non-fault-tolerant atomic broadcast with the same number of threads.

The results show that agreement protocol on multicore machines can benefit from an optimized broadcast primitive: using Smelt improves the throughput and response time up to 3x compared to sequential sending when using 28 replicas. As we increase the number of replicas, the sequential broadcast quickly becomes the bottleneck. Our results show that 1Paxos is highly tuned towards multicores as scaling behavior and performance is similar to a plain broadcast.

By using Smelt, we can improve the performance of agreement protocols on multicore machine and improve scalability to larger number of replicas.

Key-value store

We implement a replicated key-value store (KVS) based on 1Paxos to ensure consistency of updates. Several choices for existing key-value stores exists: Redis [Red], for example, is a high-performance in-memory key-value store for shared-memory machines. However, it provides replication only over the network. Communication in that case is based on a socket interface, which is not a good match for shared-memory communication and hence not a meaningful comparison to Smelt. memcached [mem16] As a consequence, it does not provide efficient replication within a single multicore machine.

Due to a lack of alternatives, we felt that rather than modifying one of the existing ones, it would be easier to design our own simple key-value

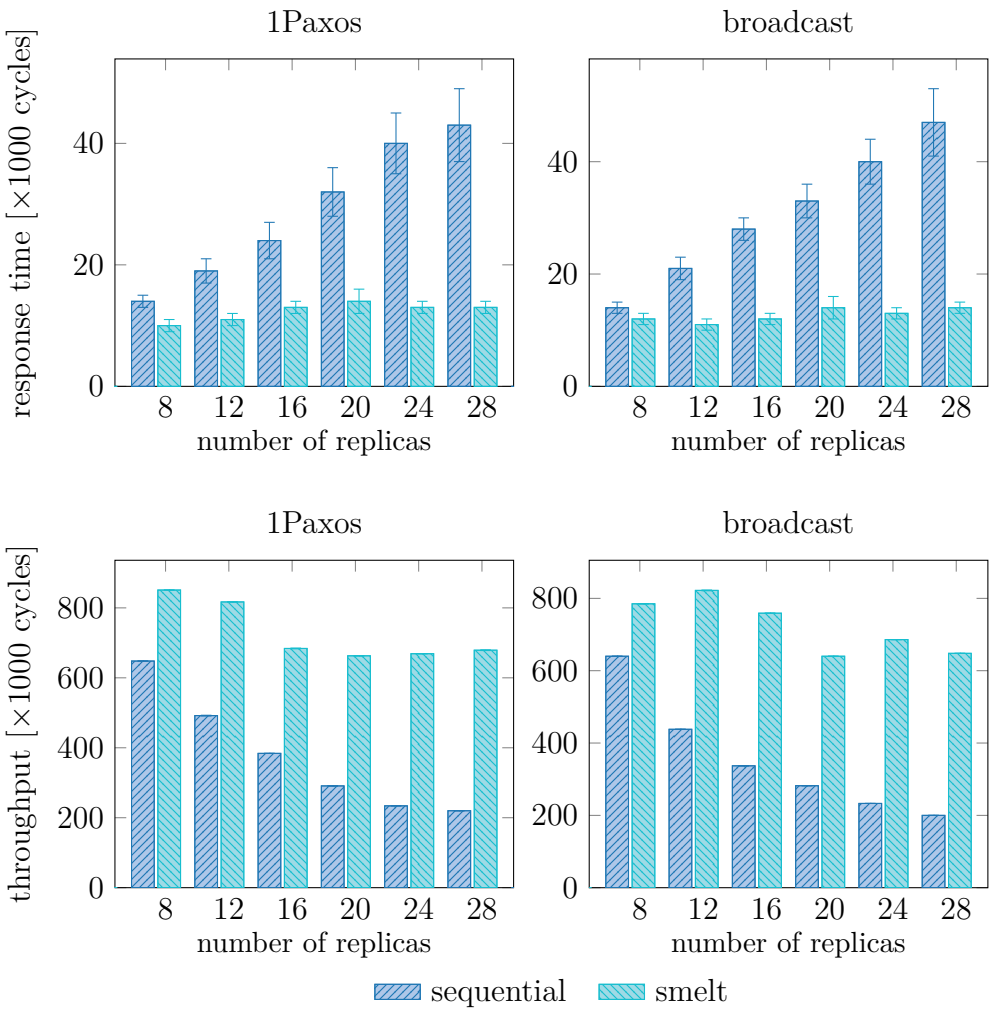


Figure 5.23: Response time and throughput for A IL 4x4x2 when executing 1Paxos and a simple atomic broadcast. For each, we compare a sequential send operation with Smelt's broadcast.

store. In our setup, we place one replica on each of the 8 NUMA nodes of A IL 4x4x2. A varying number of clients connect to their local KVS instance and issue requests. We executed the benchmark for 20 seconds and 3 runs with a get/set ration of 80/20. Our implementation supports a get/set interface and focuses on small keys (8 byte) and values (16 byte) to avoid fragmentation. We justify this as fragmentation simply increases the total number of messages. Reads are served directly by the replica, while writes have to be applied via the atomic broadcast to guarantee the same order of updates on all replicas.

The `set` and `get` performance results are shown in Figure 5.24. Our results demonstrate that, using Smelt, we are able to improve response time and throughput already for a small number of replicas. Under higher load, scalability is better resulting in an up to 3x improved throughput and re-

sponse time for set-requests. Note also that the standard error is lower with Smelt.

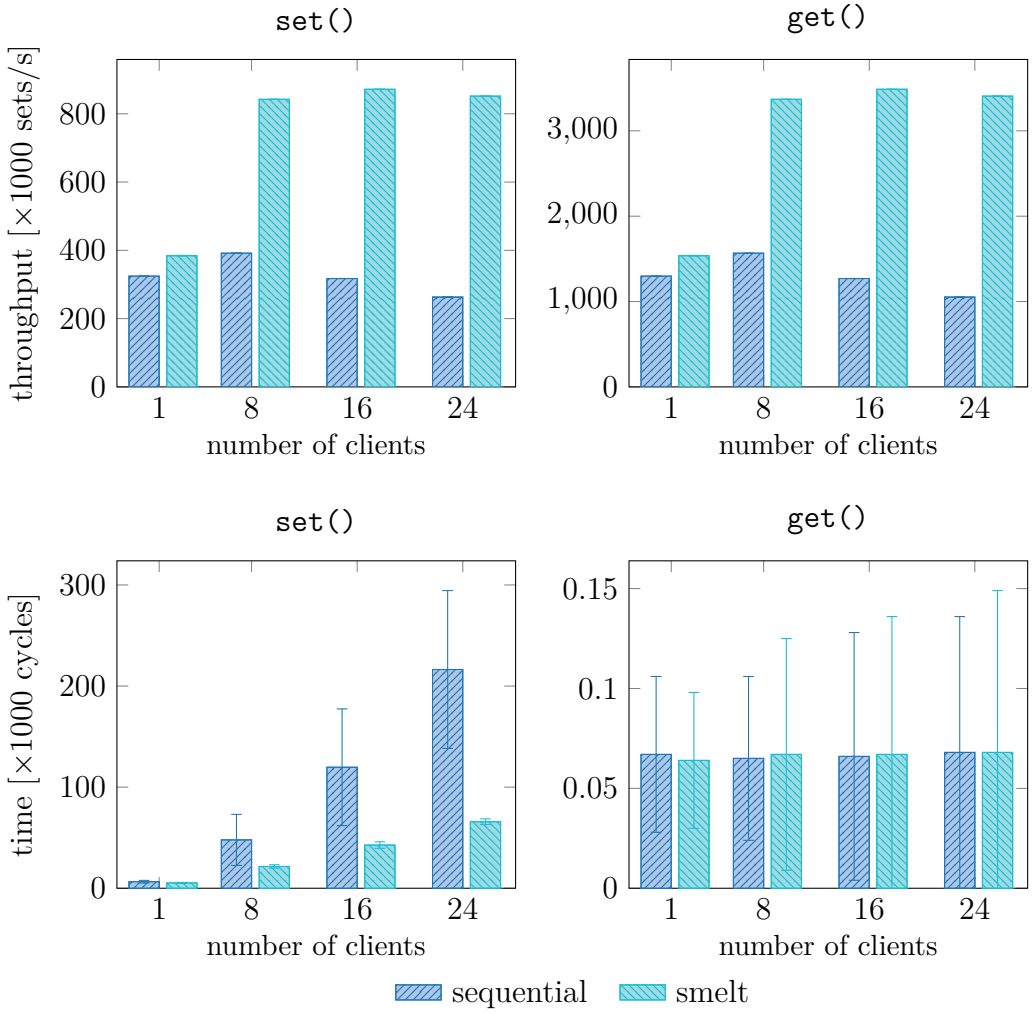


Figure 5.24: `get()` and `set()` time and throughput for A IL 4x4x2

Conclusion and Future work

In this chapter, we re-evaluate tuning broadcast algorithms for multicore machines. We present Smelt, a tool that automatically builds efficient broadcast topologies tuned to machines based on a machine model. Our machine model encodes both static hardware information and costs for sending and receiving messages generated from micro-benchmarks to capture low-level machine characteristics. Smelt provides an easy-to-use API which can be used to build high-level applications on top of it.

We show that adaptive trees generated by Smelt match or outperform the best topology on each of a variety of machines. Barriers implemented on

top of Smelt outperform state-of-the-art algorithms including shared-memory algorithms. Furthermore, we show how to achieve fault-tolerance and good scaling with the number of parallel requests in a in-memory key-value store. Automatically generated broadcast topologies proof to offer high performance without requiring programmers to have detailed understanding of a machine's topology.

Future work

In this chapter, we showed that our adaptive tree is useful for selecting a good machine topology for a wide set of machines automatically based on detailed message-passing characteristics captured in our machine model. It does not just find a topology that matches the best static topology on each machine, but we even improve performance on top the best topology on each machine.

However, we see an even bigger potential for this approach for future hardware:

Failures and system load

Future machines will likely experience partial failures [FKMM15]. That will make the broadcast more complex as underlying channels will be unreliable. A good solution to the broadcast problem will then likely be composed of various transport channels: reliability will have to be build on top of unreliable hardware channels similar to IP's TCP protocol, but a more efficient and lightweight implementation should be used if hardware already provides fault-tolerance, e.g. in fault-tolerance islands.

In case of failures on individual peer-to-peer communication links, the broadcast tree will have to be reconfigured: in contrast to sequentially sending messages, intermediate node and link failures will make the entire subtree rooted in the failing components unavailable until the tree is reconfigured such that the subtree becomes available again through redundant previously unused peer-to-peer channels. We believe that our incremental optimizations of the adaptive tree as described in Section 5.3.2.2 might be a good starting point for this kind of fixes to the tree topology. The goal for reconfiguration then is not just to build another good tree topology, but also to keep the changes required compared to the previous tree configuration small to localize the effect of reconfiguration.

Similar considerations might be applicable to changes in the system load. For example, intermediate nodes in the tree have to execute more work and a penalty in their latency has a large impact on overall broadcast latency compared to leaf nodes. Changes in the system load might therefore make it attractive for the tree to be reconfigured dynamically.

If tightly integrated with the scheduling, threads might also be migrated in response to changes in the system load, which would then require the tree

to be reconfigured similarly.

Rack-scale computing

Given the trend towards rack-scale computing [FKMM15], it is likely that future systems will have a variety of different interconnects stacked on top of each other. There will likely be a different interconnect within a socket, between sockets and between rack machines. This adds another layer to the hierarchy of interconnects, possibly with fundamentally different performance characteristics: propagation time between machines will be higher than within a machine and such links might not be reliable forcing software to build reliability on top of unreliable peer-to-peer message channels.

We believe that modeling and simulation will be useful to automatically build topologies for message-passing-based synchronization as well. With an extension of the model described in Chapter 2 capturing the failures on message links, a simulator could build such topologies similarly to Smelt's approach to solving the multicast problem for multicores.

Algorithm design

Our simulator approach might also be a first step to build an environment where algorithm designs can be evaluated with low overhead for a wide set of typical hardware. This can guide programmers when designing new algorithms. For example, for the broadcasting problem: evaluation using our Simulator immediately shows that sequentially sending messages significantly degrades performance on typical larger multicore hardware. This is in contrast to traditional distributed systems (Section 2.5).

6 Conclusions

Summary

Many parallel programs can suffer from suboptimal memory and, due to mandatory software-parallelism, their performance is often limited by the efficiency of synchronization primitives to coordinate concurrently executing threads. In response to the increasing complexity of multicore machines and the resulting challenges for programmers, we presented a machine-aware memory allocator and synchronization library.

Our approaches firstly rely on collecting application characteristics in the shape of memory access patterns. Secondly, we collect hardware information and enrich them with micro benchmarks where hardware-provided information is insufficient or imprecise. At runtime, this information can be combined to achieve smart and automatic memory management and synchronization.

In this thesis, we have shown how our frameworks achieve good performance of parallel programs on a wide range of multicore machines without programmers having to understand intricate performance characteristics and manually, and repeatedly optimize for the characteristics of a concrete machine.

Directions for future work

Support for dynamic systems

Currently, we are not considering any dynamic properties of the system. For example, we ignore potential other applications running concurrently on the system, which could affect where memory should be allocated and how broadcast trees should be configured. Allocation of memory on already heavily utilized memory controllers should be avoided in favor of others that might be less frequently access. In broadcast trees, node with a high outde-

gree often need to execute more work and should therefore chosen such that they are no co-locate with busy threads of other applications.

Since the program load is not known beforehand, a good static configuration for both memory allocation and synchronization can still be beneficially. Based on that, incremental optimizations according to the system load can be applied as needed.

Rack-scale systems

With machines becoming more complex and hierarchical, automatic tuning is increasingly attractive as well. Our idea of modeling the machine and automatic tuning should still be applicable, although more implementation of backends for synchronization and memory allocation might be required along with new algorithm to choose from them.

Machine failures

Current multicore machines fail as one: either the machine is running reliably or it crashes completely. Trends in hardware suggest that this is no longer possible to maintain in hardware and as a consequence, vendors are starting to think about failure units and how to expose failures to programmers.

This strengthens the need to use partitioning not just for performance, but also for fault-tolerance and is yet another reason to help programmers to automatically replica data where appropriate to make their software portable to different hardware platforms.

In addition to memory management, hardware failures also have implications on synchronization primitives. Reconfiguration might be necessary as a reaction to route message around failing system components. Also, in order to provide reliable communication over unreliable message channels, reliability has to be implemented by software. Ideally, this should be done selectively to achieve good performance: software overheads should be kept minimal and redundant re-implementation of mechanisms to provide reliability avoided if already provided by hardware channels. Automatically selecting backends seems crucial for every more complicated hardware to release programmers from the burden of having to understand performance implications of hardware characteristics on their application's performance.

Hybrids

Our current approach to synchronization is to use message-passing globally in the entire machine. However, it might be beneficially to still use shared memory where processors are strongly connected, e.g. with a shared last-level cache. The result then would be a hybrid system, where shared-memory is used for some parts of the machine and these shared-memory islands then

connected via message-passing links. Our approach similar to what we presented in Chapter 5 combined with an extended machine model would be a natural choice for configuration of such a system as well.

While initial results suggest that message-passing might achieve comparable performance than using shared-memory in the general case, there will almost certainly be hardware platforms where socket internal shared memory is more efficient than a manually constructed message-passing implementation in software on top of what the hardware is actually build for.

A

Machine characteristics

machine	A MC 4x12x1	I IB 2x10x2	I NL 4x8x2
CPU	AMD Opteron 6174	Intel Xeon E5-2670 v2	Intel Xeon L7555
micro arch	Magny Cours	IvyBridge	Nehalem
#cores	4x12x1	2x10x2 @ 2.50GHz	4x8x2 @ 1.87GHz
caches	64K/ 512K/ 4M	32K/ 256K/ 25M	32K/ 256K/ 24M
memory	126G	252G	110G

machine	A BC 8x4x1	I KNC 1x61x4	I SB 2x8x2
CPU	AMD Opteron 8350	Xeon Phi	Intel Xeon E5-2660 0
micro arch	Barcelona	Phi (Knights Corner) Co	SandyBridge
#cores	8x4x1	1x61x4	2x8x2 @ 2.20GHz
caches	64K/ 512K/ 2M	32K/ 512K/	32K/ 256K/ 20M
memory	15G	15G	64G

machine	A SH 4x4x1	A IL 4x4x2	I SB 4x8x2
CPU	AMD Opteron 8380	AMD Opteron 6378	Intel Xeon E5-4640 0
micro arch	Shanghai	Interlagos	SandyBridge
#cores	4x4x1	4x4x2	4x8x2 @ 2.40GHz
caches	64K/ 512K/ 6M	16K/2048K/ 6M	32K/ 256K/ 20M
memory	16G	512G	505G

machine	I BF 2x4x2	A IS 4x6x1
CPU	Intel Xeon L5520	AMD Opteron 8431
micro arch	Bloomfield	Istanbul
#cores	2x4x2 @ 2.27GHz	4x6x1
caches	32K/ 256K/ 8M	64K/ 512K/ 4M
memory	24G	15G

List of Tables

List of Figures

Bibliography

[Ach14] R. Achermann. *Message Passing and Bulk Transport on Heterogenous Multiprocessors*. ETH Zurich, 2014. Master's Thesis, http://dx.doi.org/10.3929/ethz-a-010262232.

[ADADB+03] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. "Transforming policies into mechanisms with infokernel." In *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 90–105. ACM, 2003.

[Adv13] Advanced Micro Devices. "AMD64 Architecture Programmer's Manual Volume 2: System Programming." Online, 2013. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/24593_APM_v21.pdf. Publication Number 24593. Revision 3.23.

[ALM15] N. Aravamudan, A. Litke, and E. Munson. "libhugetlbfs." Online, 2015. http://libhugetlbfs.sourceforge.net/.

[Amp] Amplab, UC Berkeley. "PARLIB, MCS Locks." Online. http://klueska.github.io/parlib/mcs.html. Accessed 05/10/2016.

[ASK+07] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. "Experience Distributing Objects in an SMMP OS." *ACM Transactions on Computer Systems*, vol. 25, no. 3, 2007.

[BALL91] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. "User-level Interprocess Communication for Shared Memory Multiprocessors." *ACM Transactions on Computer Systems*, vol. 9, no. 2, 175–198, 1991.

[Bar15] Barrelfish Project. "The Barrelfish Operating System." Online, 2015. www.barrelfish.org.

[BBB+11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. "The Gem5 Simulator." *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 1–7, 2011.

[BBD+09] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. "The Multikernel: A New OS Architecture for Scalable Multicore Systems." In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, SOSP '09, pp. 29–44. Big Sky, Montana, USA, 2009.

[BC11] S. Borkar and A. A. Chien. "The Future of Microprocessors." *Communications of the ACM*, vol. 54, no. 5, 67–77, 2011.

[BCH92] J. Bruck, R. Cypher, and C.-T. Ho. "Multiple message broadcasting with generalized Fibonacci trees." In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pp. 424–431. 1992.

[BEA+08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect." In *Digest of Technical Papers of the IEEE International Solid-State Circuits Conference*, ISSCC 2008, pp. 88–598. 2008.

[BHKL06] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. "Group Formation in Large Social Networks: Membership, Growth, and Evolution." In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pp. 44–54. ACM, New York, NY, USA, 2006.

[BKM+00] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. "Graph Structure in the Web." In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Netowrking*, pp. 309–320. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2000.

[BKSL08] C. Bienia, S. Kumar, J. P. Singh, and K. Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications." In *Proceedings of the 17th International Conference*

on Parallel Architectures and Compilation Techniques, PACT '08, pp. 72–81. ACM, New York, NY, USA, 2008.

[BPS+09] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. "Your computer is already a distributed system. Why isn't your OS?" In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*. 2009.

[Bry04] R. Bryant. "Scaling linux to the extreme." In *Proceedings of the Linux Symposium*, pp. 133–148. 2004.

[BSA+15] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. "Popcorn: bridging the programmability gap in heterogeneous-ISA platforms." In *Proceedings of the 10th European Conference on Computer Systems*, p. 29. ACM, 2015.

[BWCC+08] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. "Corey: An Operating System for Many Cores." In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pp. 43–57. USENIX Association, Berkeley, CA, USA, 2008.

[BWKMZ14] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. "OpLog: A Library for Scaling Update-Heavy Data Structures." *CSAIL Technical Reports*, 2014.

[CCZ07] B. Chamberlain, D. Callahan, and H. Zima. "Parallel Programmability and the Chapel Language." *International Journal of High Performance Computing Applications*, vol. 21, no. 3, 291–312, 2007.

[CDA+14] E. H. M. Cruz, M. Diener, M. A. Z. Alves, L. L. Pilla, and P. O. A. Navaux. "Optimizing Memory Locality Using a Locality-Aware Page Table." In *Proceedings of the 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '14, pp. 198–205. IEEE Computer Society, Washington, DC, USA, 2014.

[CGS+05] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing." In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pp. 519–538. ACM, New York, NY, USA, 2005.

[CKP+93a] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. "LogP: Towards a Realistic Model of Parallel Computation." In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pp. 1–12. ACM, San Diego, California, USA, 1993.

[CKP+93b] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. "LogP: Towards a Realistic Model of Parallel Computation." In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pp. 1–12. ACM, New York, NY, USA, 1993.

[Cor13] J. Corbet. "autoNUMA." Online. `http://lwn.net/Articles/488709/`, 2013. Accessed 08/09/2016.

[CP14] V. Cabezas and M. Puschel. "Extending the roofline model: Bottleneck analysis with microarchitectural constraints." In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pp. 222–231. 2014.

[DFF+13] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems." In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pp. 381–394. ACM, New York, NY, USA, 2013.

[DGT13] T. David, R. Guerraoui, and V. Trigonakis. "Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask." In *Proceedings of the 24th ACM Symposium on Operating System Principles*, SOSP '13, pp. 33–48. ACM, New York, NY, USA, 2013.

[DGY14] T. David, R. Guerraoui, and M. Yabandeh. "Consensus Inside." In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pp. 145–156. ACM, Bordeaux, France, 2014.

[Dre11] A. Drepper. "Futexes Are Tricky." *Tech. rep.*, Red Hat, Inc., Dec 2011.

[DZ80] N. Dershowitz and S. Zaks. "Enumerations of Ordered Trees." *Discrete Mathematics*, vol. 31, no. 1, 9–28, 1980.

[FKMM15] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic. "Beyond Processor-centric Operating Systems." In *Proceedings of the 15th USENIX Conference on Hot Topics in perating Systems*, HOTOS'15, pp. 17–17. USENIX Association, Switzerland, 2015.

[FR02] H. Franke and R. Russell. "Fuss, Futexes and Furwocks: Fast Userlevel Lockingin Linux." In *Proceedings of the 2002 Ottawa Linux Symposium*, OLS '02. Denver, Colorado, 2002.

[Fre] Free Software Foundation, Inc. . "Welcome to the home of GOMP." Online. `https://gcc.gnu.org/projects/gomp/`. Accessed 01/25/2016.

[FVP06] F. Franchetti, Y. Voronenko, and M. Püschel. "FFT Program Generation for Shared Memory: SMP and Multicore." In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06. ACM, New York, NY, USA, 2006.

[Gee05] D. Geer. "Industry Trends: Chip Makers Turn to Multicore Processors." *Computer*, vol. 38, no. 5, 11–13, 2005.

[GLD+14] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quema. "Large Pages May Be Harmful on NUMA Systems." In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, 2014.

[GLF+15] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth. "Challenges of Memory Management on Modern NUMA Systems." *Communications of the ACM*, vol. 58, no. 12, 59–66, 2015.

[gli16] glibc. "pthread barriers." Online, 2016. `https://sourceware.org/git/?p=glibc.git;a=blob;f=nptl/DESIGN-barrier.txt;h=23463c6b7e77231697db3e13933b36ce295365b1;hb=HEAD`.

[GMV08] J. Giacomoni, T. Moseley, and M. Vachharajani. "FastForward for Efficient Pipeline Parallelism: A Cache-optimized Concurrent Lock-free Queue." In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pp. 43–52. ACM, Salt Lake City, UT, USA, 2008.

[GS08] R. L. Graham and G. Shipman. "MPI Support for Multicore Architectures: Optimized Shared Memory Collectives." In *Proceedings of the 15th European PVM/MPI Users' Group*

Meeting, EuroPVM/MPI ' 08, pp. 130–140. Springer Science & Business Media, Dublin, Ireland, 2008.

[Han99] S. M. Hand. "Self-paging in the Nemesis Operating System." In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pp. 73 – 86. 1999.

[HCSO12] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. "Green-Marl: A DSL for Easy and Efficient Graph Analysis." In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pp. 349–362. ACM, New York, NY, USA, 2012.

[HDV+11] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, et al. "A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling." *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, 173–183, 2011.

[HMN09] D. Hackenberg, D. Molka, and W. E. Nagel. "Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems." In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pp. 413–422. ACM, New York, New York, USA, 2009.

[Int10] Intel Corporation. "How to Benchmark Code Execution Times on Intel-32 and IA-64 Instruction Set Architectures." Online, 2010. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf`.

[Int14] Intel Corporation. "Intel Xeon Processor E5-1600/E5-2600/E5-4600 v2 Product Families, Datasheet - Volume One of Two." Online, 2014. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v2-datasheet-vol-1.pdf`, Document Number: 329187-003.

[KAH+16] S. Kaestle, R. Achermann, R. Hacki, M. Hoffmann, S. Ramos, and T. Roscoe. "Machine-Aware Atomic Broadcast Trees for Multicores." In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. Savannah, GA, USA, 2016.

[KAR+06] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico,

M. Mergen, A. Waterland, and V. Uhlig. "K42: Building a Complete Operating System." In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, Proceedings of the 1th European Conference on Computer Systems, pp. 133–145. ACM, New York, NY, USA, 2006.

[KARH15] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris. "Shoal: Smart Allocation and Replication of Memory for Parallel Programs." In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC '15, pp. 263–276. Santa Clara, CA, 2015.

[KLPM10] H. Kwak, C. Lee, H. Park, and S. Moon. "What is Twitter, a Social Network or a News Media?" In *WWW '10: Proceedings of the 19th international conference on World wide web*, pp. 591–600. ACM, New York, NY, USA, 2010.

[Knu98] D. E. Knuth. *The Art of Computer Programming*, vol. 3. Addison-Wesley, 2nd ed., 1998.

[KRSS12] A. Kaufmann, T. Roscoe, P. Shinde, and A. Schüpbach. "Low-latency OS protocol stack analysis." *Tech. Rep. 36*, Systems Group, Department of Computer Science, ETH Zurich, 2012.

[LGP04] X. Li, M. J. Garzarán, and D. Padua. "A Dynamically Tuned Sorting Library." In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, p. 111. IEEE Computer Society, Washington, DC, USA, 2004.

[LHS13] S. Li, T. Hoefler, and M. Snir. "NUMA-aware Shared-memory Collective Communication for MPI." In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pp. 85–96. ACM, New York, New York, USA, 2013.

[Lin] Linux Programmer's Manual. "futex - fast user-space locking." Online. `http://man7.org/linux/man-pages/man2/futex.2.html`. Accessed 01/24/2016.

[Mar15] Mark Bull and Fiona Reid. "EPCC OpenMP micro-benchmark suite." Online. `https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite`, 2015. Accessed 01/24/2016.

[MBB06] E. Meijer, B. Beckman, and G. Bierman. "LINQ: Reconciling Object, Relations and XML in the .NET Framework." In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pp. 706–706. ACM, New York, NY, USA, 2006.

[MDS+15] J. Malicevic, S. Dulloor, N. Sundaram, N. Satish, J. Jackson, and W. Zwaenepoel. "Exploiting NVM in Large-scale Graph Analytics." In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pp. 2:1–2:9. ACM, New York, NY, USA, 2015.

[mem16] "memcached." Online, 2016. `http://memcached.org`.

[Mes09] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 2009. Version 2.2.

[MHS12] M. M. K. Martin, M. D. Hill, and D. J. Sorin. "Why On-chip Cache Coherence is Here to Stay." *Communications of the ACM*, vol. 55, no. 7, 78–89, 2012.

[MHS14] D. Molka, D. Hackenberg, and R. Schöne. "Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer." In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pp. 4:1–4:10. ACM, Edinburgh, United Kingdom, 2014.

[MHSN15] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel. "Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture." In *Proceedings of the 44th International Conference on Parallel Processing*, ICPP ' 15, pp. 739–748. Beijing, China, 2015.

[Moc05] P. Mochel. "The sysfs filesystem." In *Linux Symposium*, p. 313. 2005.

[Moo06] G. E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp. 114 ff." *IEEE Solid-State Circuits Newsletter*, vol. 3, no. 20, 33–35, 2006.

[MVdWF08] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. "Programming the Intel 80-core network-on-a-chip Terascale Processor." In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pp. 38:1–38:11. IEEE Press, Piscataway, NJ, USA, 2008.

[NHL96] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers." *The Journal of Supercomputing*, vol. 10, no. 2, 169–189, 1996.

[NR98] R. W. Numrich and J. Reid. "Co-Array Fortran for Parallel Programming." *SIGPLAN Fortran Forum*, vol. 17, no. 2, 1–31, 1998.

[Ope08] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, 2008. Version 3.0.

[OR14] M. Odersky and T. Rompf. "Unifying Functional and Object-oriented Programming with Scala." *Commun. ACM*, vol. 57, no. 4, 76–86, 2014.

[Ora15a] Oracle Corporation. "`madvise()` in Solaris 10." Online, 2015. `http://docs.oracle.com/cd/E19253-01/817-0547/whatsnew-updates-72/index.html`.

[Ora15b] Oracle Labs. "Fortress." Online, 2015. `https://projectfortress.java.net`.

[PBMW99] L. Page, S. Brin, R. Motwani, and T. Winograd. "The PageRank Citation Ranking: Bringing Order to the Web." *Technical Report 1999-66*, Stanford InfoLab, 1999.

[PMJ+05] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. "SPIRAL: Code Generation for DSP Transforms." *Proceedings of the IEEE*, vol. 93, no. 2, 232–275, 2005.

[Pri57] R. C. Prim. "Shortest connection networks and some generalizations." *The Bell System Technical Journal*, vol. 36, no. 6, 1389–1401, 1957.

[Raz11] K. Razavi. *Performance isolation on multicore hardware*. ETH Zurich, 2011. Master's Thesis, `http://dx.doi.org/10.3929/ethz-a-006487697`.

[Red] Redis Community. "Redis." Online. `https://http://redis.io/`. Accessed 01/29/2016.

[RH13] S. Ramos and T. Hoefler. "Modeling Communication in Cache-Coherent SMP Systems: A Case-Study with Xeon Phi." In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, HPDC'13, pp. 97–108. 2013.

[RH15] S. Ramos and T. Hoefler. "Cache Line Aware Optimizations for ccNUMA Systems." In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pp. 85–88. ACM, Portland, Oregon, USA, 2015.

[RH16] S. Ramos and T. Hoefler. "Cache Line Aware Algorithm Design for Cache-Coherent Architectures." *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, 1–1, 2016.

[SCH81] P. J. Slater, E. J. Cockayne, and S. T. Hedetniemi. "Information dissemination in trees." *SIAM Journal on Computing*, vol. 10, no. 4, 692–701, 1981.

[SHV+98] V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and J. Hennessy. "Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors." In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pp. 342–355. IEEE Computer Society, Washington, DC, USA, 1998.

[Sil15a] Silicon Graphics International Corporation. "libnuma." Online, 2015. `http://oss.sgi.com/projects/libnuma/`.

[Sil15b] Silicon Graphics International Corporation. "Origin and Onyx2 Theory of Operations Manual." Online, 2015. `http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=0650&db=bks&srch=&fname=/SGI_Developer/OrOn2_Theops/sgi_html/ch02.html`.

[SLB+11] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun. "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning." In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 609–616. 2011.

[SLL10] Y.-H. Su, C.-C. Lin, and D. Lee. "Broadcasting in heterogeneous tree networks." In *Proceedings of the 16th annual international conference on Computing and combinatorics*, pp. 368–377. Springer-Verlag, 2010.

[SMDR08] A. Singh, P. Maniatis, P. Druschel, and T. Roscoe. "BFT Protocols under Fire." In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI'08)*. San Francisco, CA, USA, 2008.

[SPB+08] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. "Embracing diversity in the Barrelfish manycore operating system." In *Proceedings of the 1st Workshop on Managed Multi-Core Systems*. 2008.

[SPT09] C. A. Schaefer, V. Pankratius, and W. F. Tichy. "Atune-IL: An Instrumentation Language for Auto-Tuning Parallel Applications." In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pp. 9–20. Springer-Verlag, Berlin, Heidelberg, 2009.

[SSGA11] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. "Database Engines on Multicores, Why Parallelize when You Can Distribute?" In *Proceedings of the 6th European Conference on Computer Systems*, pp. 17–30. ACM, New York, NY, USA, 2011.

[TaMS+08] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein. "Data and Thread Affinity in OpenMP Programs." In *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?*, MAW '08, pp. 377–384. ACM, New York, NY, USA, 2008.

[TKKN16] M. Then, M. Kaufmann, A. Kemper, and T. Neumann. "Evaluation of Parallel Graph Loading Techniques." *Fourth International Workshop on Graph Data-management Experiences & Systems (GRADES 2016)*, 2016.

[UPC13] UPC Consortium. *UPC Language and Library Specifications*, November 2013. Version 1.3. Online. `http://upc.lbl.gov/publications/upc-spec-1.3.pdf`.

[WA09] D. Wentzlaff and A. Agarwal. "Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores." *SIGOPS Operating Systems Review*, vol. 43, no. 2, 76–85, 2009.

[WSP16] Q. Wang, T. Stamler, and G. Parmer. "Parallel Sections: Scaling System-Level Data-Structures." In *Proceedings of the 11th European Conference on Computer Systems*. 2016.

[WWP09] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures." *Commun. ACM*, vol. 52, no. 4, 65–76, 2009.

[XJJP01] J. Xiong, J. Johnson, R. Johnson, and D. Padua. "SPL: A Language and Compiler for DSP Algorithms." In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming*

Language Design and Implementation, PLDI '01, pp. 298–308. ACM, New York, NY, USA, 2001.

Stefan Kaestle

PhD Student - ETH Zurich - Dept. of Computer Science

Research Interests

Operating systems, distributed systems and networks, with focus on tackling scalability challenges of parallel programs on multicore machines.

Education

from 2011 **PhD student**, *Swiss Federal Institute of Technology (ETH)*, Zurich, Switzerland, working on the Barrelfish operating system.

Multicore machines are increasingly complex. In order to retain scalability of parallel applications, data has to be distributed and replicated in memory across the machine. Furthermore, concurrently running threads need to synchronize their execution and coordinate when accessing shared data consistently.

My work explores smart machine-aware memory management and synchronization so that programmers do not have to understand complexities of modern multicore hardware.

2005 – 2011 **Diploma (equivalent to master)**, *Karlsruhe Institute of Technology (KIT)*, Karlsruhe Germany.

With focus on:

- Operating systems: including μ-kernels, power management and distributed systems
- Networks: including High-Speed Networks, Wireless Sensor Networks and Next Generation Internet

Diploma Thesis: Performance Analysis of File Processing in Mainframe Environments

Awards

2011 Master's thesis: 2nd Runner Up, SHARE Academic Award for Excellence

Professional Experience

Feb. – May 2014 **Internship**, *Oracle Labs*, Cambridge, UK.

Design of a new runtime systems that uses resource usage hints given by application programmers to automatically tune thread and data placement to machine characteristics.

May '10 –Jan. '11 **Diploma thesis**, *IBM Research and Development*, Böblingen, Germany.

Performance analysis in mainframe environments by profiling operating system components using efficient dynamic code insertion. Dynamic code insertion allows to run a low overhead analysis over long time periods even under high system load.

Schaffhauserstrasse 443 – CH-8050 Zurich – Switzerland
+41 78 615 75 33 • *stefan.kaestle@inf.ethz.ch*
people.inf.ethz.ch/skaestle • *stefankaestle* • *stefan-kaestle*

September 2009 **Workshop**, *IBM Research and Development*, Böblingen, Germany.
zSummer University: IBM System z Hardware, Software components for System z.

Dec. '08 – April '09 **Research Assistant**, *KIT - Insitute of Telematics*, Karlsruhe, Germany.
Wireless Sensor Networks: Designed a representation of the CIM network management protocol in Java and a mapping between both models; Applied for managing test-beds for Wireless Sensor Networks.

Dec. '07 – Oct. '08 **Research Assistant**, *KIT - Insitute of Telematics*, Karlsruhe, Germany.
Collaborative Research Center 588 - Humanoid Robots: Design and implementation of a program for automatic mapping of 2D images on 3D meshes.

Publications

Stefan Kaestle, Reto Achermann, Roni Hacki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. Machine-aware atomic broadcast trees for multicores. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, Savannah, GA, USA, 2016.

Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. Shoal: Smart Allocation and Replication of Memory for Parallel Programs. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC '15, Santa Clara, CA, 2015. USENIX Association.

Tim Harris and Stefan Kaestle. Callisto-RTS: Fine-grain parallel loops. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC '15, Santa Clara, CA, 2015. USENIX Association.

Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. We need to talk about NICs. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, Berkeley, CA, USA, 2013. USENIX Association.

Research Activity

Stefan Kaestle, Timothy Roscoe
Distributed Computing on Hybrid Multicore Machines
Doctoral workshop, 8th European Conference on Computer Systems, 2013

Teaching

2016 MATLAB introduction
2015 Parallel Programming
2011 – 2014 Systems Programming and Computer Architecture
2012 – 2015 Informatics for Biology, Pharmacy and HST students
2012 Informatics for Civil Engineers

Skills

Programming C/C++, Haskell, Linux shell scripting, Python, Assembler (x86/ARM), Java
Languages German: Native Proficiency, English: Professional Working Proficiency

Schaffhauserstrasse 443 – CH-8050 Zurich – Switzerland
+41 78 615 75 33 • *stefan.kaestle@inf.ethz.ch*
people.inf.ethz.ch/skaestle • *stefankaestle* • *stefan-kaestle*

Index