UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1139

Scientific Computing on Multicore Architectures

MARTIN TILLENIUS

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2014

ISSN 1651-6214
ISBN 978-91-554-8928-1
urn:nbn:se:uu:diva-221241

Dissertation presented at Uppsala University to be publicly examined in Room 2446, Polacksbacken, Lägerhyddsvägen 2, Uppsala, Friday, 23 May 2014 at 10:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Doctor Jakub Kurzak (University of Tennessee, Innovative Computing Laboratory).

Abstract
Tillenius, M. 2014. Scientific Computing on Multicore Architectures. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1139. 47 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-554-8928-1.

Computer simulations are an indispensable tool for scientists to gain new insights about nature. Simulations of natural phenomena are usually large, and limited by the available computer resources. By using the computer resources more efficiently, larger and more detailed simulations can be performed, and more information can be extracted to help advance human knowledge.

The topic of this thesis is how to make best use of modern computers for scientific computations. The challenge here is the high level of parallelism that is required to fully utilize the multicore processors in these systems.

Starting from the basics, the primitives for synchronizing between threads are investigated. Hardware transactional memory is a new construct for this, which is evaluated for a new use of importance for scientific software: atomic updates of floating point values. The evaluation includes experiments on real hardware and comparisons against standard methods.

Higher level programming models for shared memory parallelism are then considered. The state of the art for efficient use of multicore systems is dynamically scheduled task-based systems, where tasks can depend on data. In such systems, the software is divided up into many small tasks that are scheduled asynchronously according to their data dependencies. This enables a high level of parallelism, and avoids global barriers.

A new system for managing task dependencies is developed in this thesis, based on data versioning. The system is implemented as a reusable software library, and shown to be as efficient or more efficient than other shared-memory task-based systems in experimental comparisons.

The developed runtime system is then extended to distributed memory machines, and used for implementing a parallel version of a software for global climate simulations. By running the optimized and parallelized version on eight servers, an equally sized problem can be solved over 100 times faster than in the original sequential version. The parallel version also allowed significantly larger problems to be solved, previously unreachable due to memory constraints.

Keywords: multicore, scientific computing, shared memory parallelism, task-based programming, parallel programming model, task scheduling, data versioning

Martin Tillenius, Department of Information Technology, Division of Scientific Computing, Box 337, Uppsala University, SE-751 05 Uppsala, Sweden. Department of Information Technology, Computational Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-554-8928-1
urn:nbn:se:uu:diva-221241 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-221241)

List of papers

This thesis is based on the following papers, which are referred to in the text by their roman numerals, and on the software developed as part of this work.

I K. Ljungkvist, M. Tillenius, D. Black-Schaffer, S. Holmgren, M. Karlsson, and E. Larsson. Using hardware transactional memory for high-performance computing. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*, May 2011, pp. 1660–1667.[1]

Contributions: The author of this thesis and K. Ljungkvist implemented and performed all experiments in close collaboration. The manuscript was written by the first two authors in collaboration with the other authors.

II M. Tillenius. SuperGlue: a shared memory framework using data versioning for dependency-aware task-based parallelization. Technical Report 2014-010, Department of Information Technology, Uppsala University, March 2014.

Contributions: The author of this thesis is the sole author of this paper.

III M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell. Resource-aware task scheduling. Technical Report 2014-001, Department of Information Technology, Uppsala University, January 2014. Submitted for publication.

Contributions: The author of this thesis did the implementation, performed the experiments, and contributed to the analysis. Most of the manuscript was written in close collaboration with E. Larsson.

IV M. Tillenius, E. Larsson, E. Lehto, and N. Flyer. A task parallel scattered node finite difference scheme for the shallow water equations on a sphere. Technical Report 2014-011, Department of Information Technology, Uppsala University, March 2014.

Contributions: The author of this thesis developed the parallel version of the software, performed the experiments, and contributed to the manuscript.

---

[1]

V A. Zafari, M. Tillenius, and E. Larsson. Programming models based on data versioning for dependency-aware task-based parallelisation. In *2012 IEEE 15th International Conference on Computational Science and Engineering (CSE)*, December 2012, pp. 275–280.[2]

Contributions: The author of this thesis contributed to the software design, developed the shared memory version of the software, and contributed to the manuscript.

Reprints were made with permission from the publishers.

Software

A software library for data-dependent task parallelism was developed as part of this work. It is available at http://tillenius.github.io/superglue/.

[2]Copyright © 2012 IEEE. Reprinted with permission.

Related work

Although not explicitly discussed in the comprehensive summary, the following pappers are related to the contents of this thesis.

- M. Tillenius and E. Larsson. An efficient task-based approach for solving the n-body problem on multicore architectures. In *PARA 2010: State of the Art in Scientific and Parallel Computing*, University of Iceland, Reykjavík, 2010, 4 pp.
- K. Ljungkvist, M. Tillenius, S. Holmgren, M. Karlsson, and E. Larsson. Early results using hardware transactional memory for high-performance computing applications. In *Proceedings of the 3rd Swedish Workshop on Multi-Core Computing*, Chalmers University of Technology, 2010, pp. 93–97.
- M. Holm, M. Tillenius, and D. Black-Schaffer. A simple model for tuning tasks. In *Proceedings of the 4th Swedish Workshop on Multi-Core Computing*, Linköping University, 2011, pp. 45–49.
- M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell. Resource-aware task scheduling. In *4th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures (PARMA)*, Technical University of Berlin, 2013, 6 pp.
- M. Tillenius, E. Larsson, E. Lehto, and N. Flyer. A task parallel implementation of a scattered node stencil-based solver for the shallow water equations. In *Proceedings of the 3rd 6th Swedish Workshop on Multi-Core Computing*, Halmstad University, 2013, pp. 33–36.

Contents

1. Introduction

Modern processor architectures have several computational cores, and are capable of executing several instructions in parallel. To take full advantage of such hardware, the work that the processor performs needs to be divided and distributed among the cores. That is, software needs to be parallel. This raises new challenges, as parallel software is more complex than sequential software and requires more of the developer.

In addition to the difficulties of writing sequential software, the programmer must also solve the problems of how to partition the work among the cores, and how to manage interactions between work that is performed in parallel. Errors due to mistakes in the management of these interactions may be triggered only when the cores interact in a certain order, may appear and disappear with unrelated changes, and may show only spuriously. This makes these kind of mistakes hard to reproduce and track down.

Aside from correctness errors, that cause the software to do the wrong thing, parallel software also opens up for new kinds of mistakes that cause performance penalties. If a core has to wait for another core to finish some computations before it can continue with its own, performance is wasted. Even if all cores have exactly the same amount of work, some might still finish faster than others because of operating system interference and other effects. How fast a certain piece of work can be executed will depend on what the other cores are doing, since parts of the processor are shared between the cores. This makes execution times hard to predict precisely, which in turn makes it hard to ensure that all cores finish their part of the work at the same time.

Other types of performance problems consist of artificial dependencies, such as introducing unnecessary synchronization points or fixing the order in which computations must be performed, when it is not needed. Such mistakes lead to cores being idle waiting for other cores to finish instead of contributing to the computations.

The goal in this thesis is to make programming for multicore architectures easier, so that programmer efficiency is increased and the risk of both correctness and performance errors is minimized. More specifically, the goal is to find patterns in efficient multicore software that can be extracted into reusable building blocks and constructs, and raise the level of abstraction the programmer works at. For such abstractions to be useful, they must be carefully designed not to impose limitations, reduce the expressiveness or generality, or introduce performance penalties.

An application driven approach is taken to find these abstractions. The applications we are interested in are software for scientific computing software.

The search for suitable abstractions is therefore driven by implementing prototypes of applications from the area of scientific computing. Distinctive for scientific computing is that problem sizes are usually large, commonly limited by computational requirements, time, and memory. This puts demands on memory management, and makes high performance and scalability perhaps not a more important requirement than in any other type of software, but at least a more obvious one.

The state-of-the-art in programming for multicore processors is to divide software into fine-grained tasks with dependencies between them, that are scheduled dynamically by a runtime system. The dependencies are deduced by the runtime system from information about how each task access data, which the programmer supplies. The approach taken in this work is to implement a library to provide such a model as a test bed for research, and ensure that it fits the needs and requirements for scientific computing applications.

The resulting task library is verified to be useful, and to suite the needs of scientific software by using it for the development of an example application; a global climate simulation software. The model is also extended to support distributed memory systems, to be able to manage realistic problem sizes.

The outline of this thesis is as follows. The next chapter provides background on multicore architectures, parallel programming models, and introduces the necessary terminology. The thesis then follows a bottom-up approach, starting from the synchronization primitives everything else must be based upon in chapter 3. Higher level abstractions are then studied in 4, where the developed library for task-based programming is presented. Finally, chapter 5 describes the implementation of a software for global climate simulation that builds on our library for parallelism.

Contributions

The contributions of this thesis are:

- Evaluation of hardware transactional memory for atomic updates of floating-point data.
- The design and implementation of a simple, flexible, and efficient system to manage dependencies between tasks.
- An experimental comparisons between several task execution runtime systems.
- The design and implementation of a construct to avoid exhaustion of system resources in task execution runtime systems.
- A hybrid shared-/distributed memory application for global climate simulations, built upon the solution presented here.

2. Background

2.1 Multicore processors

Physical limits make it infeasible for hardware manufacturers to increase the processor clock frequency beyond where it is today, and has been for the last decade. Higher computational power is instead provided by putting several cores that run at a lower clock frequency on a single chip. The current trend is that technology advances are used to put more cores on a single chip. Hence, the best prediction we can make is that multicore processors will not disappear in the nearest future, and that the number of cores per processor will increase in future processor architectures.

Distinctive for multicore processors is that parts of the processor is shared between the cores. The most significant shared part is the memory system. An advantage of this is that cores on a multicore processor can communicate through the shared memory, and possibly even through a shared cache. This can be significantly faster than communicating between memory address spaces using message passing, which is the alternative for parallel software in distributed memory settings. However, sharing parts of the processor also means that the cores might have to compete for shared resources.

From the software perspective, it is not always relevant to talk about cores. Processors often have hardware multithreading, in which case a single core is presented as several logical processors. How fast two logical processors can communicate depends on how much they share between them. On a processor with hardware multithreading, two logical processors might be two hardware threads on the same core, sharing not only caches but also functional units. In contrast, on a computer with several processors, two logical processors might be on different processor chips and only share the memory system. Hence, how fast two logical processors can communicate, and how much they might interfere, may vary vastly. This must be taken into account to use the system efficiently.

2.2 Scope and terminology

It will be assumed that all software run under an operating system which is POSIX compliant or similar (this includes UNIX, Linux, MacOSX, and Windows). The operating system provides means to create processes and threads that will be scheduled by the operating system to run on the processor cores.

For the scope of this theses, all shared memory applications consist of a single process, and use threads to execute work in parallel. A large part of this work is about tasks, which here is always a user-space construct. The operating system has no notion of these tasks.

3. Synchronization primitives

In a multi-threaded program, threads might observe changes to memory in a different order than they were carried out, and in a different order than other threads observe them. To be able to synchronize between threads by reading and writing shared memory, it must be possible to rely on the order in which changes are visible. The order can be enforced by the insertion of memory barriers, that restrict how reads and writes can pass the barrier.

In addition to memory barriers, there are instructions to perform atomic operations on integers, such as atomic arithmetic or bitwise operations, and more powerful operations such as compare-and-swap, which allow a memory location to be changed only under the condition that it contains a certain value. All atomic instructions also imply some kind of memory barrier.

These primitives are normally used to build higher level abstractions such as locks, condition variables, and barriers. For most situations these higher level constructs are sufficient, and the programmer does not need to think about details like memory barriers. In some performance critical sections, however, it can pay off to implement synchronization directly in terms of atomic instructions and memory barriers, for instance to implement a customized lock, or a concurrent data structure.

Only just recently, a new primitive for inter-thread synchronization has been introduced in commercial processors; hardware transactional memory. Transactional memory was first suggested by Herlihy and Moss in [26], but was not implemented until Sun (later Oracle) built the Rock processor [13]. The Rock processor was canceled before it was commercially available, but hardware transactional memory is now available both in IBM's BlueGene/Q processor [25], and in Intel processors based on the Haswell architecture [28] with the Transactional Synchronization Extensions.

3.1 Hardware transactional memory

The transaction concept means that all operations within a transaction are either all visible at once, or not visible at all. This is a way to ensure that only one thread at a time can update a shared state.

The standard way of synchronizing accesses to shared state is to use locks. Unfortunately, locks have several disadvantages. A lock occupies a memory location, to store whether it is taken or not. This consumes memory and requires additional memory accesses, which may degrade performance in memory bound software. Another important problem is deadlocks. If two thread

need two different locks, but acquire them in the opposite order so that they hold one lock each and then wait for the second one to be released, they will deadlock and none of them will be able to progress. Problems also arise when a thread that holds a lock is preempted by the operating system. Other threads that wait for this lock will not be able to continue until the thread is rescheduled and releases the lock, a problem known as lock convoying. If a thread that needs the lock has higher priority than the preempted thread, the higher priority thread will have to wait for the lower priority thread to finish, a problem known as priority inversion.

Transactional memory is a way to avoid all these problems. Transactions require no additional memory, cannot deadlock, and cannot prevent other threads from progress when preempted.

An interesting property about transactions is that they are optimistic. This means that it is expected that the transaction will succeed, and then overhead can be made very small. The cost instead comes if there is a collision, and the transaction must be retried. This is in contrast to programming with locks. Locks are pessimistic in the sense that a lock is always acquired, which comes at a certain cost, regardless of it will be needed or not. Once the lock is acquired, it is guaranteed that there will be no collisions.

In Paper I we investigate if hardware transactional memory can be of use for accelerating scientific computing software. Specifically, we evaluate how efficient it is to use hardware transactional memory for updating floating point data atomically. We perform experiments on a processor with hardware transactional memory; a prototype of the Rock processor, which is described below.

3.2 The Rock processor

The Rock is a 16-core SPARC V9 processor, that implements hardware transactional memory. The support for hardware transactional memory adds two new instructions:

- `chkpt [fail_pc]`
- `commit`

The `chkpt` instruction marks the beginning of a new transaction. It takes an argument `fail_pc`, which is the address to jump to if the transaction is aborted.

The `commit` instruction marks the end of a transaction, and commits it. If a collision is detected, the program counter will be set to `fail_pc` as set by the `chkpt` instruction. Otherwise, the program will continue with the next instruction.

Mask	Description	Mask	Description
0x001	Invalid	0x040	Size
0x002	Conflict	0x080	Load
0x004	Trap Instruction	0x100	Store
0x008	Unsupported Instruction	0x200	Mispredicted Branch
0x010	Precise Exception	0x400	Floating Point
0x020	Asynchronous Interrupt	0x800	Unresolved Control Transfer

Table 3.1. *Meaning of bits in the* `%cps` *register explaining why a transaction was aborted. This information is based on a table in [15].*

3.2.1 Aborted transactions

Transactions are not guaranteed to succeed, but can fail for various reasons. If a transaction fails, a new status register `%cps` contains information about the reason. Table 3.1 shows the meaning of the bits in this register. The names in this table are not the official ones, but our interpretation of the information in [15]. The most important reasons for a transaction to fail are the following:

Conflict indicates that some other thread modified the same address during our transaction.
Size means the transaction was too large.
Load means a value was not ready to be read from memory.
Store indicates that a value could not be written to memory.

The size of a transaction is limited by the store queue. In our experiments, transactions of up to 8 double precision values (64 bytes) were successful, but larger transactions often failed with the **Size** bit set. It is worth noting that the same transaction, accessing the same amount of memory, may sometimes fail due to size, and sometimes succeed.

If the **Load** or **Store** bit is set, it means that the data is not available in the first level cache, or in the case of **Store**, that it is not exclusive in the cache coherency protocol.

When transactions fail, we use an exponential back-off scheme to wait for a short time before the transaction is retried. We also take some action depending on why the transaction failed, before retrying again. If the transaction failed with the **Store** bit set, this means it was not present, or not exclusive, in the first level cache. To force the memory system to load the data into our cache with write permission, we need to write to the location from outside a transaction. But an unsynchronized write would corrupt the data, so instead we use a trick from [16], and perform a dummy compare-and-swap operation to the location. The transaction is then retried.

If the transaction failed with the **Load** bit set, we perform a read on the data from outside a transaction, and then retry.

If the transaction failed due to a **Conflict**, we wait for a short time using the back-off scheme, and then retry.

Using this strategy, we can get all transactions to eventually succeed. A common approach is to fall back to other methods when a transaction fails, but that was not required in this case.

3.3 Experiments

We evaluated four different methods for atomic updates of floating point values: *nothing*, *locks*, *CAS*, and *transactions*. The *nothing* method performs no synchronization at all, and will generate incorrect results. It is included to be a base line. The *locks* method uses a Pthreads lock. Each lock is put in its own cache line to avoid false sharing. The *CAS* operation uses the atomic compare-and-swap instruction. This instruction is only available for integer values. To use it for floating point updates, we have to convert the value back and forth between being an integer and a floating point value. The *transactions* method uses the new hardware support for transactional memory to perform the updates atomically.

First, we conduct two micro-benchmark experiments to measure synchronization overhead and sensitivity to contention. We then look into two more realistic work-loads from scientific computing software; assembling a matrix in a finite element method code, and updating particles in a direct n-body simulation code.

3.3.1 Synchronization overhead

The first experiment is aimed at measuring the overhead of different methods for synchronizing the updates.

The experiment sums floating point values to a fixed memory location in a tight loop, and counts the number of tried updates and the number of successful updates. To make comparisons fair, both tried and successful updates are counted also for methods where updates cannot fail. This experiment is run on a single thread.

The times we are trying to measure here are very small, and care must be taken to measure the right thing. We were not able to get the C compiler to produce comparable code for all methods. Instead, we wrote this experiment directly in assembly language, which improved the performance, and gave more comparable binaries.

The number of floating point values to update in each critical section was varied between 1 and 8 double precision values. A disadvantage of the *CAS* method is that it cannot update several elements simultaneously. When *CAS* is included, it updates one value at a time.

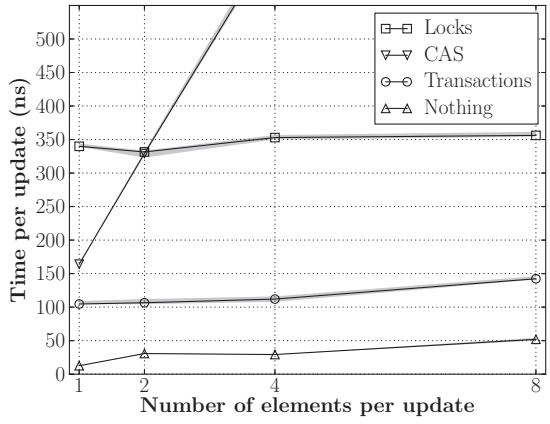


Figure 3.1. Synchronization overhead. The area between the first and third quartile of 100 runs is marked, but barely visible due to the low dispersion.

The results from this experiment are shown in Figure 3.1. The *CAS* method stands out, as its overhead scales linearly with the number of updated elements, while the other methods are almost unaffected. The *transactional memory* method was almost three times faster than Pthread locks, which is promising for its applicability to scientific applications.

3.3.2 Contention sensitivity

The overhead experiment only considered a single thread. This experiment aims to measure the behavior of the methods under contention from several threads. One thread per core runs a tight loop where it either increases a local variable, or a single shared global variable. By variating how often threads write to their own variable and how often they write to the shared variable, different levels of contention are simulated.

This experiment uses the methods described in section 3.2.1 to cope with failed transactions, and both the *CAS* and the *transactions* methods use an exponential back-off scheme.

Figure 3.2 show the results from the experiment. The behavior was surprising to us. We had expected the transactional memory method to have low overhead at low contention, when most transactions should succeed without collisions, and high overhead costs at high contention, when many transactions will fail due to collisions and will have to be retried. Instead it turned out that transactions was slower than the *CAS* method at low contention levels, even though we have to load the value both into the integer and floating point units to make compare-and-swap work with floating point data. It was also a surprise that at high contention levels, the *transactions* method levels out, and becomes less affected by increased traffic, while the other methods keep slow-

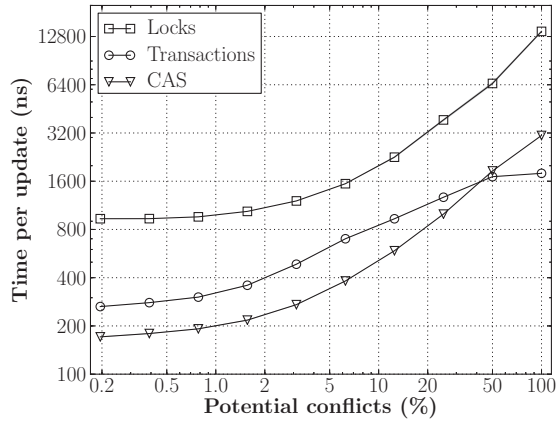


Figure 3.2. Behavior under contention. The area between the first and third quartile of 100 runs is marked, but not visible due to the low dispersion.

ing down. We performed additional tests with different back-off parameters, but we were not able to make *CAS* reproduce the behavior of the *transactions* method that way.

3.3.3 Matrix assembly

The micro-benchmarks both yielded interesting and partly unexpected results, but were designed to stress the hardware to investigate its behavior. To evaluate more realistic workloads, this experiment has several threads that performs scattered writes to a large matrix, which is shared among all the threads. This mimics the assembly of matrices in a finite element method, where an unstructured mesh is traversed, and values corresponding to the nodes of each triangle element are updated.

We simulate two different amounts of calculations on each triangle element. One where very few computations are performed, and one that simulates a heavy computation which requires 1000 times more computations.

Figure 3.3 shows the performance of the different methods. In the case with few computations, none of the methods scaled well. This is because of the high memory traffic. Here, *locks* ended up in the last place, possibly because of the additional memory footprint. Even with heavy computations between memory accesses, none of the methods scaled well, but hardware transactional memory outperformed the alternatives.

3.3.4 Direct n-body simulation

In the last experiment, we consider an application that has a more regular access pattern, and accesses a smaller amount of memory. We selected an appli-

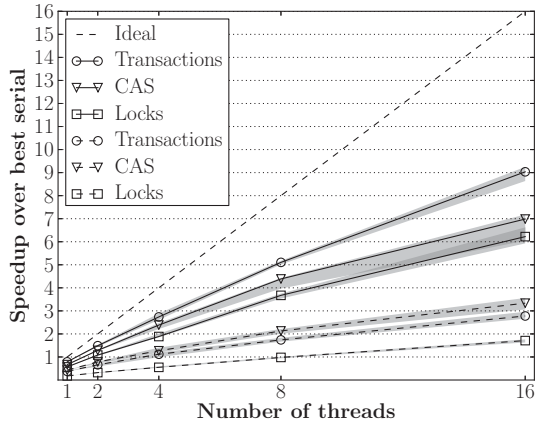


Figure 3.3. Matrix assembly application. Solid lines show results for the compute-heavy version, and the dashed lines represent the version with few calculations. The area between the first and third quartile of 100 runs is marked.

cation that simulates a set of particles that attract or repel each other through pairwise interactions. For $O(n)$ particles, there are $O(n^2)$ interactions calculated, making the application computationally intensive. The application uses a time stepping scheme, where each time step consists of first calculating all forces, and then moving the particles accordingly.

In this experiment, we could benefit from grouping several particles together and handle them as a group. This way, the *locks* and *transactions* methods can amortize the time for acquiring exclusive access, while the *CAS* method will not benefit at all.

We also implemented an alternative version where each thread wrote its result into its own buffer, and then these buffers were merged together in an additional step. This solution requires more memory, and has extra computations to merge the buffers, why it is interesting to see if transactional memory could be fast enough to make such solutions unnecessary. That would make the development easier, reduce the memory footprint, and avoid unnecessary computations.

The results of the particle simulation is shown in Figure 3.4. The *CAS* method did not scale as well as the others, as expected, since it cannot update more than one particle in a single atomic update. The methods using locks and transactions behaved similarly, but none of them came close to the alternative implementation where each thread worked in its own buffer.

3.3.5 Discussion

Hardware transactional memory is an interesting addition to the processor's instruction set. Even though the Rock processor is now discontinued, sev-

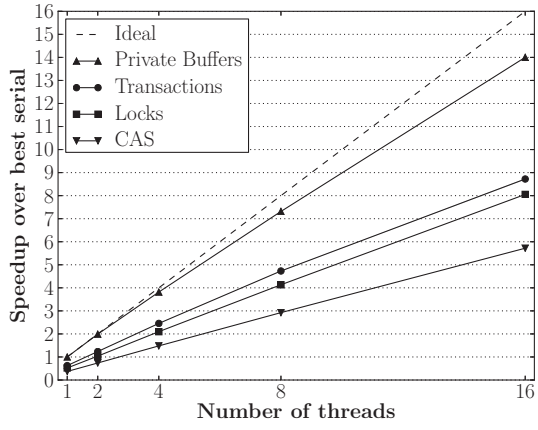


Figure 3.4. Particle simulation. The area between the first and third quartile of 100 runs is shaded.

eral other hardware manufacturers are now selling processors with hardware transactional memory, and it is becoming mainstream. The implementations differ, and it is difficult to say if some of the results here translate to newer hardware. Though the improvements in performance have been modest, and the construct is perhaps best suited for implementing lock-free data structures, we have shown that it is possible to use it to atomically update floating point values in a practical way that outperforms alternative methods. We have also shown that it is possible to rely completely on hardware, without having to fall back to other means of ensuring atomicity.

4. High-level programming models for multicores

A programming model is here the abstractions in which the programmer thinks about the software, and which the programmer uses to express the software during implementation. There are many different programming models, and this section discusses a small number of selected models that are deemed to be especially suitable for shared-memory parallelism, or related to the work presented in this thesis.

The most basic programming model is what the operating system provides, which for threads is functions to create a thread, exit from a thread, and wait for another thread to finish. All other models must build upon these. Operating systems also provide constructs for mutual exclusion and synchronization such as locks, condition variables, and barriers, but most of these could also be implemented in user-space.

The threading model provided by the operating system can be used directly, but is commonly considered to be difficult to use, and has been called the "assembly language of threading". More commonly, higher level abstractions are build upon these primitives.

4.1 Task-based programming models

The most common way to program multicore and shared memory computers is OpenMP [14]. OpenMP was initially primarily targeting data parallelism, where the same operation is performed on multiple data in parallel. In order to extract as much parallelism as possible, it is a limitation to demand that what is performed in parallel is the same operation. To allow different operations to be performed in parallel to a greater extent, the concept of tasks was introduced into OpenMP 3.0 [4]. A task is a snippet of code and the data needed for the code to execute. Tasks are user-space objects, and not known by the operating system. In the OpenMP model, a section of code can be annotated to be a task, and when the execution reaches such a section, a task is created but not necessarily executed directly, and the program continues. The task will not create a new thread, instead there are worker threads that will pick up and execute the created tasks. Tasks can be executed as soon as they are created, but they can also be deferred and executed later. To make sure that a task has finished, there are constructs to wait for all previously created tasks to finish.

This synchronization between tasks is rather coarse grained, and often introduces artificial synchronization points which limits the parallelism. A solution to this is to introduce dependencies between tasks. The Star Superscalar programming model from Barcelona Supercomputing Center supports the concept of data dependencies between tasks. Their model was first built for the Cell/BE processor, under the name CellSs [5], and was then ported to general shared memory machines with SMPSs [34]. This has now been superseded by OmpSs [18], which is an effort to collect the Star Superscalar programming models as extensions to OpenMP directives.

In the Star Superscalar model, a section of code can be declared to be a task by using a pragma statement. For each task, the programmer can specify what data the task accesses, and whether the data is read or written. The programmer then writes a sequential program that creates tasks to perform the actual work, and a runtime system deduces dependencies from the data access information, and schedules the tasks for execution, while respecting these dependencies.

This model was suggested to be included in OpenMP in [19], and was incorporated into the standard with OpenMP 4.0 [33]. In addition to the original OmpSs implementation, support for the new dependency driven tasks has also been implemented in the OpenUH [23] compiler.

The success of this model is evident from its use in linear algebra packages, where performance and efficiency is key. How to make efficient use of multicore processors for linear algebra algorithms was studied in [30]. There, it was observed that it is not enough that the underlying basic linear algebra kernels are parallel, since this leads to a fork-join model with a lot of unnecessary synchronization. The authors instead propose a task-based model where the algorithm is formulated as a task graph, where nodes are tasks and edges are dependencies. Tasks are then scheduled dynamically as soon as all their predecessors are finished.

The impact of multicore processors on mathematical software was also analyzed in [11], where it is pointed out that multicore architectures will require a much finer granularity of parallelism than earlier. This research was carried out in the initial phase of the PLASMA project (Parallel Linear Algebra for Scalable Multi-Core Architectures) [17], a dense algebra package targeting multicore architectures. PLASMA now uses a task-scheduling runtime called Quark [42, 41]. Quark is a C library that provides a programming model similar to the StarSs model. Tasks and their arguments are submitted to the runtime system, together with data direction hints, and the runtime infers dependencies and schedules the tasks dynamically.

FLAME [24] is another linear algebra package that also takes the data-driven task-based approach. It uses a runtime system called SuperMatrix [12]. This system takes a slightly different view. Instead of building a task dependency graph, dependencies are detected using an algorithm similar to the Tomasulo algorithm [38] for out-of-order executing of instructions in processors.

For general purpose programming, StarPU [3] is a C library for data-driven task-based programming that targets architectures with both CPUs and GPUs. StarPU has the capability of building models of the expected execution and data movement times, and selects to run tasks where it is most beneficial. StarPU will then also handle the data movements for the user.

Another library for general purpose task-based programming is XKaapi [22], which is a successor to Athapascan-1 [21]. It is a library solution, like StarPU and Quark, but has both a C and a C++ interface. It also supports architectures with multiple CPUs and GPUs. In XKaapi, dependencies are only calculated when tasks are stolen. Otherwise, tasks are executed in order, and no dependency checking is needed.

Two popular task-based systems that do not support task dependencies are Intel TBB [29], and Cilk [6] (now Cilk Plus [27]). Cilk is a language extension to C and C++ that adds keywords to spawn and wait for tasks. Cilk popularized task-stealing schedulers, but only provides a fork/join model for parallelism. A project called Swan [40] extends Cilk with data-driven dependencies. Tasks in Swan can have dependencies on objects, rather than on other tasks. Dependencies in Swan are tracked using a system with tickets, similar to ticket-based locks. This is similar to the data versioning method we have developed, described in the next section.

4.2 SuperGlue: a data-driven task library

Paper II presents SuperGlue: our software library for task-based parallelism. The rational for developing a new task library rather than building on an existing is that we wanted a test bed for experimenting with different designs and strategies, and to be able to customize it after our needs.

Although initially intended for experiments, SuperGlue is carefully designed to be of general use. Implemented as a C++ header-only template library, it requires no separate compilation in order to include in existing C++ projects. There is a C interface for interoperability with projects written in other languages, which was for instance used in a Fortran project in Paper V. There are no external dependencies, and it can run on top of either Pthreads or OpenMP, which makes it easy to incorporate in other projects.

Since SuperGlue is designed for experimentation, it is highly customizable and possible to tweak to fit the programmer's needs. It also strictly adheres to the "What you don't use, you don't pay for" design rule, borrowed from the design of C++ [35], so customization is entirely at compile-time, and has no runtime cost.

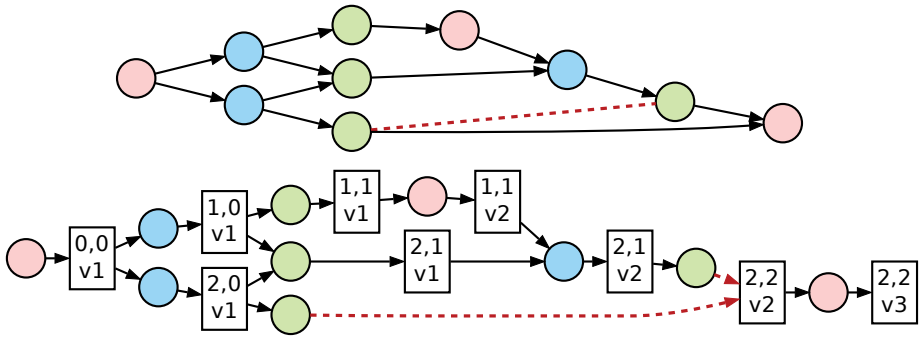


Figure 4.1. Task dependency graphs. Circles are tasks and edges are dependencies. The boxes in the bottom graph illustrates data objects. The first row in each box is the name of the object, and the second row is a version number. Top figure: Dependencies between tasks. Bottom figure: Dependencies on data. The dashed edge in the top figure means that the tasks must not run concurrently. This corresponds to the two dashed edges in the bottom figure, which mean that exclusive access to the data is required.

4.2.1 Programming model

The two main concepts in SuperGlue are *handles* and *tasks*. Tasks are represented by objects that inherit from a provided base class, and implement a `run` method which is called to execute the task. Handles are abstract objects that represent something to which accesses should be managed. The model that SuperGlue provides is that the programmer creates tasks, and submits them to a runtime system. For each task, the programmer declares which handles the task requires, and how each handle is accessed (e.g. read or write). The runtime then deduces dependencies, and schedules the tasks onto the available cores, respecting the dependencies.

Handles do not contain any data, but can represent data in any form. By decoupling the handle objects from the actual data, the interface is less intrusive than requiring that all managed data must be packaged in some class that SuperGlue provides. It also means that any kind of data structure is supported, since SuperGlue is oblivious of what structure the programmer actually protects with the handle.

Another advantage of decoupling the handles from the protected data is that it is clear that SuperGlue can not detect whether handles overlap. If alias detection is desired, this can be added as a layer on top of SuperGlue.

4.2.2 Dependency management through data versioning

One of the major differences between SuperGlue and other solutions is the way dependencies are managed.

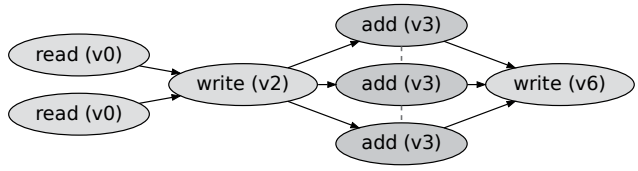


Figure 4.2. Illustration of the idea of data versions. The nodes represent tasks that all access the same handle. The label specifies the access type, and the number in the parenthesis specifies which version of the handle the task requires.

The most common approach to represent dependencies is to build a graph of tasks, where a task knows it successors and notifies them when it is finished. In our system, tasks have dependencies to handles instead of to other tasks, and have no knowledge of other tasks at all. Not knowing about other tasks is a good property, since tasks may be created, executed, and deleted at any point in time. Tasks synchronize with the handles instead of with other tasks. Handles are already naturally required to be available during the lifetime of the task, since they represent the data that the task uses.

Dependencies between tasks are also somewhat artificial; a dependency between two tasks means that the first task use or create something that the other task needs. By creating an object for this "something" and having both tasks depend on this object instead, dependencies are represented more correctly. This allows a task to depend on a result from a future task that has not been submitted yet, something that cannot be formulated in a natural way using dependencies between tasks.

Figure 4.1 illustrates a task dependency graph as dependencies are usually represented (top), and how SuperGlue views dependencies as between tasks and data (bottom). Notice that there are no edges between tasks in the SuperGlue view.

To manage dependencies, each handle has a version number, and a task depend on specific versions of the handles it accesses. Which version a handle should require is deduced when the task registers all handles it accesses. The version number is increased after each access to the handle, including read-only accesses, so that the system can know that all reads are finished before the data is overwritten.

The version number is simply a counter keeping track of how many times the handle has been accessed. That is, the number of tasks that access the handle that have been executed and finished. Each task knows how many times each handle must have been accessed before it is its turn to access it.

The idea is illustrated in Figure 4.2. Here, seven tasks all access the same data. The first two read from the data, and can run at the same time. They will require the same version of the handle, which is version 0. The next task writes to the data, and will have to wait for the two reads to finish. It will require version 2 of the data. After this, three tasks sum their results to the data. It

does not matter in which order they execute, but they need exclusive access to the data to avoid data races. They all require version 3, meaning they have to wait for three earlier tasks to finish before they can execute. These tasks will race to execute first, and acquire a lock the handle to ensure exclusive access. Finally, a task writes to the data. It requires version 6, since all six previous tasks need to finish before it can execute.

Figuring out which version a task should require is simple. Each handle counts the number of tasks that has been scheduled to access it, and what the last access type was. The main idea is then that subsequent reads will require the same version, and writes will have to wait for all previous accesses to finish, that is, the required version is the total number of scheduled accesses.

4.2.3 Scheduling

Scheduling in SuperGlue is driven by locality. When a task is submitted to SuperGlue, its dependencies are checked. Tasks whose dependencies are all fulfilled at submission are distributed among the workers in a round-robin fashion. If one of the dependencies is not yet fulfilled, i.e. if the required version of a handle is not yet available, the task is queued at that handle.

When a worker thread finishes the execution of a task, it will increase the versions of all accessed handles, and wake tasks that are waiting for the new version of these handles. The woken tasks that are ready to run will be put in the worker's queue of ready tasks. This means that the tasks will be run on the thread where the data they need was produced, unless load balancing is needed.

Load balancing is handled through work-stealing. If a worker thread runs out of tasks to execute, it will pick one of the other worker threads at random, and try to steal a task from that thread.

Scheduling is entirely distributed. There is no global information that needs to be accessed to create handles or to create or execute tasks. This is an important property, since global data structures easily become bottlenecks.

When a worker thread is to decide which task to run next, the strategy is to prioritize low scheduling overhead over an optimal schedule. The worker simply takes the first task from its ready queue. There is no information available to be used for planning ahead, since tasks can be created at any time, and by any thread, and the length of the tasks is not known.

Which task to pick from the ready queue, and which workers to try to steal from can be customized. It is also possible to enable prioritized tasks, which allows the programmer to e.g. give priority to the critical path. SuperGlue then keeps two separate task queues for each worker thread, and picks tasks from the prioritized queue first.

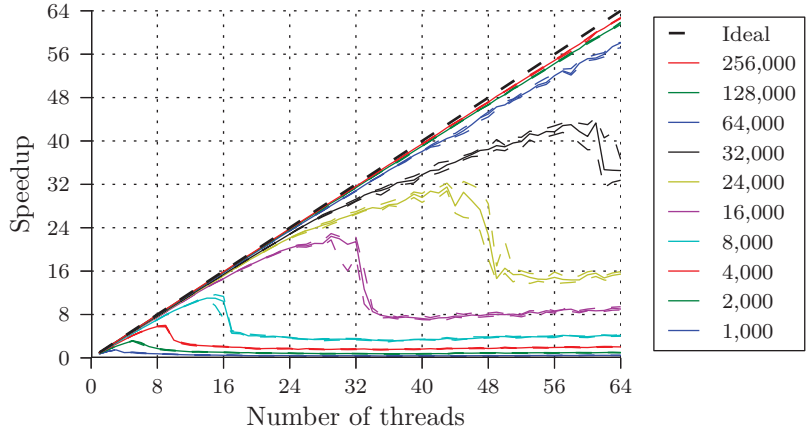


Figure 4.3. Speedup of SuperGlue for different task granularities, measured in cycles. Tasks have no dependencies, and memory management is not included. Dashed lines mark the first and third quartile of 15 runs.

4.2.4 Performance

To measure the performance of SuperGlue, we seek to find a measure that will capture the behavior of the runtime system rather than the behavior of the tasks. Therefore, we submit a number of independent tasks that perform no actual work, but only wait for a number of clock cycles. By varying the number of cycles to wait, we can investigate how fine-grained tasks SuperGlue can handle. The speedup is measured as the time from when the first task starts until the last task ends, compared with the number of tasks times the task waiting time.

Figure 4.3 shows the results. This experiment is run on a 64-core system with four AMD Opteron 6276 processors based on the Bulldozer architecture with a clock frequency of 2.3 GHz. For tasks as small as 64,000 cycles, corresponding to just below 30 microseconds, SuperGlue scales to 64 cores.

We have also compared our solution to a number of related efforts. In this experiment we borrow the dependency pattern from a numerical linear algebra algorithm; the tiled Cholesky factorization. This is a common example of an algorithm with a non-trivial dependency pattern, for which the data-dependency task-based approach is well suited. In this experiment we do not perform the actual computations. Instead, tasks again only wait for a determined number of cycles. By varying this waiting time, we see what task granularity the different frameworks need to scale well. The experiment was run on a desktop computer with a 4-core Intel Core i7 2600K processor.

Figure 4.4 shows how the efficiency of different task-based frameworks behave depending on the task granularity. SuperGlue reaches similar or higher efficiency than any other solution, for all task sizes.

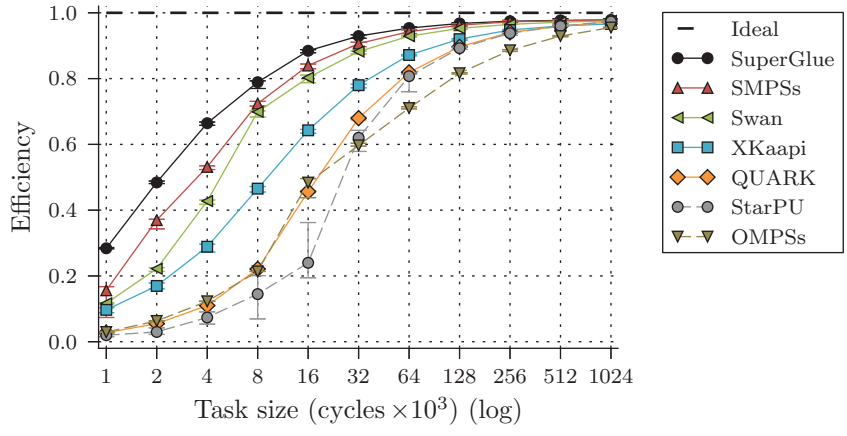


Figure 4.4. Performance comparison of different task-based frameworks. Dependencies as in a 20 × 20 tiled Cholesky factorization, but tasks only wait for a predefined number of cycles. Error bars show the first and third quartile of 20 runs.

4.3 Resource-aware task scheduling

A characteristic property of multicore architectures is that several cores share parts of the processor, such as caches and the memory bus. This means that cores have to compete for the resources, and may interfere with each other and run slower if a resource is oversubscribed.

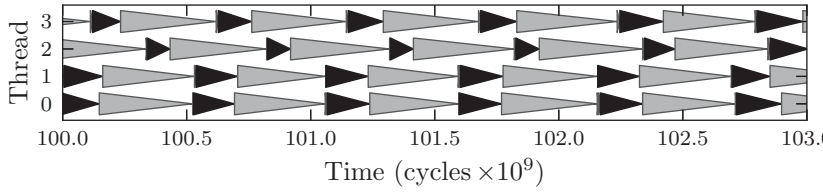
Not all workloads require the same resources. Some workloads are memory-bound and limited by the speed data can be transferred from the memory, while other workloads perform many computations on a small set of data, and are limited by how fast the processor can perform arithmetic operations.

If an application has several independent phases where one phase is, for instance, limited by memory bandwidth, and another phase is not affected by memory bandwidth at all, we want to schedule the tasks so that resources are used optimally. Instead of first running all bandwidth limited computations, and then all the memory bandwidth oblivious ones, we want to run a mixture of the different phases, to optimize the resource usage.

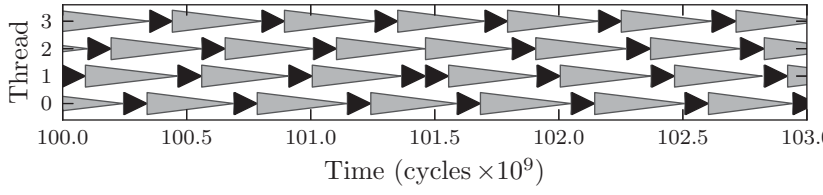
This situation holds for any shared resource. Other examples of resources where the scheduler should limit the number of concurrent accesses to avoid contention or exhaustion are memory capacity, the file system, and the network.

Taking resources into account when scheduling is investigated in Paper III. In this paper, we design a simple and efficient method to limit the number of concurrent accesses to a shared resource, and implement this in the OmpSs framework [18].

The main idea is illustrated by the execution trace in Figure 4.5. This figure shows how the task of an application are scheduled on the threads. Each task



(a) Without constraints, allowing several concurrent file accesses (black).



(b) Using the resources feature to constrain file accessing tasks (black).

Figure 4.5. Execution traces for an application that reads large data files from disk. There are three task types; read (black), compress (red), and write (green). The writes are very quick and barely visible here.

is represented by a triangle, and the color of the triangle indicates what kind of task it is.

This particular trace is from an application that reads large files from disk, compresses them, and then writes them back. Figure 4.5 (a), shows the initial schedule, where the scheduler has no notion of resources. The tasks that read from disk are represented by the black triangles. When several tasks read files concurrently, the tasks take noticeably longer time, while the read tasks that happened to be scheduled alone are much shorter.

In Figure 4.5 (b), the read tasks have been constrained so that only one task at a time is allowed to read from file. Here, all the read tasks take an equal amount of time to finish, and they are all as short as the shortest one from the unconstrained schedule.

4.3.1 The resource concept

To limit the number of concurrent accesses to a shared resource, we introduced the concept of resources into the OmpSs programming model.

With this addition, the programmer can annotate a task with resource requirements, together with the usual declarations of what data the task accesses. The programmer specifies the names of the resources, and the amounts that the task requires.

The resources are defined in a configuration file, which the runtime system reads at startup. This configuration file declares a name of the resource, and the total available quantity.

By putting the number of available resources in a configuration file, it is possible to declare hardware dependent resources, such as the amount of available memory or cache sizes, and be able to run the same binary on different computer systems, and adjust the configuration file to describe the capabilities of each system.

The special case when only a single task is allowed to use a resource is so common that we introduced a new access type; `commutative`. This access type is used to indicate that a variable can only be accessed by one task at a time, but without declaring an order of the tasks. This is the concept of the `add` access type in SuperGlue. For this special case, the resource is identified by the variable, and is not specified in the configuration file.

The resources concept is also used for implementing a way to reserve a number of worker threads for a group of tasks. In OmpSs, it is possible for a task to create child tasks, which will have to execute and finish before the parent task is considered finished. Using the resource concept, a parent task can declare that it requires a number of worker threads. The task is then scheduled only when that number of worker threads are available, and ready to exclusively execute the child tasks of the parent. Together with the socket-aware scheduler in OmpSs, this feature can be used to dedicate a whole socket to a certain task and its child tasks.

4.3.2 Automatic detection of resource requirements

Depending on how a task uses shared resources, it can be more or less sensitive to interference from other tasks. It can be seen in Figure 4.5 that the read tasks are sensitive to other read tasks, while compress tasks always take the same amount of time, regardless of which other tasks are running. This sensitivity can be detected automatically from execution traces. In Paper III, we describe a method to detect sensitivity from an execution trace of an unconstrained execution. From this information we are able to estimate the amount of a resource that a sensitive task should require to run as efficient as possible. We can also estimate what the total execution time would be if the scheduler took these resource requirements into account. This makes it possible to estimate the performance improvement that one can expect from constraining with respect to resources, from an unconstrained schedule.

4.4 Extending the model to distributed memory

As mentioned before, the class of software that we primarily target is from the area of scientific computing. In this area, the amount of data and computations can often be large, and there is a constant desire to be able to run larger and more detailed simulations. These simulations cannot be run on a single

shared-memory machine, but are run on clusters of multicore machines and supercomputers.

The industry standard for writing parallel software for distributed-memory systems is MPI (Message Passing Interface) [31]. MPI is a low-level interface which supports sending and receiving data between nodes, both in a blocking and in a non-blocking way. There are also collective communication routines, where the nodes collaborate to distribute or collect messages, and methods to synchronize a group of nodes.

The primary drawback of using a low-level interface is that the programming effort is high. This high effort introduces a risk that complex parallel patterns become inaccessible, and cause developers to implement simpler and less efficient parallelization strategies, introducing unnecessary synchronization points.

The task-based model could provide a user-friendlier and less error prone programming model than the MPI interface, and at the same time enable asynchronous execution of tasks also on distributed memory systems, avoiding unnecessary synchronization points, and thus improving the performance.

High level programming models for distributed memory do exist, for instance in the form of programming languages that provide the partitioned global address space (PGAS) model. The most prominent are UPC [39], Co-Array Fortran [32], and Titanium [43]. In the PGAS model, the programmer is presented with a global address space, and do not have to explicitly send and receive data.

Some dependency-aware task-based frameworks already have support for distributed memory. StarPU supports distributed memory over MPI in two different ways. There is one interface for sending and receiving data that is very similar to MPI itself, but that take StarPU handles instead of memory addresses [2]. This interface is useful when porting an application that already uses MPI. In the second interface, presented in [1], the user specifies which rank each data object belongs to, and then submit tasks as in the shared memory case. StarPU then automatically detects and inserts data transfers when needed.

The StarSs model has been extended to clusters with ClusterSs [36, 37], and support for distributed memory environments has also been introduced in OmpSs in [10]. Both these efforts use one-sided communication via the PGAS model instead of MPI. [10] takes a master-slave approach, where all tasks are created on the master, and then distributed among the slave nodes.

The leading dense linear algebra package, DPLASMA [7], uses the PaRSEC [8] runtime system (previously DAGuE [9]) for scheduling and managing tasks on distributed and heterogeneous architectures. It uses a symbolic representation of the task graphs, which avoids problems associated with large task graphs.

In Paper V we present DuctTeip (Distributed user-annotated concurrent tasks executed in parallel), which is the extension of our model to distributed mem-

ory platforms. We have also built another version of this framework, with slightly different design choices, for use in the software described in Paper IV.

4.4.1 Our distributed memory runtime system

Our solution is to use the data versioning approach from SuperGlue for tracking dependencies, with a few differences for distributed memory systems. Handles can no longer be decoupled from the data they represent, since the runtime system is responsible for transferring the data between nodes as needed. To make this possible, handles have a pointer to the data they represent, which must be a contiguous block of memory, and store the size of this block. Each handle also has a home node, which is the node that owns the data. The home node must be set before the handle is used for the first time.

Handles need to perform some additional book-keeping in order to know when data needs to be transferred. Each handle keeps track of which node last wrote to the data, and has a dictionary of all nodes that have a copy of the most recent version of the data.

Each node runs the same program, and submits the same sequence of tasks. When a task is submitted, the runtime system decides on which node it should be executed by looking at the accessed handles. The task will be owned by the node that owns the first handle that is written to. If no handles are written to, the task is instead owned by the first handle that is read. Tasks with no dependencies will be run on the node where they were created.

A single core per node is dedicated for MPI communication. It accepts requests to receive and send data from the worker threads. The requests are enqueued and realized through MPI calls in order. The MPI thread keeps one queue for send requests and one queue for receive requests per remote node in the system, and limits the number of outstanding transfers per node and direction.

When a task is submitted to the runtime system, and it has been decided which node should execute the task, the runtime system checks if it needs to transfer any data. The node that is selected to run the task will check if the required versions of all handles will be on the node when it is time to execute the task. Otherwise, it will enqueue a receive request from the node who last wrote the data, for each missing handle. If the submitted task is decided to be executed remotely, the current node checks if it needs to send any handles to the remote node.

If a node needs to receive data, it will schedule a future PublishData task. This means that no such task will be created just yet, but that it promises that such a task will be created later. The task that needs the data will then depend on this PublishData task. The PublishData task will be created by the MPI thread when the transfer is finished.

```
x.set_owner_rank(0);
y.set_owner_rank(1);

submit(new Write(x));
submit(new Copy(x, y));
submit(new Write(x));
```

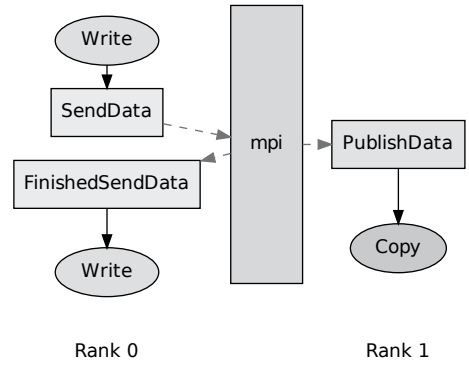


Figure 4.6. The code to the left shows the tasks that the programmer submits, and the figure to the right shows also the additional tasks that are submitted by the runtime system and how they interact with the MPI thread.

If a node needs to send data, it will submit a SendData task, and also schedule a future FinishedSendData task. Subsequent tasks that access the handle will depend on the FinishedSendData task, to prevent the data from being overwritten until the transfer is finished. The FinishedSendData task will be created by the MPI thread once the transfer is finished. Figure 4.6 illustrates the additional tasks created to transfer data between two nodes.

From the programmer's perspective, the only things that changes from the shared memory interface is that **`Handle`**s are replaced with **`MPIHandle`**s, which must be given home nodes and be associated with data, and that **`Task`**s operating on these handles are replaced by **`MPITask`**s. By deciding if handles should be of **`Handle`** or **`MPIHandle`** type (and similarly for tasks), the programmer can decide whether tasks are going to be executed locally, or be globally visible.

5. Application

5.1 Global climate simulation

The goal of the work behind this thesis is to make efficient use of multicore processors for scientific computing applications. In Paper IV, we verify that the programming model we have developed is practically useful, and that our implementations are efficient, by using our system to implement a global climate simulation application.

The application and method can be found in [20]. This article presents an interesting meshfree method for solving the shallow water equations on the sphere. In Paper IV, we translate the original MATLAB implementation to C++, and optimize and parallelize it using the SuperGlue runtime system.

The interesting part of this problem from the parallelization perspective is that it is a time-stepping algorithm where each time step depends on the previous. This limits the amount of parallelism that can be extracted from the algorithm. In addition to this, over 90 % of the execution time of the original MATLAB code was spent inside sparse matrix-vector multiplications, which are memory bandwidth limited operations. Since all cores share the same memory bandwidth, sparse matrix-vector products usually does not scale well on multicore processors.

5.1.1 Implementation

When translating the MATLAB code to a parallel C++ implementation, we first implemented a serial C++ code, and then extracted tasks and optimized for shared memory. Finally, the distributed memory version of the runtime system was developed.

In the serial C++ code, we changed the memory layout to conform to the access patterns of the algorithm. Three sparse matrices that had the same sparsity pattern were replaced with a single sparse matrix with three elements per position, so that the sparsity structure only had to be traversed once, and a number of vectors were merged into a vector of structs. In the sparse matrix vector product code, we were able to use 4-elements-wide SIMD instructions.

To rewrite the algorithm as tasks and extract parallelism, we divided the solution vector into chunks. Two different blocking strategies were evaluated: blocking the matrix row-wise, and blocking it in both dimensions. Blocking the matrix row-wise yielded blocks with equal numbers of nonzeros, but limited parallelism. With 2D blocking, some blocks had very few elements, causing very small tasks. An advantage however was that many blocks were empty

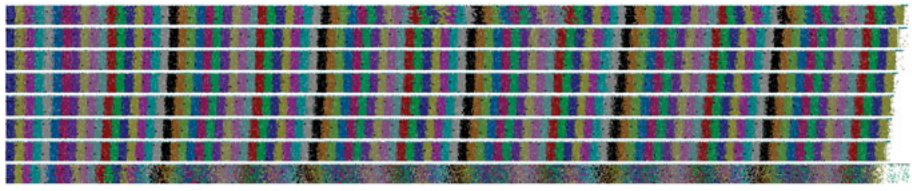


Figure 5.1. Execution trace of 100 time steps on 8 nodes. The colors indicate which time step the tasks are working on.

and could be discarded. In the end, the 2D blocking yielded more parallelism, and proved to perform better, so we selected this approach.

To be able to simulate a large number of time steps, we did not want to submit all the tasks at once, even though SuperGlue handles large number of tasks well. Instead we created a task for creating the tasks needed to take one time step. The task generation task also submits a new copy of itself, with a dependency to one of the blocks of the solution vector. This way, every time a time step is finished, a new task generation task has been submitted. The task generation tasks knows which time step it is creating tasks for, and stop submitting new tasks when the desired end time is reached. To start the program, we submit 5 task generation tasks, to have about 5 concurrent time steps to work on at any time during the execution.

In this implementation, we ended up having practical use of several features of SuperGlue. To generate tasks in an practical way, we submitted tasks from tasks, and we also use a feature to schedule and wait for a future task, submitted from an unknown task at some point in the future, to manage when to shut down the execution.

5.1.2 Performance results

The translation from MATLAB to sequential C++ alone yielded a speedup of 5 times. The shared memory version yielded only 5 times speedup over the serial when running on 16 cores. This was on an AMD Bulldozer architecture, where each pair of cores share a single floating point unit. The best theoretical speedup would thus be 8. We could measure that the sparse matrix-vector multiplication tasks took between 140 % and 160 % longer when running on all 16 threads compared to running on a single core, and that 98.9 % of the execution time was spent within tasks.

In the distributed memory version, running on 8 nodes gave a speedup of 6.9 times the shared memory version. This was with virtually no changes in the application code. The only work required was to decide how to distribute the blocks among the nodes.

Figure 5.1 shows an execution trace of the simulation code running 100 time steps on 8 nodes with 16 cores each. The colors show which time step the task

is working on. Node 0, drawn at the bottom, mixes tasks from different time step to a larger extent than the other nodes. In the end, it can be seen how all nodes send their part of the solution vector to the first node. All nodes start by working on the same time steps, but the last node take longer to execute the tasks, and towards the end there is about two time steps difference between node 1 and node 7. This is caused by an imperfect load balance. From the trace it looks like node 0 has too little work, which is probably why it is mixing tasks from different time steps more aggressively than the other nodes.

Compared with the sequential MATLAB code, our final version was able to execute 178 times faster, when running on 8 nodes.

6. Summary of papers

6.1 Paper I

K. Ljungkvist, M. Tillenius, D. Black-Schaffer, S. Holmgren, M. Karlsson, and E. Larsson. Using hardware transactional memory for high-performance computing. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*, May 2011, pp. 1660–1667.

Hardware transactional memory is a new primitive for synchronizing accesses to shared memory in multicore processors. In this paper, we evaluate if hardware transactional memory can be of benefit for scientific computing applications. Experiments are conducted on a prototype of the Rock processor from Sun, the first processor to implement hardware transactional memory.

We evaluate how efficient it is to use hardware transactional memory for updating floating point data atomically. Both micro-benchmark experiments and more realistic work-loads are implemented and evaluated. The results are compared against standard methods for concurrent updates of floating point values.

The conclusion is that hardware transactional memory can be used for this purpose, and if several elements can be updated at once, hardware transactional memory was faster than other approaches. However, when the algorithm could be rewritten to avoid concurrency, hardware transactional memory was not fast enough to be competitive.

6.2 Paper II

M. Tillenius. SuperGlue: a shared memory framework using data versioning for dependency-aware task-based parallelization. Technical Report 2014-010, Department of Information Technology, Uppsala University, March 2014.

In this paper, we develop a task-based library with data-dependent tasks targeting shared memory architectures.

What is new in comparison to previous work is that tasks depend on the data and resources they need, instead of on other tasks. This increases flexibility, since it does not need to be known which task produces or consumes a required piece of data.

Dependencies are managed through a new system that uses data versioning. The idea is to associate a version number with each managed object, which counts the number of times the object has been accessed. Dependencies are then expressed as requiring a certain version of an object.

Performance experiments verify that this scheme can be implemented very efficiently. We also perform experimental comparisons against several other well-known task based systems, and show that our runtime system performs as well as or better than the other runtime systems, and that it scales better for fine-graind task sizes.

6.3 Paper III

M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell. Resource-aware task scheduling. Technical Report 2014-001, Department of Information Technology, Uppsala University, January 2014.

In this paper, we consider the problem of resource contention in multicore processors, and how to take shared resources into account when scheduling tasks in a dynamic task execution runtime system.

The basic idea here is to limit the number of tasks of a certain type that are allowed to be scheduled concurrently. This is done by introducing the concept of a resource. A resource has a name and a quantity. The available quantity is defined by the system. Tasks can then declare that they require a certain amount of a resource. The runtime system uses this information and make sure not to schedule tasks that use more resources than there are available at a time.

The functionality is implemented in the OmpSs programming environment, in collaboration with Barcelona Supercomputing Center. We also implement a related special case; a new access type that indicates that only one task at a time can access a variable, without restricting the order of accesses. This permits a way to express that several tasks sum their contributions to the output, and can run in any order, which was previously difficult to accomplish.

We analyze the performance benefits that resource-aware scheduling may yield, and show how this can be predicted from execution traces of unconstrained runs, when the scheduler does not take resource contention into account.

6.4 Paper IV

M. Tillenius, E. Larsson, E. Lehto, and N. Flyer. A task parallel scattered node finite difference scheme for the shallow water equations on a sphere. Technical Report 2014-011, Department of Information Technology, Uppsala University, March 2014.

In Paper IV, we implement a software for global climate simulation. It uses a recently developed meshfree method for solving the shallow water equations on the sphere. This paper deals with the parallelization of this method.

The numerical method is dominated by sparse matrix-vector products, which are bandwidth-limited operations. Such operations typically do not scale well on multicore processors, where the memory bandwidth is shared among the cores. Another challenge is that the simulation uses a time-stepping scheme, where each time step depends on the previous. This limits the amount of parallelism that can be extracted.

To parallelize the method, we use a version of the SuperGlue library extended to support distributed memory systems. Compared to the original MATLAB implementation, our parallel version runs a simulation 178 times faster, when running on 8 nodes.

6.5 Paper V

A. Zafari, M. Tillenius, and E. Larsson. Programming models based on data versioning for dependency-aware task-based parallelisation. In *2012 IEEE 15th International Conference on Computational Science and Engineering (CSE)*, December 2012, pp. 275–280.

In this paper we present the DuctTeip framework, a dependency-aware task-based framework for distributed memory systems. Here we extend the data-versioning scheme for dependency management to distributed systems.

We take a hybrid approach, where SuperGlue is used for shared-memory parallelism on each node, and MPI is used for communication between the nodes in a cluster. That is, we use SuperGlue to fix the details within a multicore node, and then employ DuctTeip to tie the whole cluster together.

The result is a programming model where all threading and MPI calls are managed by the framework, and the programmer only works with tasks. Experimental results show that this is accomplished without sacrificing performance.

7. Swedish summary

Moderna processorer har flera beräkningskärnor, och kan alltså utföra flera beräkningar samtidigt. För att utnyttja dessa processorer effektivt måste arbetsuppgfiterna som ska utföras delas upp mellan processorkärnorna. Detta medför fler detaljer att hålla koll på, och öppnar för nya typer av programmeringsmisstag.

Syftet med den här avhandlingen är att förenkla utvecklingen av programvara som på ett effektivt sätt kan utnyttja flerkärniga processorer. Olika typer av programvaror kan ha olika behov. Fokus i det här arbetet är programvara för vetenskapliga beräkningar, och målet är att hitta mönster och byggstenar som går att abstrahera till återanvändbara konstruktioner.

För att effektivt utnyttja flerkärniga processorer måste mjukvaran delas upp i många små uppgifter, så att dessa kan utföras parallellt på de olika kärnorna. Vanligtvis finns det beroenden mellan dessa arbetsuppgifter, vilket begränsar dem från att alla utföras samtidigt. Dessa beroenden är en följd av att en arbetsuppgift behöver ett delresultat från en tidigare, eller från att det finns någon delad resurs som flera arbetsuppgifter behöver, men bara en kan utnyttja i taget.

Vi börjar med att titta på en ny konstruktion för synkronisering mellan kärnor i flerkärniga system; transaktionellt minne. Med transaktionellt minne kan man definiera att en bit av ett program utgör en transaktion. Då garanterar processorn att andra kärnor antingen kan observera hela transaktionen när den är klar, eller inte kan se den alls. Inga mellanliggande tillstånd är synliga. När transaktionen avslutas försöker processorn genomföra de ändringar som utförts inom den, så att de blir synliga för övriga kärnor. Transaktioner är inte garanterade att lyckas. Om någon annan kärna skulle ändra på en minnesposition som också ändras inom transaktionen så måste transaktionen avbrytas och förkastas, och programmeraren måste ta hand om den avbrutna transaktionen. Exempelvis genom att försöka igen.

I den här avhandlingen utför vi experiment på en prototyp av världens första processor med hårdvarustöd för transaktionellt minne. Vi genomför först stresstester för att undersöka hur systemet fungerar, och för att utvärdera om den nya funktionaliteten kan ge prestandaförbättringar för de fall vi är intresserade av. Vi testar sedan att utnyttja transaktionellt minne på mer realistiska applikationer, där vi använder det till att låta flera kärnor samtidigt göra atomära uppdateringar av matriser och vektorer av flyttal. Här är förhoppningen att den nya konstruktionen hanterar kollisioner mellan uppdateringar från olika trådar så effektivt att de vanliga strategierna som används för att undvika samtidiga uppdateringar kan undvikas, vilket skulle innebära betydligt enklare program.

Slutsatsen är att transaktionellt minne faktiskt är snabbare än de andra konstruktionerna vi använder för att göra samma sak, men att det fortfarande lönar sig att undvika att uppdatera gemensamt minne så långt det är möjligt.

Efter att ha undersökt primitiverna för synkronisering mellan kärnor tittar vi på vad som är lämpliga programmeringsmodeller för att på ett enkelt sätt utveckla parallella program för flerkärniga processorer, utan att förlora i prestanda.

För att höja abstraktionsnivån utvecklas i den här avhandlingen ett mjukvarubibliotek för programmering av flerkärniga processorer. Huvuduppgiften för biblioteket är att sköta schemaläggningen av deluppgifterna och hantera beroenden mellan dem. För att hantera beroenden får programmeraren skapa ett antal objekt, som motsvarar minnesblock och resurser. För varje arbetsuppgift beskriver programmeraren vad som är indata, utdata, och vilka resurser som behövs, genom att ange en lista av dessa objekt. För varje objekt definieras även en typ som berättar hur resursen används, till exempel om det är för läsning eller skrivning. Arbetsuppgifterna skapas och lämnas in till ett system som under programmets körning bestämmer i vilken ordning och på vilken processorkärna varje arbetsuppgift ska utföras, med hjälp av informationen om vilka objekt som arbetsuppgiften behöver.

Den här lösningen innebär att programmeraren inte behöver reda ut vilka beräkningar som beror av vilka andra, och inte heller behöver bestämma på vilken processorkärna en uppgift ska utföras. Samma program kan alltså utan ändring köras på datorer med olika många processorkärnor.

Det nya med systemet som beskrivs här är att beroenden hanteras mellan en deluppgift och det data eller de resurser deluppgiften använder, i stället för mellan deluppgifter. Nytt är även att beroenden hanteras genom att räkna versioner. Detta är både enkelt, flexibelt och effektivt. I jämförande experiment med andra liknande system visar sig den lösning som presenteras här vara den mest effektiva.

Vi utvecklar också ett sätt att begränsa schemaläggningen av uppgifter som utnyttjar delade resurser. Med den här begränsningen kan man undvika prestandaförluster på grund av att allt för många deluppgifter samtidigt försöker utnyttja en begränsad resurs som minne, bandbredd till minnet, filsystemet, eller nätverket.

Systemet utökas sedan till att även hantera datorer med flerkärniga processorer som är ihopkopplade med varandra för att ge ännu högre beräkningskraft. Den nya utökningen byggs som ytterligare ett bibliotek, som ligger som ett lager ovanpå det tidigare. Mot programmeraren behålls i stort sett samma användargränssnitt, vilket innebär att program skrivna med vår lösning för att använda delat minne med mycket små ändringar även kan användas på system med distribuerat minne.

Slutligen utvecklar vi en programvara för klimatsimulering som vi parallelliserar med det system vi byggt upp. Jämfört med en tidigare version av programmet, som inte utnyttjade parallellism alls, kan den nya parallella koden

när den kör på åtta datorer räkna fram ett resultat ungefär 175 gånger snabbare. Dessutom kan betydligt större problem än tidigare hanteras.

Acknowledgments

First of all I would like to express my gratitude to my advisor, Elisabeth Larsson. Thank you for all the support and for all your hard work!

Thanks also to my co-authors; Rosa M. Badia, Xavier Martorell, Afshin Zafari, Marcus Holm, Erik Lehto, Karl Ljungkvist, David Black-Schaffer, Sverker Holmgren, Martin Karlsson, and Natasha Flyer. It has been a pleasure to work with you.

On a more personal note, I would like to say hi to my mom, my dad, and my brother. Hi! You are like family to me.

I am also very grateful for all the wonderful colleagues I have had. I would like to especially mention Jens Berg, Sofia Eriksson, Magnus Grandin, Pavol Bauer, Stefan Hellander, Sven-Erik Ekström, Josefin Ahlkrona, Daniel Elfverson, Fredrik Hellman, Patrick Henning, and Hanna Holmgren. I love you, guys!

This work was supported by the Swedish Research Council and carried out within the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

References

[1] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault, *StarPU-MPI: task programming over clusters of machines enhanced with accelerators*, in Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface, EuroMPI'12, Berlin, Heidelberg, 2012, Springer-Verlag, pp. 298–299.

[2] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, *Data-aware task scheduling on multi-accelerator based platforms*, in 2010 IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS), Dec 2010, pp. 291–298.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, *StarPU: a unified platform for task scheduling on heterogeneous multicore architectures*, Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, 23 (2011), pp. 187–198.

[4] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, *The design of OpenMP tasks*, IEEE Trans. Parallel Distrib. Syst., 20 (2009), pp. 404–418.

[5] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta, *CellSs: a programming model for the Cell BE architecture*, in Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC '06, New York, NY, USA, 2006, ACM.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: an efficient multithreaded runtime system*, SIGPLAN Not., 30 (1995), pp. 207–216.

[7] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. Yarkhan, and J. Dongarra, *Distibuted dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA*, Tech. Rep. UT-CS-10-660, Innovative Computing Laboratory, University of Tennessee, September 2010.

[8] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra, *PaRSEC: exploiting heterogeneity to enhance scalability*, Computing in Science Engineering, 15 (2013), pp. 36–45.

[9] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, *DAGuE: a generic distributed DAG engine for high performance computing*, Parallel Computing, 38 (2012), pp. 37–51. Extensions for Next-Generation Parallel Programming Models.

[10] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. Badia, E. Ayguade, and J. Labarta, *Productive cluster programming with OmpSs*, in Euro-Par 2011 Parallel Processing, E. Jeannot, R. Namyst, and J. Roman, eds., vol. 6852 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 555–566.

[11] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, *The impact of multicore on math software*, in Applied Parallel Computing. State of the Art in Scientific Computing, B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, eds., vol. 4699 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 1–10.

[12] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, *SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks*, in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '08, New York, NY, USA, 2008, ACM, pp. 123–132.

[13] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, *Rock: a high-performance SPARC CMT processor*, IEEE Micro, 29 (2009), pp. 6–16.

[14] L. Dagum and R. Menon, *OpenMP: an industry standard API for shared-memory programming*, Computational Science Engineering, IEEE, 5 (1998), pp. 46–55.

[15] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, *Early experience with a commercial hardware transactional memory implementation*, in ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, New York, NY, USA, 2009, ACM, pp. 157–168.

[16] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski, *Early experience with a commercial hardware transactional memory implementation*, Tech. Rep. TR-2009-180, Sun Laboratories, 2009.

[17] J. Dongarra, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, A. YarKhan, W. Alvaro, M. Faverge, A. Haidar, J. Hoffman, E. Agullo, A. Buttari, and B. Hadri, *PLASMA users' guide*. `http://icl.cs.utk.edu/plasma/`.

[18] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, *OmpSs: a proposal for programming heterogeneous multi-core architectures*, Parallel Processing Letters, 21 (2011), pp. 173–193.

[19] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta, *Extending the OpenMP tasking model to allow dependent tasks*, in Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism, IWOMP'08, Berlin, Heidelberg, 2008, Springer-Verlag, pp. 111–122.

[20] N. Flyer, E. Lehto, S. Blaise, G. B. Wright, and A. St-Cyr, *A guide to RBF-generated finite differences for nonlinear transport: shallow water simulations on a sphere*, Journal of Computational Physics, 231 (2012), pp. 4078–4095.

[21] F. Galilée, J.-L. Roch, G. G. H. Cavalheiro, and M. Doreille, *Athapascan-1: on-line building data flow graph in a parallel language*, in Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, PACT '98, Washington, DC, USA, 1998, IEEE Computer Society, pp. 88–95.

[22] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, *XKaapi: a runtime system for data-flow task programming on heterogeneous architectures*, in 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS),

2013, pp. 1299–1308.
[23] P. Ghosh, Y. Yan, and B. Chapman, *Support for dependency driven executions among OpenMP tasks*, in Data-Flow Execution Models for Extreme Scale Computing (DFM), 2012, 2012, pp. 48–54.
[24] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, *FLAME: formal linear algebra methods environment*, ACM Trans. Math. Softw., 27 (2001), pp. 422–455.
[25] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim, *The IBM Blue Gene/Q compute chip*, Micro, IEEE, 32 (2012), pp. 48–60.
[26] M. Herlihy and J. E. B. Moss, *Transactional memory: architectural support for lock-free data structures*, SIGARCH Comput. Archit. News, 21 (1993), pp. 289–300.
[27] *Intel Cilk Plus*. `http://www.cilkplus.org/`.
[28] Intel Corporation, *Intel 64 and IA-32 architectures software developer's manual, volume 1*, February 2014.
[29] *Intel Threading Building Blocks*. `http://threadingbuildingblocks.org/`.
[30] J. Kurzak and J. Dongarra, *Implementing linear algebra routines on multi-core processors with pipelining and a look ahead*, in Applied Parallel Computing. State of the Art in Scientific Computing, B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, eds., vol. 4699 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 147–156.
[31] Message Passing Interface Forum, *MPI: a message-passing interface standard version 3.0*, September 2012.
[32] R. W. Numrich and J. Reid, *Co-array fortran for parallel programming*, SIGPLAN Fortran Forum, 17 (1998), pp. 1–31.
[33] OpenMP Architecture Review Board, *OpenMP application program interface version 4.0*, July 2013.
[34] J. M. Pérez, R. M. Badia, and J. Labarta, *A dependency-aware task-based programming environment for multi-core architectures*, in 2008 IEEE International Conference on Cluster Computing, 2008, pp. 142–151.
[35] B. Stroustrup, *The design and evolution of C++*, Addison-Wesley, New York, NY, USA, 1994.
[36] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta, *ClusterSs: a task-based programming model for clusters*, in Proceedings of the 20th International Symposium on High Performance Distributed Computing, HPDC '11, New York, NY, USA, 2011, ACM, pp. 267–268.
[37] ——, *ClusterSs: a task-based programming model for clusters*, Tech. Rep. UPC-DAC-RR-2011-14, Universitat Politecnica de Catalunya, 2011.
[38] R. M. Tomasulo, *An efficient algorithm for exploiting multiple arithmetic units*, IBM J. Res. Dev., 11 (1967), pp. 25–33.
[39] UPC Consortium, *UPC language specifications, v1.2*, Tech. Rep. LBNL-59208, Lawrence Berkeley National Lab, 2005.
[40] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, *A unified scheduler for recursive and task dataflow parallelism*, in 2011 International Conference

on Parallel Architectures and Compilation Techniques (PACT), October 2011, pp. 1–11.
[41] A. YarKhan, *Dynamic task execution on shared and distributed memory architectures*, doctoral dissertation, University of Tennessee, December 2012.
[42] A. YarKhan, J. Kurzak, and J. Dongarra, *QUARK users' guide: queueing and runtime for kernels*, Tech. Rep. ICL-UT-11-02, Innovative Computing Laboratory, University of Tennessee, 2011.
[43] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, *Titanium: a high-performance Java dialect*, in ACM, 1998, pp. 10–11.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1139

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-221241

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2014