universität
wien

Masterarbeit

Titel der Masterarbeit

"Programming Models for Parallel Computing"

Verfasser

Martin Wimmer, Bakk.

angestrebter akademischer Grad

Diplom-Ingenieur (DI)

Wien, 2010

Studienkennzahl lt. Studienblatt	A 066 940
Studienrichtung lt. Studienblatt	Scientific Computing
Betreuer	Ao. Univ.-Prof. DI Dr. Siegfried Benkner

Abstract

With the emergence of multi-core processors in the consumer market, parallel computing is moving to the mainstream. Currently parallelism is still very restricted as modern consumer computers only contain a small number of cores. Nonetheless, the number is constantly increasing, and the time will come when we move to hundreds of cores.

For software developers it is becoming more difficult to keep up with these new developments. Parallel programming requires a new way of thinking. No longer will a new processor generation accelerate every existing program. On the contrary, some programs might even get slower because good single-thread performance of a processor is traded in for a higher level of parallelism. For that reason, it becomes necessary to exploit parallelism explicitly and to make sure that the program scales well.

Unfortunately, parallelism in current programming models is mostly based on the "assembler of parallel programming", namely low level threading for shared multiprocessors and message passing for distributed multiprocessors. This leads to low programmer productivity and erroneous programs. Because of this, a lot of effort is put into developing new high level programming models, languages and tools that should help parallel programming to keep up with hardware development. Although there have been successes in different areas, no good all-round solution has emerged until now, and there are doubts that there ever will be one.

The aim of this work is to give an overview of current developments in the area of parallel programming models. The focus is put onto programming models for multi- and many-core architectures as this is the area most relevant for the near future. Through the comparison of different approaches, including experimental ones, the reader will be able to see which existing programming models can be used for which tasks and to anticipate future developments.

Zusammenfassung

Mit dem Auftauchen von Multicore Prozessoren beginnt parallele Programmierung den Massenmarkt zu erobern. Derzeit ist der Parallelismus noch relativ eingeschränkt, da aktuelle Prozessoren nur über eine geringe Anzahl an Kernen verfügen, doch schon bald wird der Schritt zu Prozessoren mit Hunderten an Kernen vollzogen sein.

Während sich die Hardware unaufhaltsam in Richtung Parallelismus weiterentwickelt, ist es für Softwareentwickler schwierig, mit diesen Entwicklungen Schritt zu halten. Parallele Programmierung erfordert neue Ansätze gegenüber den bisher verwendeten sequentiellen Programmiermodellen. In der Vergangenheit war es ausreichend, die nächste Prozessorgeneration abzuwarten, um Computerprogramme zu beschleunigen. Heute jedoch kann ein sequentielles Programm mit einem neuen Prozessor sogar langsamer werden, da die Geschwindigkeit eines einzelnen Prozessorkerns nun oft zugunsten einer größeren Gesamtzahl an Kernen in einem Prozessor reduziert wird. Angesichts dieser Tatsache wird es in der Softwareentwicklung in Zukunft notwendig sein, Parallelismus explizit auszunutzen, um weiterhin performante Programme zu entwickeln, die auch auf zukünftigen Prozessorgenerationen skalieren.

Die Problematik liegt dabei darin, dass aktuelle Programmiermodelle weiterhin auf dem sogenannten „Assembler der parallelen Programmierung“, d.h. auf Multithreading für Shared-Memory- sowie auf Message Passing für Distributed-Memory Architekturen basieren, was zu einer geringen Produktivität und einer hohen Fehleranfälligkeit führt. Um dies zu ändern, wird an neuen Programmiermodellen, -sprachen und -werkzeugen, die Parallelismus auf einer höheren Abstraktionsebene als bisherige Programmiermodelle zu behandeln versprechen, geforscht. Auch wenn bereits einige Teilerfolge erzielt wurden und es gute, performante Lösungen für bestimmte Bereiche gibt, konnte bis jetzt noch kein allgemeingültiges paralleles Programmiermodell entwickelt werden - viele bezweifeln, dass das überhaupt möglich ist.

Das Ziel dieser Arbeit ist es, einen Überblick über aktuelle Entwicklungen bei parallelen Programmiermodellen zu geben. Da homogenen Multi- und Manycore Prozessoren in nächster Zukunft die meiste Bedeutung zukommen wird, wird das Hauptaugenmerk darauf gelegt, inwieweit die behandelten Programmiermodelle für diese Plattformen nützlich sind. Durch den Vergleich unterschiedlicher, auch experimenteller Ansätze soll erkennbar werden, wohin die Entwicklung geht und welche Werkzeuge aktuell verwendet werden können.

Contents

Chapter 1

Introduction

In 1965 Gordon Moore first observed that the number of circuits on microprocessors had doubled roughly every two years, and he predicted that this trend would hold on for at least some time [Moore, 1965]. *Moore's law*, which is how this prediction is called today, still holds true. Until the year 2002, the exponential increase in processor density was accompanied by an exponential increase of processor performance, due to increased clock-rates of computer processors. Unfortunately, this is not the case any more.

One reason for this is the, so called, *power wall*, a phenomenon coming from a cubic relationship between power consumption and the processor clock rate. Partly, the rising power consumption can be countered by decreasing the size of the integrated circuits. The problem is that, as the size decreases, the power leakage rises. Parallel computing has the advantage of increasing theoretical processor throughput without increasing the clock rate, which means that power consumption does only increase with die size.

Another problem, apart from the power wall, is that memory latency does not decrease and the bandwidth does not increase at the same rate as the processor speed increases. This means that memory access becomes a bottleneck, and processors spend a large amount of time waiting for data. While once memory access was cheap, it can now take a few hundred cycles to fetch data from memory. Therefore, a lot of effort is put into caching and prefetching to reduce the negative effects of the *memory wall*. [Wulf and McKee, 1995]

For some time *instruction level parallelism* (*ILP*) has been seen as a solution for increasing processor performance. For ILP to work efficiently with long pipelines, complex branch prediction and speculative execution techniques were needed. Unfortunately, the gains from ILP are limited as the time lost to pipeline stalls increases with pipeline length. After a certain point it is impractical to enhance branch prediction and speculative execution to reduce pipeline stalls because of increasing processor complexity and power usage. This phenomenon is called the *ILP Wall* [Asanovic et al., 2006].

[Asanovic et al., 2006] formulates the dilemma created by these three "walls" as: "Power Wall + Memory Wall + ILP Wall = Brick Wall". This means that through these effects the increase of serial performance is diminishing.

The only known way to overcome this dilemma is to introduce parallelism. On parallel platforms the performance of a single processing unit is sacrificed in favour of a higher number of processing units. This allows to overcome the power wall by increasing the theoretical peak performance without increasing power consumption. The memory wall can be overcome by ensuring a parallel workload which is high enough. When a thread stalls because of a memory access, the processor can meanwhile switch to execute another thread until the data is delivered.

1.1 Amdahl's and Gustavson's law

Using parallelism to increase program performance is actually not a new idea, only that for a long time exploitable parallelism was thought to be limited. This was first stated by Gene M. Amdahl in [Amdahl, 1967]. The central point of Amdahl's argument, which is often referred to as "Amdahl's law", is that the speedup of a parallel application is always limited by the serial portion. This is demonstrated by the following formula:

$$Speedup = \frac{1}{r_s + \frac{r_p}{n}} \tag{1.1}$$

If we increase the number of processors (n) to infinity, the maximum achievable speedup is $\frac{1}{r_s}$, where r_s denotes the sequential portion of the code. As an example, a program with a serial portion of ten percent would have a maximum possible speedup of 10.

Later on "Amdahl's law" was often criticized. The best known argument against it was thought up by John L. Gustafson in [Gustafson, 1988]. He noticed that faster processors were generally not used to calculate the same problems in less time, but to solve larger problems while keeping the run-time constant. Interestingly, for most scientific problems the serial portion stays the same when increasing the problem size, whereas the parallel portion grows larger. Thus, when fixing the run-time instead of the problem size we get the following formula:

$$Scaled\ speedup = n + (1 - n) * r_s \tag{1.2}$$

In this case, linear speedup is achieved, which is what is desired. This argument is often called "Gustafson's law"

Certainly, there are also problems that are difficult or even impossible to parallelize. However, when we hit the three "walls" described in Section 1, the question is not about getting linear speedup anymore but about getting any speedup at all. Parallelism seems to be the only way to achieve it.

Chapter 2

Architectures

2.1 Shared memory architectures

In *shared memory* architectures multiple processors share a global address space. Shared memory architectures allow for convenient parallel programming models as data can be accessed from any processor. Coordination between processes can be done via shared variables. Shared memory architectures can be categorized into *Uniform Memory Access* (*UMA*) and *Non-Uniform Memory Access* (*NUMA*) architectures. In UMA architectures memory accesses by all processors have the same latency. On the other hand, in NUMA architectures this restriction is taken away, and each processor has its own local memory, which can be accessed faster than remote memory. A problem with NUMA architectures is that the difference between local and remote memory is usually not reflected in the shared memory programming model. This can introduce some inefficiencies.

As memory accesses are slow in general, modern CPUs have a hierarchy of caches that is used to reduce latency for subsequent accesses to memory pages. Typical consumer processors nowadays have two or three levels of cache where level 1 cache is nearest to the processor but has only little capacity. In most cases level 1 and level 2 caches are exclusive for each processor. This generates the problem of having to maintain coherency between caches, which ensures that as soon as a processor modifies a memory location, the corresponding cache lines of all other processors are invalidated. An overview over different cache coherency protocols can be found in [Archibald and Baer, 1986].

One problem that can limit scalability of cache-coherent systems is *false sharing*. It comes from the fact, that coherency is not maintained for single bytes, but for cache lines. (The line size is dependent on the processor architecture.) This can lead to processors competing for exclusive access to the same cache line even if they access different memory locations.

Shared memory architectures are widely used as most personal computers sold today contain a *multi-core* processor. The roadmaps of all major processor vendors show that the number of cores on a chip is going to increase in the next years. This means that multi-core processors will be the main source of parallelism on shared memory architectures in the near future. *Si-*

multaneous Multithreading (*SMT*) is a special type of shared memory architecture. An SMT processor provides hardware support for a given number of threads to increase the utilization of the processor's functional units and to reduce pipeline stalls. More information about SMT can be found in [Tullsen et al., 1995]. Another type of shared memory architecture is *Symmetric Multiprocessing* (*SMP*), where multiple processors are built into one machine. With multi-core processors the importance of SMP is decreasing. When the limits of shared memory architectures are reached, SMP architectures might completely disappear in favour of multi-core architectures.

2.1.1 Memory consistency

In order to enable the writing of correct parallel programs on shared memory architectures, some guarantees concerning memory semantics have to be given to the programmer. A formal specification of memory semantics [Adve and Gharachorloo, 1996] is called a *memory consistency model*. If a programmer follows the rules of a memory consistency model it is guaranteed that the results of all memory operations are predictable and that the memory itself is consistent.

The easiest way to achieve memory consistency is through the *sequential consistency model*. In sequential consistency, all memory operations in a code block have the same semantics as in a sequential program. Although this consistency model would also be the most intuitive for programmers, it is not supported in most modern parallel systems. The reason is that sequential consistency restricts the use of many performance optimizations that are commonly used in processors and compilers. One example of such an optimization is *latency hiding* where high-latency read operations are moved before other memory operations to reduce the time lost waiting for the data.

Therefore, most compilers and processors only support *relaxed memory consistency* nowadays. [Adve and Gharachorloo, 1996] describes some relaxed memory consistency models: relaxing the *Write* to *Read* order, relaxing the *Write* to *Read* and *Write* to *Write* program orders, *weak ordering*, *release consistency* and the models used in the *Alpha*, *RMO* and *PowerPC* architectures.

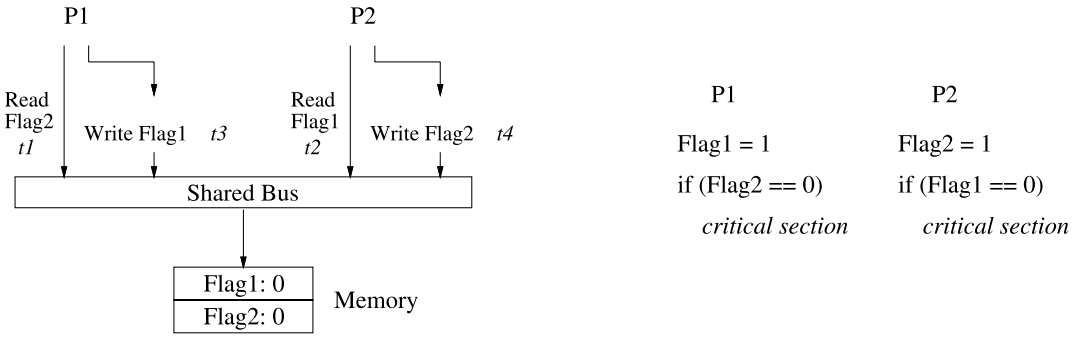


Figure 2.1: How a write buffer can violate sequential consistency. (Source: [Adve and Gharachorloo, 1996])

Relaxing the *Write* to *Read* program order

Relaxing the program order constraints to allow reordering of *Write* operations in respect to *Read* operations to different memory locations allows the use of some optimizations. Figure 2.1 shows an optimization where a write buffer with bypassing capability is introduced. Write operations are issued to the write buffer, and the processor does not wait for the write operation to complete. In the program in Figure 2.1, which depicts Dekker's algorithm [Dijkstra, 1965], sequential consistency is violated as the read operations can execute before the write operation completes. In a sequentially consistent system it is not possible that both read operations return zero, and so the critical section can only be executed by one processor. This assumption does not hold true when the Write to Read program order is relaxed.

Relaxing the *Write* to *Read* and *Write* to *Write* program orders

Allowing the reordering of *Write* operations with respect to other *Write* operations to different memory locations in addition to the relaxation of the *Write* to *Read* order, enables optimizations through pipelining or overlapping of writes. Figure 2.2 shows an example where this kind of optimization can lead to problems: If the write operation to the variable *Head* completes before the write to *Data*, it can happen that the second processor reads out the old value of *Data*.

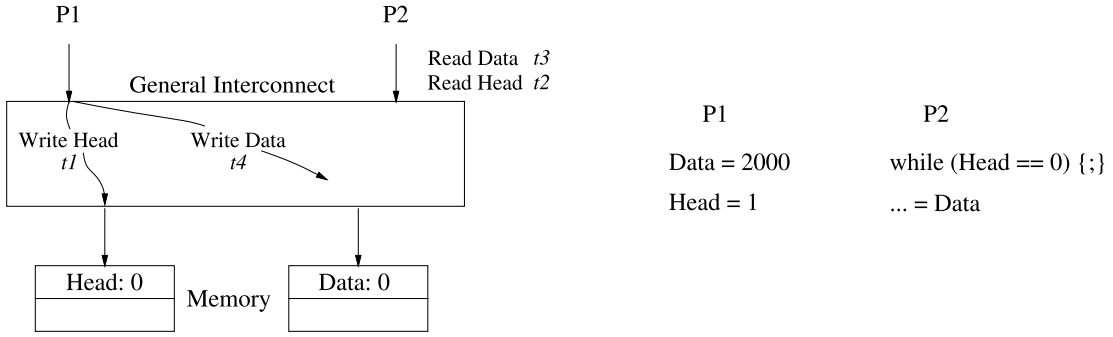


Figure 2.2: How overlapped writes can violate sequential consistency. (Source: [Adve and Gharachorloo, 1996])

Relaxing the order between all data operations

Some memory consistency models go even further by relaxing the order between *Read* operations and a following *Read* or *Write* operation. This relaxation allows further optimization as *Read* operations can be executed in a non-blocking manner. The problems with this optimization are depicted in Figure 2.3. It uses the same code as Figure 2.2 but this time we assume that we only optimize by allowing non-blocking reads. In this case, the second processor may read the variable *Data* before it has been written by the first processor.

To overcome the problems created by relaxing the order of data operations, relaxed consistency models typically provide some form of fence statement. A fence statement creates a

reordering barrier that allows to specify which statements have to be sequentially consistent. Some models, like the *Weak Ordering* and the *Release Consistency* models, which are described below, add additional semantics that makes it easier to ensure consistency.

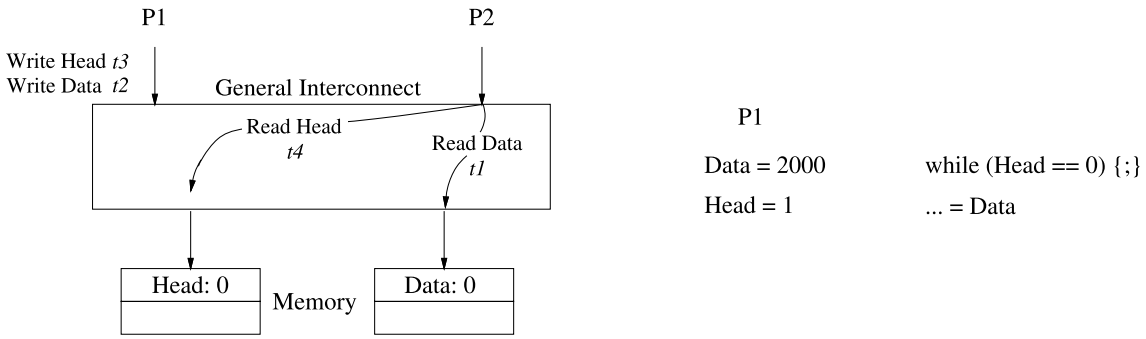


Figure 2.3: How non-blocking reads can violate sequential consistency. (Source: [Adve and Gharachorloo, 1996])

The Weak Ordering model

In the *Weak Ordering* (*WO*) model, operations are classified into *data* operations and *synchronization* operations. There are no reordering restrictions between data operations. When a synchronization operation is issued, the processor has to make sure that all previous operations have been completed before it is processed. Moreover, no operations are issued until the previous synchronization operation completes.

Release Consistency

Release Consistency is a more fine-grained model than Weak Ordering as it uses a hierarchical model of operations. On the top level, operations are distinguished between *special* and *ordinary* operations. *Special* operations roughly correspond to synchronization operations in the Weak Ordering model, whereas *ordinary* operations roughly correspond to data operations. In Release Consistency special operations are further distinguished into *sync* and *nsync* operations. *Nsync* operations correspond to asynchronous data operations or other special operations that are not used for synchronization. *Sync* operations are further distinguished into *acquire* and *release* operations. An *acquire* operation is a read operation used to gain access to a set of shared locations, as, for example, a lock operation. A *release* operation is a write operation used to grant permission for access to a shared location. There are two variants of Release Consistency which differ in the program order that is enforced between the operations. They are described in more detail in [Adve and Gharachorloo, 1996].

2.1.2 Difficulties

Although they are relatively easy to program, shared memory architectures are not without problems. One major problem is that maintaining cache coherency and memory consistency does not scale well for a large number of processors. This puts an upper limit to the number of processors in a shared memory system. In fact, current supercomputers only use shared memory inside computation nodes, but not on the large scale. Another problem is that shared memory programming models are currently not able to express locality of memory, which can help to provide better efficiency on NUMA architectures. On the other hand, UMA architectures, where a concept of locality is not needed, are very difficult to implement efficiently for a larger number of processors. This means that in the future NUMA architectures will become dominant.

2.2 Distributed memory architectures

In *distributed memory* architectures each processor has its own local memory that it operates on. Communication between processes is typically done via messages. To be able to operate on a remote memory location, a processor first has to retrieve the data in it and copy it into local memory. Distributed memory systems provide better scalability compared to shared memory architectures. The main reason for this is that shared memory architectures usually use a bus topology for inter-processor communication, whereas network topologies are used in distributed memory systems. Another reason is the overhead created by memory consistency and cache coherency protocols, which increases with the number of processors. In fact, there are no more shared memory systems in the Top500 list of supercomputing sites today (see [Top, 2009]).

Unfortunately, the good scalability of distributed memory architectures comes at the price of difficult programmability. As communication between processors is costly, the programmer needs to ensure that each processor has the data it needs in its local memory so that only little communication is necessary. Some programming models like *HPF*, *Chapel* and *X10* have tried to abstract part of this away by automatically distributing the data according to patterns specified by the programmer. This is described in more detail in Chapter 4.

Distributed architectures exist on different scales. They differ in the type of interconnect between computation nodes. Depending on the problem and the machine it is run on, the latency and bandwidth of the interconnect can become the limiting factors for scalability.

2.3 Modern cluster architectures

As both shared and distributed memory architectures have their advantages, they are often combined in modern cluster architectures. Modern clusters consist of distributed memory nodes, where each node is a shared memory system. For the purpose of this work we will primarily focus on homogeneous clusters, where all nodes consist of identical hardware.

An example of a modern cluster architecture is depicted in Figure 2.4. This example cluster consists of four shared-memory nodes, where each node contains four dual-core processors. These

nodes are connected with each other using a mesh topology, which means that each node has a direct link to each other node. Please note that the mesh topology is only practical for a small number of nodes.

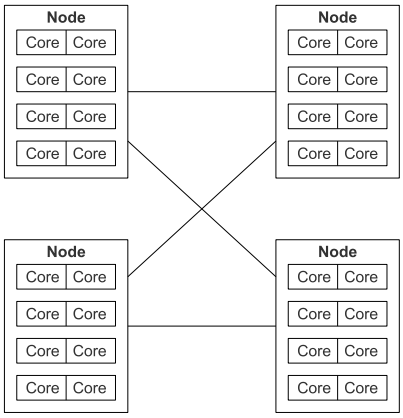


Figure 2.4: Example of a modern cluster architecture.

Cluster architectures add additional complexity to parallel architectures, as there is a distinction between parallelism inside each node and parallelism on the cluster level. For a program to efficiently utilize such an architecture, it has to be aware of the architecture. For example, tasks that have to do a lot of communication with each other should be placed inside a shared memory node, whereas tasks with little communication among each other could be put on different nodes.

When combining distributed memory architectures with shared memory architectures, there is also a discrepancy between the programming models that can be used. The programmer either has to use a distributed memory programming model for the whole program or he can combine two programming models like *MPI* and *OpenMP* instead, where one programming model is used for the distributed memory level and the other is used for shared memory parallelism.

Being able to efficiently utilize modern cluster architectures is one of the most challenging tasks. Cluster architectures are already very common, therefore it is necessary for programming models to be able to utilize them.

2.4 Heterogeneous architectures

Shared memory as well as distributed memory architectures have one thing in common: They are both based on the idea that all processors in a system are homogeneous. Over time it has become clear that for some tasks it can be an advantage to use specialized processors instead of general purpose ones. Architectures mixing conventional processors with specialized ones are often called *heterogeneous architectures*. Throughout this work the term *heterogeneous architectures* always refers to such architectures.

Especially accelerator cards for cryptography and graphics processing have become popular

over time. As the requirements in graphics processing rose, graphics processors (*GPU*) began to support more programmability, which led to the creation of *General Purpose Graphics Processing Unit*s (*GPGPU*s). GPGPUs are processors that are optimized for computationally demanding highly parallel tasks with little control code. Such tasks cannot only be found in image processing, but also in other fields like molecular biology or physics. More information about graphics processors can be found in [Blythe, 2008].

Another approach to heterogeneous architectures, where accelerator units are put directly on a processor, was realized by IBM in the *Cell Processor*. The Cell Processor is a heterogeneous multi-core processor that consists of one general purpose core called the *PowerPC Processing Element* (*PPE*) and eight specialized accelerator cores called the *Synergistic Processing Element*s (*SPE*s).

Both approaches, the GPGPUs and the Cell Processor, have in common that they force the programmer to explicitly decide which code is to run on which processor. Standard shared memory processors provide cache coherency and memory consistency in their shared address space and are typically used to implement the control logic of the program. Specialized processing units, on the other hand, are used as accelerators for computationally intensive tasks.

Heterogeneous architectures add another level of complexity to existing architectures. They provide a high theoretical peak performance at the cost of additional complexity for the programmer. Especially in combination with modern cluster architectures they can become very challenging to program. It has yet to be shown how this complexity can be handled.

Chapter 3

Concepts

Programming of parallel applications provides additional complexity compared to serial programming. This section presents some parallel computing concepts that are relevant for the application programmer. Chapter 4, which gives an overview over parallel programming models, shows how these concepts are supported in different programming models.

3.1 Locality of reference

Although computer programs might require large amounts of memory, they only access a relatively small amount of it at any instant of time. This property allows modern virtual memory systems to create the illusion of large amounts of fast memory by making sure that memory locations which will be used in the near future are stored in caches. The difficulty is, of course, to predict which memory locations will be accessed in the near future. This is done using the *locality principle*.

There are two types of locality:

1. *Temporal locality*: Memory locations that are referenced are more likely to be referenced again in the near future.

2. *Spatial locality*: Memory locations near a referenced memory location are likely to be referenced too.

Modern computer architectures take advantage of the locality principle to implement cache hierarchies. Caches provide quick access to some memory locations, but are very restricted in size. When a referenced memory location is not available in a cache, a whole memory line is fetched and stored in the cache. If the cache is full, one line is chosen to be replaced using a specific strategy (e.g. least recently used). Temporal locality is used in caches, as memory locations that are referenced are kept in the cache for future accesses. Caches also make use of spatial locality, as whole cache lines are fetched, not only the referenced memory location.

In parallel systems, the locality principle can be used to predict which processor will access which memory location. On shared memory systems, each processor has its own memory hier-

archy, and *cache coherence protocols* ensure that cache lines are invalidated as soon as another processor modifies them. Page migration strategies can be used to migrate memory pages from one processor to another if the other processor accesses them more frequently. [Nikolopoulos et al., 2000] This can reduce memory access latency in some programs.

One common problem with caching is *thrashing*, which is a situation where system performance degenerates. [Denning, 2005] Thrashing can for example occur when multiple memory locations compete for the same line in memory so that both processor's caches are invalidated all the time. This can even happen if each processor works on different memory locations, as long as these memory locations are in the same cache line. This is called *false sharing*. [Patterson and Hennessy, 2008, p468ff.]

3.2 Work distribution

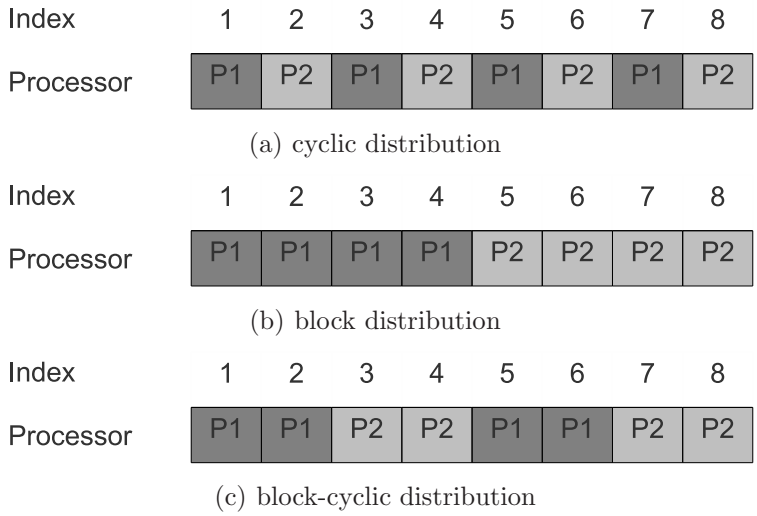
For a program to be able to run in parallel, it has to be decomposed into computationally independent work units that are then distributed onto different processors. In general, such a *work distribution* may either be derived from *task* or from *data parallelism*.

When deriving a work distribution from task parallelism, a problem has to be broken down into independent tasks. Those tasks do not necessarily have to be completely independent, as long as managing the dependencies only uses little time compared to the runtime of the tasks. On one side this can be achieved by allowing computationally independent kernels to run in parallel. Such tasks are easy to find, but can only provide minimal gains, as the number of such tasks is limited and does not scale with the problem size.

Fortunately, other types of task decomposition scale better. For example, distinct calls to a function can be mapped to tasks. This approach works especially well for recursive problems, like the divide and conquer class of algorithms, but can also be used in a variety of other settings. The Cilk programming language described in Section 4.1.3 is focused on such a *functional decomposition* as primary source of parallelism. Another source of task parallelism can be found in computationally independent loop iterations, where each iteration can be seen as a separate task. Most parallel programming languages provide some kind of parallel loop directives that allow to execute loop iterations in parallel. As the number of iterations often exceeds the number of processors, different strategies exist for how the iterations can be mapped onto processors. Figure 3.1 shows standard strategies that are often used.

In many cases, the most computationally intensive part of a program revolves around computations on, or manipulations of, a large data structure. In such programs, similar operations are executed for each element of the data structure. The goal of *data decomposition* is to split a data structure into multiple parts, which are then operated on by parallel tasks. As an example, a matrix may be split into submatrices, and each task can work on one of these submatrices. This type of decomposition has been used for one of the matrix multiplication kernels used for the experiments in the context of this work. It is described in more detail in Chapter 5.

Whether task or data decomposition should be used to parallelize an application, depends



(a) cyclic distribution

(b) block distribution

(c) block-cyclic distribution

Figure 3.1: Different types of loop distribution.

on the problem. For problems revolving around large data structures, data decomposition often has the advantage of having less data dependencies compared to a solution based on task decomposition. In distributed memory settings this becomes even more relevant, as the data is typically distributed onto different nodes. Therefore, the work distribution is often directly derived from the data distribution to minimize the communication overhead.

Often, task and data decomposition are not used on their own, but instead they are combined to be able to expose more parallelism. The programmer starts by using one of these patterns, and then further splits up tasks using the other pattern. A special case of such a combination can be found in *pipelining*. It allows working around problems with operations that are dependent on the results of previous computations. Analogous to an assembly line, data resulting from one operation is passed to sequentially dependent operations. The advantage is that a dependent task does not have to wait until the previous task has finished its work on all data, but it can start after a small latency instead. Figure 3.2 shows how a pipeline operates: In the first step, pipeline stage 1 works on the data element C_1. In the second step, C_1 is processed by pipeline stage 2 and pipeline stage 1 works on C_2. At step 4, the pipeline is filled completely and can operate at its full speed. The figure shows that a pipeline needs a few steps at the beginning to achieve full concurrency. At the end of a computation, concurrency decreases again until the pipeline is emptied. To minimize the negative effect on performance by *filling* and *draining the pipeline*, the number of pipeline stages should be small compared to the amount of processed data.

In general, pipelines increase the throughput of data by reducing the run-time per piece of data from the time needed to calculate the whole pipeline to the time needed for the longest pipeline stage. This means that for good performance of a pipeline, it is necessary to make the pipeline stages equally long. Wherever possible, non-deterministic stages should be avoided. Especially stages that might invalidate data following in the pipeline can be limiting to average

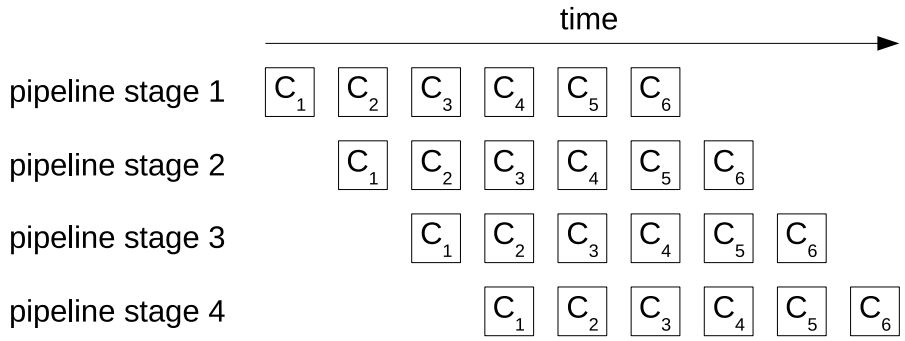


Figure 3.2: Operation of a pipeline. (Source: [Massingill et al., 2005])

throughput and might even make performance worse than in a design without a pipeline. Most modern computer processors use some form of *Instruction level parallelism* (*ILP*), which is realized using a pipeline for processing instructions. The possible gains through the use of ILP are limited though, as the results of conditional statements cannot be determined before they are actually executed. Therefore, the pipeline can not always be completely filled. [Massingill et al., 2005]

3.3 Data distribution

A problem relevant primarily for distributed memory architectures is data distribution. Algorithms in scientific applications often operate on large arrays. For these arrays it has to be decided how they are distributed between distributed memory nodes to minimize the communication demand between nodes. Replication of complete arrays is only an option in rare cases, as it increases the memory demand of an application. In most cases data is distributed over computation nodes by a specific pattern. Depending on the algorithm, different patterns might provide good efficiency. Work distribution for an algorithm is often derived then from the data distribution to minimize communication demand.

Data distribution is usually less relevant for shared memory architectures. However, for NUMA architectures, a good memory layout may help to optimize an algorithm by reducing the number of remote memory accesses. As the latency for remote memory accesses in NUMA architectures is lower compared to distributed memory architectures, manual data distribution is not the only way to achieve efficient data layout. [Nikolopoulos et al., 2000] describes how performance similar to manual data distribution can be achieved for some applications with page migration strategies. Still, manual data distribution can reduce the number of false sharing situations compared to page migration strategies, and, therefore, reduce problems with thrashing.

In general, to distribute a data structure onto multiple processors, its index space has to be split up into multiple sub-spaces that can be mapped to different processors. This process is called *domain decomposition*. As an example, a simple form of domain decomposition is the

cyclic decomposition where indices of the domain of the data structure are mapped to indices in the processor array in a cyclic fashion. Figure 3.3 shows an example where a 4x4 matrix is mapped onto a 2x2 array of processors using cyclic decomposition. A more complex example of a domain decomposition can be found in [Chapman et al., 1992].

$$\begin{pmatrix} P(0,0) & P(0,1) & P(0,0) & P(0,1) \\ P(1,0) & P(1,1) & P(1,0) & P(1,1) \\ \\ P(0,0) & P(0,1) & P(0,0) & P(0,1) \\ P(1,0) & P(1,1) & P(1,0) & P(1,1) \end{pmatrix}$$

Figure 3.3: Mapping of a 4x4 matrix onto a 2x2 processor array using cyclic domain decomposition.

In low-level parallel programming models like MPI, the programmer has to do domain decomposition manually and to explicitly write communication code for remote memory accesses. Because of the complexity of doing this, research has been done to provide a high-level interface for data distributions. The programmer should only need to specify the kind of data distribution that should be used for a data structure, and the compiler and run-time should automatically map global array indices to processors and provide implicit communication where needed. One of the first programming languages supporting such structures was *Kali* [Koelbel and Mehrotra, 1991]. This approach has been developed further in *High Performance Fortran* (*HPF*) [Loveman, 1993], which is described in Section 4.2.2. Although HPF itself was not successful commercially, its approach to data distribution has been refined and is now used in some experimental programming models like *Chapel* and *X10*.

3.4 Synchronization and Communication

In typical parallel programs, some type of coordination and exchange of data is necessary. In distributed memory settings, this is typically realized with communication primitives that allow for explicitly sending messages and data from one node to another. In shared memory environments, exchange of data is not necessary as each processor has complete access to data in the shared memory. Instead, synchronization is important on shared memory machines, which is essential to ensure consistency of shared data and can also be used as a means to coordinate computations. Some typical synchronization and communication constructs will be described in the following paragraphs.

3.4.1 Synchronization

Race conditions

Non-deterministic sections of a program that can lead to an erroneous program behaviour are called *race conditions*. [Netzer and Miller, 1992] defines two types of race conditions: *data* or

atomicity race and *general* or *determinacy race*.

A *data* or *atomicity race* results in a critical section not being executed atomically due to improper synchronization, which can lead to an improper update of a shared variable. All other race conditions are called *general races*. (Sometimes they are also called *determinacy races*.) A typical example of a general race occurs when a specific order should be enforced between operations, and the order is not correctly enforced.

Mutual exclusion

A *lock* or *mutual exclusion* is a resource that can be exclusively acquired by one thread. When one thread holds a lock, no other thread can acquire it. Locks can be used to coordinate accesses to critical sections by only allowing a thread that holds a specific lock to process the critical section. Some programming models, like *OpenMP*, provide higher-level constructs to define critical sections. Additionally, locks are often used to limit access to a shared resource by requiring a thread to acquire a lock before doing certain operations on the shared resource. When multiple locks are combined, more complex lock semantics is possible, as, for example a *read-write lock* that allows an arbitrary number of read accesses at the same time but ensures that no other thread reads or writes the resource as soon as one thread has locked it for writing. Some programming models, like *Threading Building Blocks*, have built-in support for read-write locks.

Mutual exclusion is only relevant in shared memory settings, as there are no shared resources in distributed memory environments.

Barriers

A *barrier* is a high-level synchronization construct used to suspend the execution of threads at a certain point in the code until it is reached by all threads. This construct is typically used to make sure a calculation has been completed before doing some processing on the results. Barriers are relevant in shared memory as well as in distributed memory settings.

Transactional Memory

The concept of transactions originally comes from the database community [Gray and Reuter, 1993]. A transaction provides atomic execution of a critical section and ensures that changes on variables performed during execution of the section become visible after the transaction has been completed successfully. Transactional memory can be implemented with either optimistic or pessimistic concurrency control. In optimistic concurrency control, a transaction is executed and it is assumed that no conflict occurs. If a conflict occurs, one of the conflicting transactions is aborted and rolled back. In pessimistic concurrency control, exclusive access to a resource is acquired before it is modified. In this case transactions are only rolled back when a deadlock occurs. Transactional memory can either be implemented entirely in software or with hardware

support. Software transactional memory systems have not been able yet to deliver good performance because of the high overhead created by transactions. Hardware transactional memory systems can reduce the overhead. [Larus and Kozyrakis, 2008]

3.4.2 Communication

Although they can also be used in shared memory environments, communication primitives are primarily relevant in distributed memory settings. Communication primitives can be divided into *basic communication primitives* and *collective communication primitives*.

The simplest possible communication primitive is *point-to-point communication*, which is the basic communication primitive primarily used in MPI. In point-to-point communication one process sends data and the other process actively receives the data and processes it. The difficulty with this approach is that both the sending and the receiving process have to be actively involved in the communication process, so both need to know when the communication has to take place. This can become a problem in programs with irregular communication patterns. In general, developing algorithms using point-to-point communication is difficult and error-prone. *One-sided communication* helps to avoid some problems, like starving receiver processes, by allowing a process to write directly into the memory of another process without requiring involvement of the process on the receiving node. One-sided communication is the primary mode of communication in Partitioned Global Address Space (PGAS) languages and support for one-sided communication has also been added to MPI 2.0. [Bell et al., 2006]

Collective communication primitives represent a kind of synchronization point where a group of processors exchanges data. Examples of collective communication primitives include *gather* and *scatter*, where an array is gathered from or scattered to all nodes, and *broadcasts*, where one message is sent to all nodes. These primitives are generally only relevant for distributed memory setups. *Barriers*, which have already been described as synchronization primitives, can also be seen as a collective communication primitive. Collective communication primitives can often benefit from different, sometimes architecture dependent optimizations. Unfortunately there also exist some collective communication primitives that do not scale well to large numbers of processors, which has been shown in [Balaji et al., 2009].

Although in theory it would be possible to simulate communication primitives in a shared memory environment, this is seldom done in practice. Of higher interest for shared memory environments are *reduction* operations. A reduction operation processes data elements and combines them to a smaller amount of data using a binary operator. Associative and commutative operators are usually used for this kind of operation. Figure 3.4 exemplifies reduction with the summation operator over a sixteen-element array that is distributed to four nodes: First, the array is reduced inside each node. This is usually done by using a sequential algorithm. Then the local sums of the nodes are further reduced until one global sum has been calculated. Reduction between nodes is usually based on a tree-based algorithm as in Figure 3.4, which can be used for associative reduction operators.

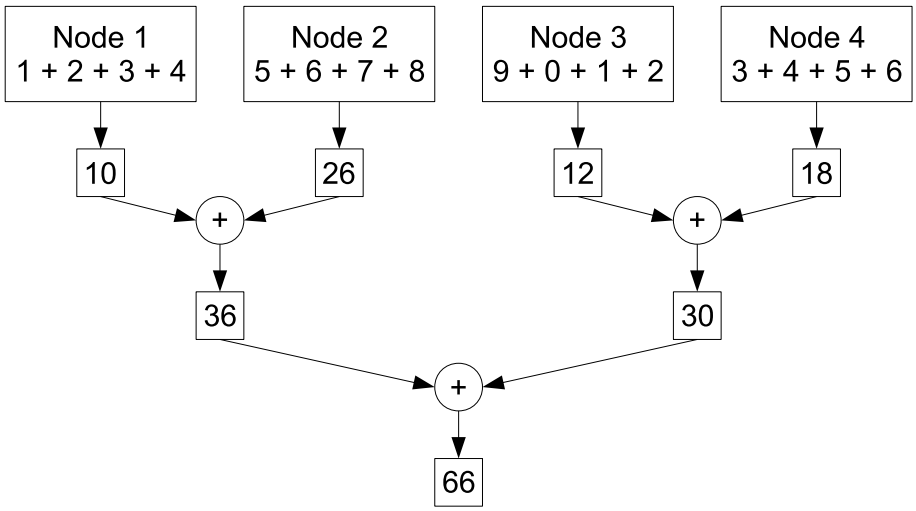


Figure 3.4: Calculating the sum of an array with a reduction operation.

3.5 Views on data

In parallel computing usually a distinction is made between *global* and *local* views on data. A global view is a consistent view of shared data that is accessible by all processors, whereas a local view is a specific processor's view on data. The local view contains variables private to a specific processor and may also contain copies of shared variables that are synchronized with the global view every time a synchronization operation is issued. In shared memory environments, the local view of a global variable typically corresponds to the value that can be found in the caches or registers of a processor. In some distributed memory environments, like the *PGAS* group of programming models, the global view on data is the view of the owner of the variable, whereas the local view is a cached copy at some other processor. Some distributed memory programming models, such as *MPI*, only provide a local view on data.

For arrays, the distinction between global and local views is also often made in distributed memory setups where the array is distributed between nodes. In this case the global view of an array corresponds to the complete array, whereas the local view corresponds to the part of the array that is local to the processor. In programming models where the programmer explicitly operates on the local view of an array, array indices have to be translated between indices in the global view and indices in the local view. Accesses to non-local parts of the array have to be specified explicitly and complicate the code. High-level parallel programming models for distributed memory architectures often try to abstract away the distinction between the global and the local view on an array. The difficulty of this approach is the high complexity of doing this abstraction efficiently.

Chapter 4

Programming Models

One of the major problems in parallel programming is the trade-off between programmer productivity and program performance and scalability. Unlike sequential programming, where high-level abstractions have allowed to increase productivity and where efficient compilers exist, most of parallel computing is still being done through *multithreading* and *message passing*. Programs where performance is not essential are often still written sequentially. Hence, the gap between the performance possible with modern parallel systems and the actual utilization of it is widening. [Hofstee, 2009]

For parallel programming to be adopted in the mainstream, high level programming models are necessary. Still, they need to deliver speedup for most applications, and make sure that applications do not get slower when executed on a higher number of processors. Unfortunately, these goals are still far away.

This chapter describes which established programming models exist and what their limits are. Additionally, it discusses some experimental new high level programming models. Some high-level programming models, namely *OpenMP*, *Cilk++*, *Threading Building Blocks*, *Chapel* and *X10*, are described in more detail and compared based on four criteria. The criteria are support for *data distribution*, support for *work distribution*, the *memory model* and the *coordination primitives* provided by the programming model. These programming models have also been used for the experiments described in the next Chapter.

In the analysis of the *support for data distribution* each programming model is looked at with regard to how data can be split up to be processed in parallel. It is evaluated how different types of data distributions are supported and how the programming model can help to access data in an efficient manner. In [Nikolopoulos et al., 2000] it has been shown that for NUMA architectures a good data distribution between processors does not necessarily have to be implemented on the language level to achieve good performance, because dynamic data-migration strategies can provide similar effects with only little overhead. In the context of this work, a lack of means for data distribution is not seen as a problem for a programming model for shared memory architectures. Still, a good data-layout can reduce problems with cache invalidation due to false sharing that can lead to degrading performance in some applications.

For the overview of the *support for work distribution*, the focus is laid on how work distribution can be realized in each programming model. This can either happen implicitly when the work distribution is derived from the data distribution or explicitly through work distribution primitives like parallel *for*-loops.

The *memory model* is evaluated by looking at how program-memory is presented to the programmer. One major point is the memory consistency model used in the programming model, and how the memory consistency model provided by the hardware is abstracted away. Another point is how global and local views on data are realized, and how the programmer can utilize them.

Finally, the *coordination primitives* provided by each programming model are looked at. In general, coordination in programs can either be *message based*, like in MPI, or *memory based*, e.g. through the use of locks. Some high-level coordination primitives, such as *barriers*, cannot be clearly categorized as message or memory based because they can be built on top of both paradigms with similar complexity.

4.1 Programming models for shared memory architectures

This section gives an overview over programming models for shared memory architectures. All these models have in common that they are not suitable for distributed memory architectures. To utilize modern cluster architectures, shared memory programming models are sometimes used in combination with a distributed memory programming model, where the shared memory programming model is used within every shared memory node.

4.1.1 Multithreading

On modern operating systems, multithreading is supported by default. This means that every process in the operating system can consist of one or more threads of execution, which can run concurrently. Every thread has its own execution stack but shares the address space with all other threads in the same process. The execution of threads is scheduled by the operating system scheduler. On single processor systems, the scheduler tries to give a fair share of CPU time to every running thread, by swapping them at regular intervals. The advantage of this method is that long running tasks can be moved into their own thread and do not block the rest of the program.

On multiprocessor and multi-core architectures threads can be run on all cores or processors, which allows to exploit parallelism by splitting a program into multiple threads. Standard low level threading libraries are better suited for task decomposition than for data decomposition, because a function call can easily be converted to a fork. Unfortunately, to get good performance out of threading, the programmer has to find the optimum granularity of tasks manually.

Threading is often criticised for being very low level, and, in fact, synchronization is a big issue. Write operations on shared data-structures often require complex locking routines, and, while read operations can usually be done concurrently, they cannot be done while writing the

data structure. Locks can be omitted using non-blocking synchronization constructs like the one presented in [Herlihy et al., 2003]. They can sometimes reduce synchronization overhead, but can become very difficult to implement.

[Lee, 2006] criticizes multithreading for being non-deterministic by default and requiring the programmer to prune away that non-determinism. Lee argues that instead a programming model should provide deterministic behaviour by default and allow the programmer to explicitly add non-determinism where necessary.

In addition to that, multithreading, especially on parallel architectures where threads really run in parallel, can often lead to effects that create indeterministic behaviour.

Race conditions are one example where the result of an operation is dependent on the timing of the threads as shown in Figure 4.1. Depending on the order of the operations on x, the result will either be 24 or 22, and when the operations on x are not atomic, which is usually the case, the result can also be 12 or 20.

Main program:	thread1:	thread2:
`x = 10`	`\\ do something`	`\\ do something`
	`x = x + 2`	`x = x * 2`
`fork thread1()`		
`fork thread2()`		
`print x`		

Figure 4.1: Race condition example

Deadlocks are another common problem that appears in multithreading. It can be best explained with the dining philosophers problem described in [Dijkstra, nd]. It is summarized as a situation where five philosophers sit around a table and do one of two things: eating or thinking. Every philosopher has a fork on either side, which he has to share with his neighbour. To be able to eat, a philosopher needs both forks. If we suppose that every philosopher gets hungry at the same time and takes his right fork first, every single of them will end up with only one fork. None of the philosophers will then be able to eat and they will all end up waiting for the other fork. However, as not a single philosopher gives his fork back, they all end up in a deadlock and therefore have to wait forever. Hence, if only one philosopher had got both forks, he would have been able to eat and then give the forks back so that his neighbours could have used them. Deadlocks are difficult to reproduce and difficult to track, which makes multithreaded programming even more complex.

A big problem with low level threading libraries is that they can only provide memory consistency guarantees through the use of synchronization statements. More advanced consistency models that do not solely rely on reordering constraints over synchronization statements can only be realized with the help of the compiler.

Examples of standard low level programming libraries include POSIX threads [Barney, 2009],

which is the standard threading library used in UNIX systems, Windows API threads [Microsoft, 2009] and Java Threads [Java, 2009].

Since Version 5.0, Java also supports some higher level threading constructs like *executors* and *futures*. Executors are used to execute tasks using a given strategy, which is defined by the type of executor that is used. One of the standard executors provided by the Java library is the *ThreadPoolExecutor*, which provides a fixed pool of threads where the given tasks are executed on. This can reduce the overhead of creating threads, as threads are reused after finishing a task. The *ScheduledThreadPoolExecutor* is an enhanced version of the *ThreadPoolExecutor*, where tasks can be executed after a certain amount of time, or execution of a task can be repeated periodically.

Java's *futures* are objects that represent the result of an asynchronous computation. The calling thread can poll whether the future already has a result. When the calling thread tries to read the result, it blocks the calling thread until the result is ready and then returns the result. [Java, 2009]

Moreover, the memory model of Java has been improved to provide a modern model for multithreaded languages. It provides semantics that should be intuitive for well-synchronized programs, and that minimize security hazards for incorrectly synchronized programs. [Lea, 2009, Gosling et al., 2005]

4.1.2 OpenMP

OpenMP has been jointly developed by a group of hardware vendors and compiler developers since 1997. The main idea is to use the *SPMD* model, where all processors execute the same program, but work on different pieces of data. Figure 4.2 shows how this is realized in OpenMP: At program start, a master thread is run which forks off multiple threads as soon as a parallel region is reached.

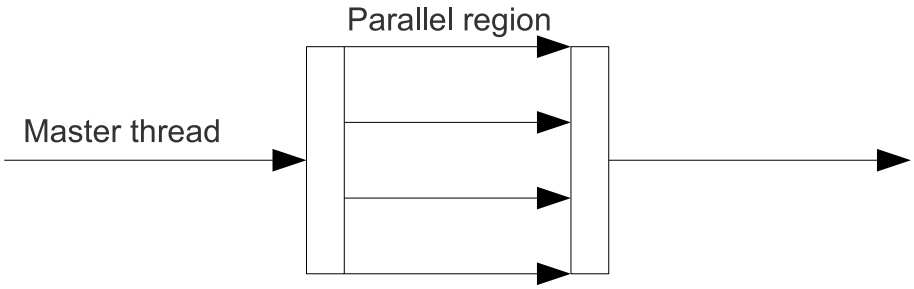


Figure 4.2: OpenMP's execution model.

By adding OpenMP directives before regions which should be parallelized, a sequential C/C++ or Fortran program can be transformed into an OpenMP program. OpenMP directives are implemented as comments so that OpenMP programs can still be compiled with a normal sequential compiler. This non-invasive approach has two advantages. The first one is

that OpenMP allows for converting existing code to OpenMP code easily without destroying the sequential semantics of the program. As long as the algorithm used in the code is suitable for parallel execution, it is often only a matter of adding a small amount of comments to parallelize it. This allows incremental parallelization of existing code, which is very useful for parallelizing existing sequential applications. The second advantage is that the program can still be compiled as a sequential program to simplify debugging. Moreover, when compiling the program as a sequential program code, analysis tools built for sequential programs can still be used.

A major advantage of OpenMP is the wide support in the industry. Most of the major C/C++ and Fortran compilers contain OpenMP support, and there exist quite optimized implementations. IBM even announced a version of its XLC compiler that supports automatic optimization of OpenMP code for the Cell architecture. Some tests with this compiler have also been done in the context of this work to see the potential of this approach.

Support for data distribution

In OpenMP there is no support for data distribution, although extensions to support data distribution have been proposed. [Chandra et al., 1997] The reason why this has never been implemented is that it would introduce additional complexity for the programmer that has only a negligible effect on SMP systems. In fact, it has been shown in [Nikolopoulos et al., 2000] that implementing data distribution on the language level would not bring any performance advantage on NUMA systems compared to *dynamic page migration*. Dynamic page migration, which is implemented at the compiler and runtime level and, therefore, does not introduce additional complexity on the language level, is a technique where a memory page is moved to a specific processor depending on the data access patterns [Nikolopoulos et al., 2000].

Still, with support for data distribution, the programmer would have more control over data-layout. This would allow the programmer to reduce situations where false sharing occurs, and, therefore, to avoid some cases of thrashing.

Although it would be possible in theory, there do not seem to be any plans to adapt OpenMP for distributed memory architectures. Therefore, it is highly unlikely that data distribution directives will be added to OpenMP in the near future.

Support for work distribution

Traditionally, work distribution may be specified in the context of OpenMP *parallel sections* and *parallel loops*. To make OpenMP better suitable for irregular work distribution, the concept of *tasks* was introduced in OpenMP 3.0.

Parallel sections provided by OpenMP are sections of code that are executed in parallel on a fixed number of threads (by default the maximum number of threads available to OpenMP). Figure 4.3 shows an example where each OpenMP thread spawned in the parallel section outputs a message including its unique thread number, which is retrieved using the *omp_get_thread_num* function.

```
#pragma omp parallel
printf("Hello from thread %d!\n", omp_get_thread_num());
```

Figure 4.3: Using parallel sections in OpenMP.

The support for *parallel loops* is a concept which is supported by most of today's high-level parallel programming models. OpenMP allows parallelizing simple linear *for*-loops (or *do* loops in Fortran). The work distribution in OpenMP parallel loops can be influenced using the schedule clause, which allows to specify whether iterations are scheduled dynamically or statically, and how large the chunk-size of the used block-cyclic distribution should be.

The *task* directive introduced in OpenMP 3.0 provides a *fork-join* model of parallelism that launches lightweight tasks to execute a statement in parallel with the containing block of code. Figure 4.4 shows how the *task* directive can be utilized to parallelize the *quicksort* algorithm. The quicksort algorithm is a sorting algorithm that works by choosing a pivot element in the array and moving the contained data so that all data on one side of the pivot element is smaller than the pivot element, and the data on the other side is larger. After the data has been partitioned, the quicksort algorithm can be applied recursively to both array parts separated by the pivot element. As both recursive calls operate on different data, they can be executed in parallel, which is how the algorithm has been parallelized in Figure 4.4. There, the OpenMP *task*-directive is used to create a new task for the first recursive call to the quicksort function so that it can execute in parallel with the second call.

```
void quicksort(int data[], int left, int right) {
    if(right > left) {
        int i = partition(data, left, right)
#pragma omp task
        quicksort(data, left, i-1);
        quicksort(data, i+1, right);
    }
}
```

Figure 4.4: OpenMP implementation of the quicksort algorithm.

Memory model

The memory model of OpenMP is defined as a "relaxed-consistency, shared-memory model" [OpenMP, 2008]. It differentiates between a *global view*, which is accessible by all threads, and a temporary *local view* for each thread, which can be used to represent any kind of intervening structure between thread and memory (e.g. caches). By default, all variables in OpenMP are shared between threads. To allow different behaviour, OpenMP introduces *data-sharing clauses*. When a variable is declared as *private*, each thread gets a copy of the variable that is not synchronized with the other threads. A *shared* variable, on the other hand, is a variable

accessible to all threads of a parallel region. Shared variables follow a relaxed consistency model similar to the *weak ordering* model described in [Adve and Gharachorloo, 1996] with only small differences. In this consistency model, changes to a shared variable may stay in a thread's temporary view until the next synchronization operation, instead of being instantly propagated to the global view. Some OpenMP operations provide implicit synchronization.

The *flush* statement, which takes one or more variables as parameters, can be used to explicitly synchronize the local view of variables in a thread with their global view. During a flush the temporary view of the given variables is discarded. In addition, if some data has been written to a temporary local view of the given variables, it is written back to the global view. The program proceeds after the flush as soon as the write operation has been completed. The flush operation also makes sure that accesses to the given variables are not reordered in respect to the flush operation. Synchronization operations like the *barrier* statement and locks contain implicit flushes to ensure consistency.

OpenMP also supports other types of data sharing attribute clauses. They are special cases of private variables that only differ in how the variable is initialized and what happens to the variable and data after the end of the parallel region. For variables used in reductions, OpenMP provides the *reduction* clause. With it the programmer can define a variable that works as a private variable throughout a parallel region. After the end of the parallel region the values of all local instances of the variable are reduced to a global value by a specified operator. Figure 4.5 shows an example of how to calculate the sum of an array with OpenMP. In this example, the variable *sum* is used for reduction with the "+"-operator. First, each thread calculates the sum of its own piece of the array and stores it in its local view of *sum*. After the loop is finished, all local views of *sum* are added up by OpenMP and stored in the global view of sum, which is then accessed in the following *printf*-statement.

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i=0; i<n ; i++)
    sum += data[i];

printf("The sum is: %d", sum);
```

Figure 4.5: Calculating the sum of an array with OpenMP reduction.

Coordination primitives

The primary means of coordination in OpenMP are *critical* sections and *barriers*. Declaring a statement block as a *critical section*, ensures that it can only be executed by one thread at a time. Inside parallel loops an *ordered* section can be used to ensure that a block of code is executed in sequential loop order. A *barrier* statement blocks the execution until all threads executing a parallel section or loop have reached the barrier. To ensure that all spawned asynchronous tasks

have terminated before resuming execution, the *taskwait* directive can be used.

In addition to the synchronization constructs, OpenMP provides atomic statements, which provide atomic execution of mathematical operations on a variable.

4.1.3 Cilk/Cilk++

Cilk is an extension to the C programming language and a corresponding runtime, specialized on simple parallel programming under C. It was developed by a research group at the MIT and is mainly based on task parallelism. Creating a parallel task in Cilk simply works by using the *spawn* directive when calling a function. A function called with spawn can run in parallel with the caller. Similar to Threading Building Blocks, Cilk manages threads in pools and maps tasks onto threads.

To make sure all threads are utilized, Cilk employs a *work-stealing* scheduler. In a work-stealing scheduler, each thread has its own task queue that it processes. New tasks are added to the queue of the parent's thread. As soon as one thread runs out of work, either because the queue is empty or because all tasks are blocked, a thread tries to "steal" work from another thread. [Blumofe and Leiserson, 1994] describes work-stealing in more detail.

A lot of research has been done in Cilk to create an efficient work scheduler. For example, it uses the information it has about the parent-child relationship between tasks to create a task-graph. This graph can then be used by the work-stealing scheduler to make better decisions, like stealing shallow tasks in a graph first, as they are more likely to spawn other tasks.

Cilk has inspired the development of *Cilk++* by a commercial company named Cilk Arts, which has been bought by Intel in July 2009. It employs the techniques used in Cilk in a C++ setting and provides additional tools for verification and testing like a race condition tester. [Cilk, 2009b, Leiserson and Mirman, 2008] For the purpose of this work the focus has been laid on Cilk++, but most findings should also apply to Cilk.

Support for data distribution

Similar to OpenMP, Cilk++ does not aim to provide support for data distribution. As Cilk++ is a programming model primarily aimed at shared memory architectures as well, the same findings should apply as for OpenMP.

Support for work distribution

The programming model of Cilk++ is mostly focused around its *spawn* statement, which creates tasks that are executed in parallel with the calling code. A task in Cilk++ roughly corresponds to a task in OpenMP 3.0 and is, therefore, a good candidate for implementing divide and conquer algorithms like *quicksort*. Figure 4.6 depicts a Cilk++ implementation of the quicksort algorithm. This example differs only little from the example in Figure 4.4 which shows an OpenMP implementation of the same algorithm. In this example, the *cilk_spawn* directive is

used to create a new task executing the first recursive call to the *quicksort* method. The call to *cilk_sync* blocks further execution until all tasks spawned inside the function have completed.

```
void quicksort(int data[], int left, int right) {
    if(right > left) {
        int i = partition(data, left, right)
        cilk_spawn quicksort(data, left, i-1);
        quicksort(data, i+1, right);
        cilk_sync;
    }
}
```

Figure 4.6: Cilk++ implementation of the quicksort algorithm.

In addition to the *spawn* directive, Cilk++ also provides parallel for-loops with the *cilk_for* directive. This directive is used in Figure 4.7 to implement a matrix multiplication kernel. The only difference to a standard sequential matrix multiplication kernel is that the outer loop is executed in parallel.

```
cilk_for(int i = 0; i < N; i++)
    for(int j = 0; j < N; j++)
        for(int k = 0; k < N; k++)
            C[i*N + j] += A[i*N + k] * B[k*N + j];
```

Figure 4.7: Matrix multiplication in Cilk++.

The power of Cilk's work distribution concepts comes from the optimizations for task creation and scheduling employed by Cilk's compiler and scheduler. These optimizations ensure that overhead for creating tasks is small and that Cilk++ provides good scalability for large numbers of spawned tasks [Frigo et al., 1998]. In fact, some of Cilk's concepts have already been taken up by other programming models (e.g. TBB uses Cilk's work-stealing scheduler).

Memory model

Unlike the memory consistency models used in most programming models where consistency is defined in terms of actions by physical processors, Cilk's memory consistency model is based on *dag consistency*. Dag consistency is a relaxed consistency model based on the dag spanned by the user-level threads spawned by a computation. In dag consistency a read operation can only see a write operation if there exists a sequential execution order consistent with the dag in which the read operation sees the write operation. Dag consistency is described in more detail in [Blumofe et al., 1996].

Further research in this direction was done in [Frigo, 1998]. In his work Frigo, who was also involved in the development of dag consistency, tries to identify the "weakest reasonable memory model". The following properties were defined by Frigo as being necessary for a weak memory

consistency model to remain "reasonable": *Completeness*, *Monotonicity*, *Constructibility*, *Nondeterminism confinement*, *Classicality*. Interestingly, dag consistency violates the constructibility property, which means that the actual memory consistency model that can be implemented is stronger than the theoretical model of dag consistency. Frigo proves in [Frigo, 1998] and [Frigo, 1999] that another consistency model, which he calls *location consistency*, is a constructible version of dag consistency, which is identical to the actual implementation of Cilk's memory consistency model.

Cilk provides a *fence* statement which makes sure that statements are not reordered over the fence and that all read and write operations are completed before the fence. Unfortunately, unlike the OpenMP statement, the Cilk programming model does not allow for specification of a set of variables for which this fence applies. Instead, this fence applies to all variables, which makes the Cilk implementation less flexible.

It is worth pointing out that the fence statement does not seem to be supported in Cilk++. Instead, Cilk++ introduces the concept of *hyperobjects*. Hyperobjects are objects that allow every task to have its own view of itself. Therefore, the access to a hyperobject inside a task is consistent. After spawned tasks have completed, changes in different views of a hyperobject are merged back into one object. Currently, there only exists one type of hyperobjects, namely the reducer, which allows the reduction of a value with associative operations. It has yet to be shown by the Cilk++ developers that hyperobjects can be useful for use cases other than reductions. Except for these changes, the memory consistency model of Cilk++ seems to be similar to the one of Cilk.

To make the work with the memory consistency model used in Cilk++ easier, the company behind Cilk++ offers an analysis tool that detects *data races* (but not every *general race*) to allow the developer to verify his program. According to their documentation, it is guaranteed to detect every *data race* in a program. This tool is contained in the commercial version of Cilk++. [Leiserson and Mirman, 2008, Cilk, 2009a]

Coordination primitives

As Cilk++ mainly focuses on fork-join parallelism, the most relevant coordination primitive is the *cilk_sync* statement, which can be seen in Figure 4.6. It blocks execution of the current function until all tasks spawned by the function have terminated. Synchronization is also done implicitly at the end of each function which means that the example in Figure 4.6 should also be correct without the *cilk_sync* statement. Like most programming models Cilk++ provides support for mutexes. Cilk++'s mutexes are kept very simple and support only locking and unlocking. [Supercomputing, 2001, Frigo et al., 1998, Randall, 1998]

As an additional coordination construct Cilk++ provides *hyperobjects*. A hyperobject is defined as being an object that provides a local view on a variable for each thread and that can merge its local views back into one global view. In its current implementation Cilk++ only provides hyperobjects for reduction operations. Only little information can be found about the general concepts behind hyperobjects and it is difficult to say whether hyperobjects may be

useful for tasks other than reductions.

4.1.4 Threading Building Blocks

Threading Building Blocks (*TBB*) [Willhalm and Popovici, 2008, Reinders, 2007] is a C++ library created by Intel that tries to address common issues in multithreaded programming without reducing the programmers flexibility. It is designed to be used as a normal C++ library, and it utilizes template metaprogramming to implement its functionality.

TBB utilizes a thread pool that contains as many threads as there are virtual processors on the machine. Each processor has its own task queue, but it can do work-stealing to make sure its queues do not run empty. Having its own scheduler allows TBB to use the additional information it has about its tasks to optimize the scheduling for data locality. Another advantage is that tasks are more lightweight than system threads.

In addition to its scheduler and its parallelization primitives, TBB also provides a library of parallelized replacements of some standard C/C++ and STL classes and functions like concurrent collections and memory allocators.

Support for data distribution

As Threading Building Blocks (*TBB*) is a programming model for shared memory architectures, there is only little support for actual data distribution. Still, the focus of TBB's programming model lies on data parallel programming, and, therefore, it provides at least some means to optimize memory access patterns for cache locality. Specifically, it is possible to implement different *range* classes for the use in data-parallel operations like *parallel_for* and *parallel_scan*. A range is an object that represents a part of the array domain used in a parallel operation. When starting a parallel operation, the programmer passes a range object representing the whole array domain to the function executing the operation. TBB then uses a partitioner object (which can also be implemented by the programmer) to partition the range into smaller subranges, which are then passed to the computational kernel. The functionality of the range object can be used to optimize memory access patterns and caching behaviour for data-parallel applications.

Figure 4.8 depicts how TBB's ranges work. In this example a blocked range from 0 to n is passed to a parallel kernel (e.g. *parallel_for*). TBB internally splits the range up into a range from 0 to $\frac{n}{2}$ and a range from $\frac{n}{2}$ to n. Each subrange is then passed to the sequential kernel provided by the programmer.

Support for work distribution

TBB provides a wide variety of operations that can be used for work distribution. The flexible *range* class, which has already been described above, allows to flexibly configure how work is distributed in parallel algorithms. TBB also provides a *task* class that allows to spawn lightweight tasks.

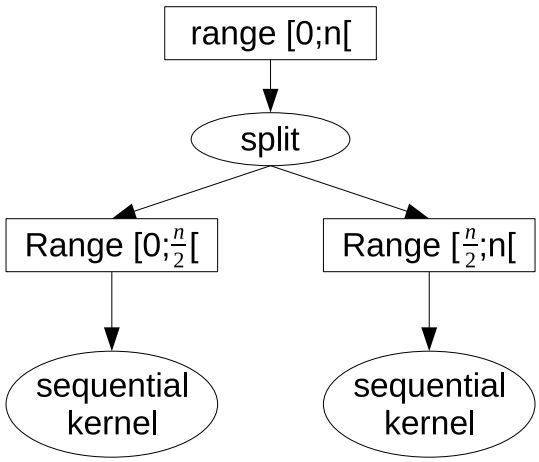


Figure 4.8: How TBB's ranges work.

Figure 4.9 shows how *tasks* can be used in TBB. In this example, a root task is started in the main program that recursively creates asynchronous tasks and waits for them to finish. The actual code of the task is found in the *execute* method of the *MyTask* class.

```
class MyTask: public task {
public:
    task* execute() {
        // Do computation
        if(recurse) {
            MyTask t1 = *new(allocate_child()) MyTask();
            MyTask t2 = *new(allocate_child()) MyTask();
            spawn(t1);
            spawn_and_wait_for_all(t2);
        }
        return NULL;
    }
};

int main(void) {
    task_scheduler_init init(NUM_THREADS);
    MyTask& t = *new(task::allocate_root()) MyTask();
    task::spawn_root_and_wait(t);
}
```

Figure 4.9: How to use tasks in TBB.

Like most high-level programming models, TBB also provides a parallel *for*-loop. Parallel for-loops utilize the *range* class, which has been described in the previous section, to determine the work distribution. The range object and a function object containing a sequential kernel that can be applied to a subrange of the given range are passed as parameters to the *parallel_for*

function to start a parallel iteration. TBB also uses range objects for the *parallel_reduce* and *parallel_scan* methods, which can be used to do a reduction or scan on data.

The flexibility of *parallel_for*-loops in combination with *range* objects is revealed when they are used to implement divide and conquer algorithms like the *quicksort* algorithm. The quicksort algorithm can be implemented in TBB by using the range object as a partitioner for the array to sort. This approach maps well to TBB's range objects as they are always partitioned using a divide and conquer approach (see Figure 4.8). During partitioning the range object is free to manipulate the array the range operates on, as it is guaranteed that the region it is operating on is not operated on by other range objects.

For problems where the amount of work cannot be determined at the beginning, the *parallel_while* template class can be used. In addition to a computational kernel, it requires an input stream as parameter which returns the next item to calculate. The disadvantage of this approach is that operations on the input stream are inherently sequential, which may limit achievable parallelism.

TBB also provides a *pipeline* class that can be used to implement pipelined algorithms. The native support for pipelines in TBB makes implementing pipelined algorithms an easy task. Unfortunately, TBB does only support linear pipelines, so that pipeline stages that can theoretically be executed in parallel either have to be linearized or have to be implemented in one pipeline stage. According to [Reinders, 2007, p.83], the pipeline implementation provided by TBB performs better compared to manual pipeline implementations using the *concurrent_queue* class provided by TBB because TBB pipelines are optimized for cache locality. Figure 4.10 shows an example code for the implementation of a pipeline. To create a pipeline step, the programmer has to implement a class derived from TBB's *filter* class and to override the *()* operator. In this example the class *MyFilter* implements a pipeline step. Inside the method *runPipeline* the actual pipeline is created, and an instance of the *MyFilter* class is added as a filter to the pipeline. The *run* method of the pipeline object starts pipeline execution and blocks until it is complete. Finally, the memory used by the pipeline is cleared using the *clear* method.

Memory model

As TBB is implemented as a C++ library and not a compiler, it cannot influence the memory consistency model of the complete application. For that reason, memory consistency for normal operations is highly dependent on the consistency model provided by the compiler and the target processor architecture. To provide consistent access, TBB contains the *atomic* template class, which provides five basic statements, namely *read*, *write*, *fetch and store*, *fetch and add* and *compare and swap*. These statements are guaranteed to be executed atomically on the given template data-type, which may be an integral or pointer type. To provide consistency between atomic statements, atomic statements provide *release consistency*. In this consistency model two operations exist: *acquire* and *release*. An *acquire* makes sure that operations following this statement are not reordered over it. This is the default behaviour for atomic *read* operations. A *release*, on the other hand, makes sure that operations before the statement are not moved over

```
class MyFilter: public tbb::filter {
    // ...
    void* operator() (void* item);
};
void* MyFilter::operator() (void* item) {
    // Do something with item
    return output;
}

void runPipeline() {
    tbb::pipeline pipeline;
    MyFilter filter(/* ... */);
    pipeline.add_filter(filter);
    // Add more filters
    pipeline.run( input );
    pipeline.clear();
}
```

Figure 4.10: Using TBB's pipeline class to implement a pipeline.

it. This is the default behaviour for atomic *write* operations. All other atomic operations are by default handled as both *acquire* and *release* operations, but TBB allows to explicitly specify weaker consistency for them.

Coordination primitives

TBB focuses on data-parallel programming and tries to keep most coordination on the algorithm design and library level. This means that for common problems no explicit coordination should be necessary. TBB also provides concurrent collection objects with interfaces mostly similar to their C++ *Standard Template Library* (*STL*) counterparts, which can be used for high-level coordination (e.g. a concurrent queue can be used as a message queue). For cases where explicit coordination is necessary, TBB provides various types of mutexes (*mutex*, *spin_mutex*, *queuing_mutex*, *spin_rw_mutex* and *queuing_rw_mutex*) which differ in their characteristics. The *spin_rw_mutex* for example, which is used in Figure 4.11 to demonstrate the lock syntax of TBB, provides a read-write lock with good performance, but is *unfair*, which means that it does not avoid starving tasks.

The *atomic* template class in TBB provides the programmer with variables that support various atomic operations. In Figure 4.12, the atomic template class is used to create an atomic integer variable. The atomic operation *fetch_and_add*, which adds a value to the variable and returns its previous value, is then used on the atomic variable. One speciality of TBB's atomic operations is that they follow a *release consistency* memory consistency model. As TBB does not provide memory consistency for normal variables, atomic variables are necessary to write memory-consistent code.

```
spin_rw_mutex my_lock;

void critical() {
    spin_rw_mutex::scoped_lock = lock(my_lock, /*is_writer=*/true);
    // write something
}
```

Figure 4.11: Using TBB's mutexes.

```
atomic<int> x = 5;
// ...
int y = x.fetch_and_add(3);
// x = 8 and y = 5
```

Figure 4.12: Using TBB's atomic template class

4.2 Models for distributed and cluster architectures

One of the major drawbacks of shared memory architectures is that they do not scale well to a high number of processors. Therefore, to build machines with a large number of processors the shared memory abstraction has to be given up, which leads to more complex programming models. The distributed memory programming models presented in this section are able to handle this complexity. Nonetheless, they can also be used on shared memory architectures.

4.2.1 Message passing

Message passing is a programming model for distributed memory architectures, which is often used in an *SPMD* (single program multiple data) setting. In the SPMD paradigm, which was first described by M. J. Flynn in 1972 [Flynn, 1972], multiple processes execute the same program, but operate on different data. Every *MPI* program is designed to have multiple instances of itself running in parallel on different processors. Each process has its own local memory and communicates with other instances by sending messages to them. To be able to distinguish between those processes, a unique id is assigned to each of them.

Depending on the architecture and the MPI implementation, communication is realized in different ways. In networks a message is typically implemented as a network package that is stored in a receive buffer on receiving, whereas on shared memory architectures, communication can be internally handled by shared data queues.

Programs using message passing are even more difficult to program than multithreaded programs as all the communication between nodes has to be done explicitly. A lot of care has to be taken to maximize the communication throughput by grouping communication tasks where it is possible. This is the reason why message passing is often called the "assembler of parallel programming".

Interestingly, a program written in MPI and running on shared memory architectures does

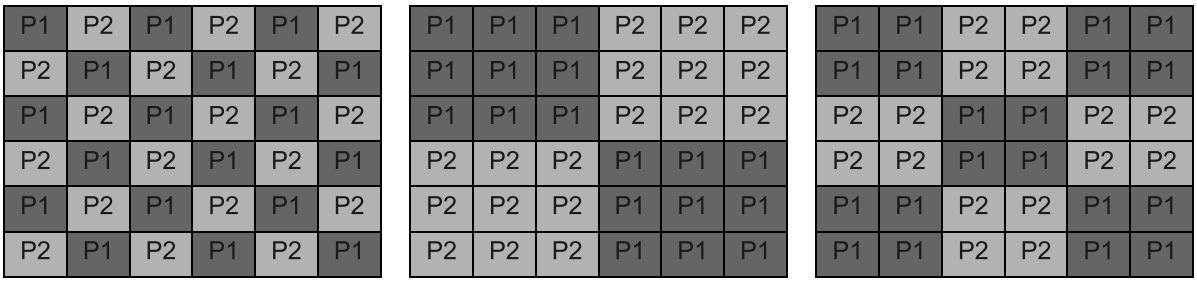
not necessarily have to be slower than its multithreaded counterpart. In fact, MPI programs can be faster as well. The overhead created by message passing can be outweighed by the advantage of explicitly expressing locality of a process. Making the communication between processes explicit also reduces problems with *false sharing*.

Nonetheless, on modern cluster architectures, such as SMP clusters, MPI is often combined with threading libraries. Most common is the combination with OpenMP, where MPI is used on the network level and OpenMP within every shared memory node [Quinn, 2003, p436ff]. Using a different programming model for each hierarchy level of the hybrid architecture makes it easier to do specific optimizations per hierarchy level.

4.2.2 HPF

High Performance Fortran (*HPF*) [HPF, 1997, Loveman, 1993, Benkner and Zima, 1999] is an extension to the Fortran 90 standard aimed at creating a high-level high performance programming language for scientific codes based on a data parallel programming model. HPF is suitable for a wide variety of architectures, but focusses mainly on distributed memory architectures. The first version of the HPF specification was released by the *High Performance Fortran Forum* in 1993. In 1997 version 2.0 was released.

As in OpenMP, HPF directives are comments, which means that an HPF program can still be compiled as a sequential program with a normal Fortran compiler. The core feature of HPF is that for every array that is declared in the program a distribution can be specified. Some 2-dimensional standard distributions are depicted in Figure 4.13.



(a) cyclic distribution (b) block distribution (c) block-cyclic distribution

Figure 4.13: Different types of 2-dimensional data distribution onto two nodes.

The compiler tries to transform the sequential Fortran code to MPI code where the owner of every memory region does the calculations which are based on the memory region. This behaviour is called the *owner computes rule*. For accesses to memory regions from other processors than the owner, the compiler automatically creates the necessary MPI calls to retrieve the data. To increase efficiency, it tries merging remote memory accesses to reduce the overhead created by the calls and to hide latency. The programmer can use the *INDEPENDENT* directive to explicitly specify that there are no data dependencies between loop iterations. Independent loops can be automatically parallelized by the HPF compiler. Many HPF compilers can actu-

ally automatically detect independent loops, but not all independent loops can be recognized by the compiler. [Loveman, 1993]

Although the HPF compilers work well for some problems, achieving good performance is often difficult. Especially irregular problems lead to difficulties, and some research has been done to extend HPF to support irregular problems [Benkner, 1999]. One major problem with HPF is that it is difficult to build up a performance model of an HPF application. In fact, the programmer still has to think about code optimization techniques, however, it is much more difficult to apply them. During migration of existing kernels to HPF, optimization might become necessary because in some cases the communication overhead created by HPF might exceed the gains from parallelization, so that an application actually slows down when executed on a parallel machine. Additionally, the need to optimize for a specific compiler has reduced the portability of HPF. [Kennedy et al., 2007, Mehrotra et al., 2002] Some of the mentioned problems can also be found in OpenMP, but on distributed memory architectures they more often lead to degrading performance.

After some initial enthusiasm, the interest for HPF faded in the late 1990's. In [Kennedy et al., 2007] four main reasons therefore were identified:

> "(1) inadequate compiler technology, combined with a lack of patience in the HPC community; (2) insufficient support for important features that would make the language suitable for a broad range of problems; (3) the inconsistency of implementations, which made it hard for a user to achieve portable performance; and (4) the complex relationship between program and performance, which made performance problems difficult to identify and eliminate." [Kennedy et al., 2007]

The problems with HPF and the limitations imposed by Fortran 90 are the reason that nowadays HPF is mostly interesting for theoretical reasons. Still, some of the features of HPF were actually included into the Fortran 95 standard. Some concepts developed in HPF can now be found in newer high-level programming languages like *Chapel* and *X10*.

4.2.3 Partitioned Global Address Space (PGAS) languages

After the decrease of the popularity of HPF, a new distributed programming model came up and was adopted in several new programming languages. This programming model is called *Partitioned Global Address Space* (*PGAS*). In PGAS, a virtual shared memory exists that is split up in several parts, where each part is local to a specific processor. A schematic view of the PGAS memory model can be seen in Figure 4.14. It depicts an address space that consists of the local memories of multiple processors. Each processor provides a shared memory area that can be accessed by all processors, but also has a private area that can only be accessed locally. The difference to HPF's model is that the programmer explicitly specifies in which memory location to store which piece of data.

The abstraction of a virtual shared memory makes programming in a PGAS easier compared to MPI, and it still allows to optimize for locality. As all memory locations in the address

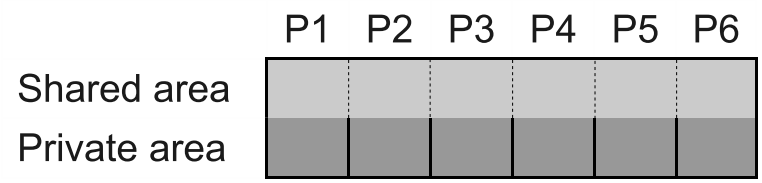


Figure 4.14: The PGAS memory model.

space have a unique address, it is possible to use *one-sided communication*. In one sided communication, the process in the target system does not have to actively receive and process messages but instead communication is only initiated by one side. (See Section 3.4.2 for more details.) On some systems this communication even can be handled by *Remote Direct Memory Accesses* (*RDMA*), which allows the network interface to directly service the memory access so that the remote processor is not involved in the communication. Because of one-sided communication, programs in PGAS languages may sometimes be faster than their MPI counterparts, which are traditionally programmed using a *two-sided communication* pattern [Yelick et al., 2007]. Actually since version 2.0, MPI has also supported one-sided communication, which is, however, handled on a lower abstraction level than in PGAS languages.

Compared to HPF, the programming model behind PGAS languages is more low-level. The programmer has to explicitly specify where data is stored. Also, no concept of data distributions exists in the standard PGAS languages, which means that the programmer has to manually distribute data. This low-level approach allows the programmer to have more influence on the actual program performance while still maintaining a higher productivity than with MPI code. Moreover, building a PGAS compiler is less complex than building an HPF compiler. To ease the development of PGAS languages, low-level libraries like *GASNet* [GASNet, 2009] and *ARMCI* [ARMCI, 2009], which provide a communication layer for remote memory accesses, have been implemented.

A wide variety of programming languages have been based on the PGAS model. *Unified Parallel C* (*UPC*) [UPC, 2009, UPC, 2005], *Co-Array Fortran* [Numrich and Reid, 1998, Co-Array, 2009] and *Titanium* [Hilfinger et al., 2001, Titanium, 2009] are all based on an existing programming language (UPC on C, Co-Array Fortran on Fortran 95, Titanium on Java 1.4) and combine it with an SPMD programming model. Newer programming languages like *Chapel* and *X10* utilize PGAS libraries at their core but provide a higher-level view on the data.

4.2.4 X10

X10 is an experimental programming language that has been developed by IBM in the DARPA program on High Productivity Computer Systems since 2004. It is based on the serial subset of the Java language with added concepts for parallelism and a PGAS. Four main goals were formulated for X10: *safety*, *analyzability*, *scalability* and *flexibility*. *Safety* means that the programming model should be safe from typical problems in low-level programming languages, like

invalid pointer references and type errors. Furthermore, it should guarantee to preserve determinacy and avoid deadlocks for common usage patterns. *Analyzability* requires programs written in X10 to be analyzable by programs. This can be achieved best with a programming language that features simple programming constructs and strong data-encapsulation and orthogonality properties. *Scalability* is, of course, a major goal for any programming model aimed at parallel systems. *Flexibility* shows the need for a programming language to support different design patterns for parallelism like task parallelism and data parallelism. The lack of flexibility is a major drawback of the SPMD programming paradigm used by some programming models.

X10 tries to achieve the given goals by using Java, which satisfies the given criteria for sequential programming. The use of PGAS provides a flexible memory model that allows to maintain good scalability without making the programming model too complex. Asynchronous activities, which are comparable to OpenMP tasks, allow to parallelize the program in a flexible way. Tasks can be mapped to a specific *place*, and a place corresponds to a physical computation node. X10 replaces Java arrays by its own array type. Each array is associated with a region specifying which indices are contained in the array. Additionally, a distribution can be specified so that different array regions can be mapped to different places. Reduction operations and scans can be called for arrays and are then automatically applied over all places. Arrays can be iterated over with the *for*, *foreach* and *ateach* constructs. The *for* construct iterates sequentially over all points in a region. The *foreach* construct spawns activities in the same place for every iteration. The *ateach* construct spawns activities for each point in the region and every iteration is executed in the place specified by the distribution. [Saraswat and Nystrom, 2008,Charles et al., 2005]

Currently, there exist two official implementations of the X10 language. One translates X10 code into Java code that can be used on an SMP machine, whereas the other is based on C++ and contains a PGAS runtime allowing it to run on distributed systems. Both implementations can be found at [X10, 2009].

Support for data distribution

For arrays X10 provides the concepts of *regions* and *distributions*, which can be compared with Chapel's *domain* and *distribution* concepts. A *region* maps an index space consisting of points, which are n-dimensional tupels of integers, to memory locations. X10 provides various types of regions apart from standard n-dimensional arithmetic regions, such as, for example, regions representing a banded matrix. A *distribution* is used in X10 to map array indices to different *places*. A place represents a node in a distributed memory setup. X10 provides some typical distributions, like block and cyclic distributions, and also an indirect distribution that uses an array as a map.

In X10 regions as well as distributions allow different operations like doing an intersection or union between two regions or distributions. This is a feature unique to X10 as this is not possible with Chapel's domains and distributions. The disadvantage of this concept is that X10's regions and distributions are more restricted as they need to allow static reasoning, which might not be

possible for some irregular data structures.

At the time of writing no concepts for user-defined regions and distributions have been released for X10 yet, but, according to the specification, future versions might support them in a way that supports static reasoning [Saraswat and Nystrom, 2008, p122].

Support for work distribution

The smallest unit of parallel execution in X10 is an *activity*. An activity can be spawned locally or in a remote place and is executed in parallel to the spawning task. Each activity except for the *root activity* has a parent activity, and X10 even propagates exceptions from child activities to their parents. This means that, unlike some other parallel programming models, exceptions cannot get lost over activity boundaries. X10's scheduler is based on the concepts introduced with the work-stealing scheduler of Cilk, so it should exhibit similar characteristics.

```
async (SOMEPLACE) {
    System.out.println("Async hello world!");
}
```

Figure 4.15: Spawning an activity in X10.

In addition to activities, X10 provides the *foreach* and *ateach* statements. Both statements start a parallel iteration with the difference being the place the iterations are executed on. In the foreach statement the loop body is executed in the place local to the point returned by the iteration. The ateach statement, on the other hand, requires a distribution as a parameter and executes each iteration in the place specified by the distribution for the current index.

X10 also allows to synchronously execute statements in another place with the *at* statement. Code following an *at*-block will only execute after the *at*-block has terminated. (See Figure 4.17.)

```
ateach( point[i] : dist.factory.unique() ) {
    System.out.println("Hello World: " + i);
}
```

Figure 4.16: Iterating through a distribution in X10 with the *ateach* statement.

```
at (SOMEPLACE) {
    System.out.println("Hello world from another place!");
}
System.out.println("This will always be shown last.");
```

Figure 4.17: Launching a synchronous activity in another place in X10 with the *at* statement.

Memory model

X10's memory model is based on the *partitioned global address space* (*PGAS*) model. The traditional PGAS model is extended by X10 to be *globally asynchronous, locally synchronous* (*GALS*), which means that non-local memory accesses implicitly launch an asynchronous activity local to the memory location.

The memory consistency model used in X10 is based on research presented in [Saraswat et al., 2007]. It guarantees sequential consistency for programs whose sequentially consistent executions do not contain race conditions. This means that it allows code transformations as long as only programs with race conditions can distinguish between them. X10 requires the programmer to use the high-level coordination primitives, such as *finish* and *atomic* blocks, to ensure race-free programs.

For remote memory accesses the memory consistency model does not apply because they are always done using asynchronous activities.

Coordination primitives

X10 provides some high-level coordination primitives that allow programmers to create parallel programs that are guaranteed to be deadlock-free. The coordination primitives introduced by X10 are the *finish* directive, *clocks* and *atomic* blocks.

The *finish* directive ensures that the following statement or statement block terminates after all tasks spawned inside the statement or statement block have terminated. Figure 4.18 shows how the quicksort algorithm can be implemented in X10 using asynchronous tasks and the finish directive. In this example, both recursive calls to the quicksort function run in parallel. The finish directive ensures that the quicksort function terminates only after both recursive calls have finished.

```
public def quicksort(val A:Array[double], val left: int, val right: int) {
    if(right > left) {
        val i = partition(A, left, right);
        finish {
            async quicksort(A, left, i - 1);
            quicksort(A, i + 1, right);
        }
    }
}
```

Figure 4.18: A simple X10 implementation of the quicksort algorithm.

With *clocks* X10 provides a unique concept that can best be compared to barriers. Clocks allow to split the execution of asynchronous tasks into multiple phases where each phase is ended by a task with the blocking *next* statement. As soon as all tasks registered for a clock have executed the *next* statement, the next phase of the clock is started and execution resumes.

Figure 4.19 shows an example of how clocks can be used in X10. In this example, a clock is used to synchronize the output of status messages. In the first phase, both tasks output "Phase 1". The spawned task waits then for the first phase to be completed with the *next* statement, whereas the main task signals the clock with the call to the *resume* method that it is ready to advance to the next phase and then resumes execution to output "After phase 1" before executing *next* to finally advance to the second phase. During execution of the second phase, the main task unregisters from the clock with the *drop* method to ensure the other task does not wait for this task to advance to the next stage.

```
c : Clock = new Clock();

async clocked(c) {
    System.out.println("Phase 1");
    next;
    System.out.println("Phase 2");
    next;
    System.out.println("Phase 3");
}

System.out.println("Phase 1");
c.resume();
System.out.println("After phase 1");
next;
System.out.println("Phase 2");
c.drop();
System.out.println("Not using the clock any more");
```

Figure 4.19: Using clocks to synchronize task execution.

X10 places restrictions on clocks that ensure that programs using clocks can never deadlock. Tasks can only be registered with a clock at creation time, and the parent task has to be registered for this clock as well. The blocking *next* statement always advances every clock the task is registered on, whereas the more fine-grained *resume* method, which is clock-specific, is non-blocking so that it is not possible for two tasks to deadlock because they block on different clocks. Finally, a task is automatically deregistered from a clock as soon as the task terminates.

X10 provides two types of *atomic blocks*: *conditional-* and *unconditional atomic blocks*. *Unconditional atomic blocks*, which are introduced in X10 using the *atomic* statement, provide transactional semantics for the statements executed inside them. X10 places some restrictions on atomic blocks as they are not allowed to contain blocking statements, they are not allowed to spawn asynchronous tasks and they have to access only local memory locations. Also, atomicity is not guaranteed for statements that throw exceptions so that the programmer has to catch the exceptions and provide undo-code to ensure atomicity.

Conditional atomic blocks, which can be used in X10 with the *when* statement, are similar to *unconditional atomic blocks* with the difference that the execution of the statement blocks

until a certain condition, which is evaluated atomically, holds true. X10 allows the programmer to provide two or more alternative conditional atomic blocks.

4.2.5 Chapel

Similar to X10, *Chapel* is a language originating from the DARPA program on High Productivity Computer Systems. [Chapel, 2009a, Chapel, 2009b, Callahan et al., 2004, Chamberlain et al., 2007] It has been developed by Cray Inc. since 2004. The design of this language is tailored mainly for the purposes of scientific computing where most algorithms operate on arrays.

One of the major goals in the design of Chapel was to provide support for flexible data distributions. The experience gained in HPF development [Mehrotra et al., 2002] showed that for a programming model to be widely accepted, it has to be able to support arbitrary data distributions. For this reason, Chapel provides means to implement custom array domains and data distributions with an object oriented interface.

One of advantages of Chapel is its expressiveness for array operations. Arrays can be sliced by using a *range* in the index field. *Scalar promotion* allows to specify arrays or functions returning arrays as a parameter to functions that accept scalar values. In these cases, the function is called for every array element and an array containing the results is returned. Reductions can be applied to every array with the *reduce* keyword.

The expressiveness of Chapel's array statements can be seen in Figure 4.20. It depicts a Jacobi iteration written in Chapel. In it, a point in the *XNew* matrix is calculated by calculating the average of its neighbours in the old matrix. This is done by using a forall iteration iterating through the complete problem space. After this, the maximum difference between the old and the new matrix is calculated by using an array statement to calculate the difference and a maximum reduction operation to get the maximum. It is worth noting that scalar promotion is used to calculate the absolute value of each element in the difference matrix using the *abs* function. At the end, the new matrix is assigned to the old matrix using a simple array statement.

```
const north = (-1,0), south = (1,0), east = (0,1), west = (0,-1);

do {
    // compute next approximation using Jacobi method and store in XNew
    forall ij in ProblemSpace do
      XNew(ij) = (X(ij+north) + X(ij+south) + X(ij+east) + X(ij+west)) / 4.0;

    // compute difference between next and current approximations
    delta = max reduce abs(XNew[ProblemSpace] - X[ProblemSpace]);

    // update X with next approximation
    X[ProblemSpace] = XNew[ProblemSpace];
} while (delta > epsilon);
```

Figure 4.20: Shortened version of the Jacobi sample program delivered with the Chapel compiler.

The current implementation of Chapel supports the use of the GASNet and ARMCI libraries as a communication layer, which provide a PGAS for distributed setups. Although PGAS is used internally, only a global memory address space is presented to the programmer. Only when implementing data distributions, storage locations can be explicitly specified. Each computation node is represented in Chapel by a *locale* object, which can be used with the *on* statement to map execution regions to specific nodes. All locales can be accessed through a global locale array.

For Chapel to gain wider acceptance, object oriented programming and generics are also supported. Both concepts allow for further increasing the programmer productivity in comparison to other programming models that are currently often used in scientific computing. Nonetheless, these concepts are seen as optional, and they are not as powerful as in some purely object oriented languages like X10. [Chapel, 2009a, Chapel, 2009b, Callahan et al., 2004]

Support for data distribution

Chapel provides two concepts that are used for the implementation of arrays: *domains* and *distributions*. A domain is a mapping of an arbitrary index space to memory locations where indices are not restricted to integral types. As domains are first-class objects that can be implemented by the user, Chapel is able to support any type of array domains. Examples of array domains include sparse domains and associative domains, which are already included in the Chapel distribution. As the actual implementation of a domain is transparent to the user, the programmer can easily switch between different domain types without having to change his algorithm. The possibility to implement custom domains can even bring advantages in simple cases like two-dimensional arithmetic domains. They can be implemented to store a matrix blockwise to provide more efficient caching behaviour in matrix multiplication algorithms. Each domain also provides an iterator that can be used in *for* and *forall* loops. In case of a domain that stores its data blockwise, the iterator may be implemented to access the elements in a blockwise manner similar to the order in which the matrix is stored.

Currently, Chapel supports five types of array domains. *Arithmetic domains* are used for arrays which have indices in an arithmetic range for one or more dimensions. *Sparse domains* are used to specify sparse arrays. For these arrays, only the given indices are actually stored in the array. Reading every other index of the arithmetic domain containing the subdomain out of the array returns 0. Indices can be dynamically added or removed from the sparse domain, and these changes are automatically reflected in the corresponding arrays. An *associative domain* can be used to declare associative arrays like hash tables. *Opaque domains* are similar to associative domains but with different performance characteristics. If changes to the domain are rare, they should be faster than an array with an associative domain. An *enumerated domain* creates associative arrays where each entry maps to a value in an enumeration data type.

The other concept provided by Chapel, data distribution, allows to map indices in a source domain to indices in a target domain. The target domain corresponds to an array of physical memory locations, which are called *locales* in Chapel. This mapping allows Chapel to decide on

which locale to store each array element. Similar to domains, distributions are first-class objects that can be implemented by the user. Moreover, like in domains, the implementation of data distributions is completely transparent to the implementer so that the programmer can easily switch to another distribution without having to change his algorithm.

Figure 4.21 shows an example program written in Chapel that operates on a matrix which is distributed using a two-dimensional block distribution. Changing the distribution to another one only requires importing the corresponding package and modifying the line initializing the data distribution.

The ability to implement custom data distributions is one of the major strengths of Chapel. The goal of the Chapel developers is to provide a framework to allow development of data distributions by third party developers. This might lead to the availability of a wide variety of third party libraries that provide different types of domains and distributions. [Diaconescu and Zima, 2007]

Support for work distribution

In Chapel work distribution is often automatically derived from the data distribution. Each array domain provides a sequential and a parallel iterator that can be used to iterate through all indices in the domain. The parallel iterator is responsible for distributing work onto different shared memory threads in a parallel loop. When iterating over a domain that is distributed over multiple locales using a distribution, each iteration is executed local in respect to the corresponding index.

Chapel's arrays also provide a very high-level interface. Scalar operations, like additions and multiplications, can be applied to whole arrays and are interpreted as an element-wise application of this operation. Arrays can also be passed as parameters to scalar functions, which leads to the function being applied to each element of the array, and the results being combined into an array of the same domain as the input array. Where possible, the execution of these array statements is done in parallel. The implementation of the example matrix multiplication kernel (Figure 5.14) takes advantage of array statements.

In addition to implicit work distribution that is derived from the data distribution, Chapel also provides primitives that allow manual work distribution. For distributed memory settings Chapel offers the *on* clause (see Figure 4.22) that allows explicit specification of the locale the following statement or block should run on. The *on* clause can also be used inside *for* and *forall* loops to override the implicit work distribution onto distributed memory nodes.

To distribute work inside a shared memory node, Chapel provides the *begin* and *cobegin* clauses, which launch one or more tasks in parallel, and the *coforall* loop, where each iteration is executed as a separate task. Figure 4.23 depicts an implementation of the quicksort algorithm in Chapel. In this example parallelism is introduced by the *cobegin* statement that launches both recursive function calls contained in the block following cobegin as separate tasks. The *cobegin* statement provides an implicit synchronization after the end of the block.

```
use BlockDist;

// Size of each dimension of our domain.  Note that int(64) is the
// default for domains and arrays distributed with the Block
// distribution.
config const n: int(64) = 8;

// Declare and initialize an instance of the Block distribution Dist,
// a distributed domain Dom, and a distributed array A.  By default,
// the Block distribution distributes the domain across all locales.
var Dist = distributionValue(new Block(rank=2, bbox=[1..n, 1..n]));
var Dom: domain(2,int(64)) distributed Dist = [1..n, 1..n];
var A: [Dom] int;

//
// Loop over the array using a serial for-loop and assign each element
// an increasing number.
//
var j = 1;
for a in A {
    a = j;
    j += 1;
}
writeln("Initialized array");
writeln(A);
writeln();

//
// In parallel, subtract one from each element of the array.
//
forall i in Dom do {
    A(i) = A(i) - 1;
}
writeln("Subtracted 1 via parallel iteration over the domain");
writeln(A);
writeln();
```

Figure 4.21: Shortened version of the Block2D sample program delivered with the Chapel compiler.

Reductions and *Scans* are implemented in Chapel as a native language element and automatically distribute the necessary work. Figure 4.24 depicts a simple example where the addition operator is used to scan over an array that is initially filled with ones. In the implementation of the example matrix multiplication kernel in Figure 5.14, a sum reduction is used to calculate each element of the resulting matrix. Chapel also allows for implementing custom reduction and scan operations, but no details can be found in the documentation yet.

```
on Locales(1) {
  writeln(here.id);
}
```

Figure 4.22: Explicit work distribution in Chapel using the *on* clause.

```
def quicksort(data: [],
    left: int = data.domain.low,
    right: int = data.domain.high) {

    if(right > left) {
        const i = partition(data, left, right)
        cobegin {
            quicksort(data, left, i-1);
            quicksort(data, i+1, right);
        }
    }
}
```

Figure 4.23: A simple Chapel implementation of the quicksort algorithm.

```
var A: [1..3] int = 1;
writeln(+ scan A);
// Outputs: 1 2 3
```

Figure 4.24: Using scans in Chapel.

Memory model

Internally, Chapel uses a *partitioned global address space* (*PGAS*) as an abstraction layer on top of the distributed memory model. On the user side, the APGAS is mostly abstracted away so that, in general, the programmer does not have to specify locality of data accesses. In Chapel's syntax no distinction is made between local and remote memory accesses. For arrays data distributions can be specified that are then translated by the compiler and runtime to local or remote memory accesses. The address space is only partitioned between distributed memory nodes, which means that inside one node a global address space is used.

In the current Chapel specification, the memory consistency model is defined as being sequentially consistent for *data-race-free* programs, but this point is still open for discussion and might, therefore, be subject to change.

Coordination primitives

Coordination in Chapel is completely memory based and Chapel does not provide message passing capabilities. One way to coordinate Chapel programs is by the use of *sync* variables.

Sync variables provide lock semantics in that they can have two states: *full* and *empty*. A write operation on a sync variable transitions the variable to *full* state or blocks execution if the variable is already full and waits for it to empty. Read operations on a sync variable put the variable back to *empty* state or block execution as long as the variable is *empty*. Figure 4.25 depicts how a sync variable can be used as a lock. In this example, the task launched with *begin* blocks at the first line, where the sync variable is read. It can only continue after the end of the main task, where the sync variable is assigned a value.

```
var finishedMainOutput$: sync bool;
begin {
  finishedMainOutput$;
  writeln("output from spawned task");
}
writeln("output from main task");
finishedMainOutput$ = true;
```

Figure 4.25: Using sync variables as locks Chapel.

Another more high-level means of coordination in Chapel is the *atomic* statement. It allows to specify a block of code to be executed atomically in respect to the rest of the program so that no variable assignment becomes visible until the whole atomic block has executed. Figure 4.26 shows how a delete statement in a doubly linked list can be implemented using atomic blocks. Here the atomic block ensures that changes to both neighbours become visible at the same time so that the linked list stays consistent. It is still an open issue whether Chapel's atomic statements should provide *strong atomicity* (an atomic block is atomic in respect to the whole program) or *weak atomicity* (an atomic block is only atomic in respect to other atomic blocks).

```
atomic {
    if(prev != null) {
        prev.next = next; }
    if(next != null) {
        next.prev = prev; }
}
```

Figure 4.26: Simplified implementation of a delete in a doubly linked list using atomic blocks in Chapel.

4.3 Models for heterogeneous architectures

In the context of this work, the term *heterogeneous architecture* describes architectures that consist of different types of processors. Heterogeneous architectures are still a young research topic and present additional difficulties compared to the other presented architectures. There

has been some research to add support for heterogeneous architectures to MPI [Kumar et al., 2007], but the question remains whether MPI can fully realize their potential. This section gives a short overview over some models that have been proposed to solve the difficulties posed by heterogeneous architectures.

4.3.1 Computation offloading (OpenCL)

Message passing and multithreading are programming models that assume that they operate on homogeneous architectures. These programming models do not differentiate between the performance characteristics and memory restrictions of different nodes. So for programs to work well on heterogeneous architectures, the programmer has to do the differentiation manually. In addition, architectural differences between accelerator chips and the CPU often make it impossible to use these programming models. Therefore, the *computation offloading* programming model has emerged, which is a concept orthogonal to message passing and multithreading. In this programming model the program is separated into program logic and computational kernels. The program logic is executed on the normal CPU, and the computation of a kernel is started by sending the kernel program, compiled for the accelerator architecture, and parameters to one or more accelerator units. After the computation has been completed, the program logic pulls the results from the accelerator units.

Some accelerators, like the *Synergistic Processor Elements* (*SPE*) in the *Cell processor*, support more complex communication patterns where accelerators can directly communicate with each other through non-coherent DMA transfers. This allows the use of programming models other than computation offloading for the Cell processor. But as there is no memory consistency for accesses outside of the local memory of each processing element, computation offloading is often preferable for programmer productivity and program stability reasons.

The computation offloading model allows optimizing programs for *throughput* through *latency hiding*. Communication is only done at the time when a kernel is uploaded to an accelerator, and when the results are fetched back. During the kernel computation, all data lies in the local memory of the accelerator unit, which usually has low latency.

Most libraries used for computation offloading are specific for a vendor of accelerators. Only at the end of 2008, the *OpenCL* specification was released. OpenCL is a vendor-independent standard developed by the Khronos Group in cooperation with many industry-leading companies. It aims at unifying the access to accelerators and thus increasing program portability. The OpenCL specification explicitly states that OpenCL does not aim to provide high-level constructs, but instead only low-level interfaces which abstract away hardware specifics. The concepts used in OpenCL are mostly taken from existing libraries for computation offloading like *CUDA* by nVidia.

At the time of writing, AMD and nVidia have already released software development kits that provide support for OpenCL for their GPUs. IBM has also released an OpenCL implementation which supports the Cell processor. OpenCL will make it easier to create high-level programming models for heterogeneous architectures and might lead to some new developments in the near

future. [OpenCL, 2009]

Except for OpenCL, there also exist numerous other vendor independent computation offloading toolkits like the *HMPP Workbench* [HMPP, 2010], *Codeplay Offload* [Codeplay, 2010] and *ASTEX* [Petit et al., 2006]. These toolkits all try to provide an approach to computation offloading on a higher level compared to OpenCL. OpenCL may allow the vendors of these toolkits to provide support for a wider range of hardware platforms by removing the requirement to create a backend for each platform.

4.3.2 Stream programming

One of the problems with the given high-level programming models is that they still do not provide good support for heterogeneous architectures. This means that to be able to take advantage of accelerator chips like *General Purpose Graphics Processing Units* (*GPGPUs*) or the *Synergistic Processing Elements* (*SPEs*) of the Cell processor, the programmer would have to resort to techniques like *computation offloading*. (see Section 4.3.1)

A more high-level approach to computation offloading is *stream programming*. As in computation offloading, the program is split into program logic and computational kernels. In the program logic, data that has to be worked upon is put into streams using *gather* operation and then passed to one or more computational kernels. Output streams containing results are passed back to the main program using *scatter* operations. The computational kernels themselves only operate on their input and output streams.

Figure 4.27 shows a simple example of a stream program. In it, Kernel 1 receives three streams of data (a, b and c), which it uses to calculate d. The output stream d, and another input stream (e) are then used as input to Kernel 2. The result from Kernel 2 can then either be scattered back to the main program or further used in other kernels. The runtime of a streaming language automatically distributes the data in the streams to the accelerator units. It sends the streaming data to the local memory of the accelerator units and retrieves the resulting data after the calculation for this streaming element has been finished. Sophisticated implementations can use the information about subsequent stream operations to keep some input or output data in the local memory for the next streaming operation so that data does not have to be transferred between these iterations. By utilizing some compiler optimization techniques in the streaming runtime, it is even possible to split up or merge different streaming operations as it is done in Ct. (see Section 4.3.3)

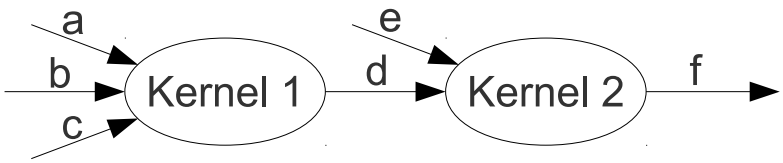


Figure 4.27: Dataflow Graph of a stream program.

Most development is currently done to use stream programming for GPGPU programming, like in the programming languages *BrookGPU* and *Sh*. *Rapidmind* [Monteyne, 2008], a commercial stream programming language which is based on Sh, aims to support a wide variety of accelerator chips through different backends. Nonetheless, stream programming can also be of value for parallelization of kernels on normal CPUs. In [Gummaraju and Rosenblum, 2005] techniques are described to utilize CPUs in stream programming languages. The programming language *Ct* by Intel, which will be described in the next section, is specialised to support Intel's future terra-scale processor architectures.

4.3.3 Ct

Ct is a research effort by Intel to create an autoparallelizing C++ library based on the stream programming model. It introduces vector primitive templates, which can be used with any scalar type, and operations that can be applied to these vectors. A vector object is immutable and all operations which can be applied to it are side-effect free and return a new vector object. This results in less locking overhead and allows for aggressive optimizations.

Ct supports the use of nested parallelism, thus making it easier to implement certain classes of algorithms based on recursion, such as the divide and conquer class of algorithms (e.g. Quicksort), in a natural and efficient way. The vector primitives themselves can also be nested, which allows for supporting complex data structures, like sparse matrices, in an efficient manner. All operators supported by Ct can cope with nested data structures.

Compilation can be done using standard C++ compilers for the x86 and x86-64 architectures. Ct operations are implemented as function calls to the Ct dynamic engine, which in fact is a second compiler that resides in the runtime of a Ct program. This runtime compiler uses information from function calls to create an intermediate code representation of current calls. Data flow analysis helps the compiler to reduce communication overhead, as results from one computation can be kept locally in the memory of the processor core for the next computation. In addition to that, the compiler can take advantage of kernel splitting and merging techniques, which, in combination with other compiler optimization techniques, can optimize execution. The advantage of this runtime approach is that the runtime compiler has some information about the streaming operations which cannot be determined by a static compiler.

At the time of writing, no public release of Ct is available and, therefore, no experiments with Ct can be performed in the context of this work. Nonetheless, concepts found in Ct provide an interesting outlook on the theoretical potential for optimization in the stream programming model. It has yet to be shown, how high the gains provided by these techniques will be in practice. [Ghuloum et al., 2007b, Ghuloum et al., 2007a]

Chapter 5

Experiments

This chapter is aimed at providing a comparison of high-level programming models from the programmer's point of view. The evaluation thereby focuses on programming models for shared memory architectures due to the increased public interest in these models since the emergence of multi-core processors. Furthermore, the suitability of the tested programming models for distributed memory architectures will be described but, however, not evaluated. Programming models suitable for heterogeneous architectures are not included in this evaluation.

5.1 Approach used

For the experiments, a matrix multiplication kernel is implemented in each programming model. As a reference implementation a sequential matrix multiplication kernel written in C++ is used. The reference kernel is implemented in a simple and intuitive way (*ijk-form*) that does not consider performance. For comparison, variants of the kernel are used which contain some simple optimizations that are well known and often used by programmers. Ideally, the manual use of such optimizations should not be necessary in a programming model, but should instead be automatically applied by the compiler.

Depending on the tested programming model, either the reference implementation is ported to it or, if the intuitive implementation of this kernel in this model differs from the reference implementation, it is reimplemented.

For languages which have not been optimized for performance yet, and that are, therefore, still not used in production systems, no performance measurements are done. Measuring the absolute performance for these languages would not be representative, especially when comparing them to other models. Still some interesting information can be gained by looking at the speedup, so some performance data is evaluated to show this. For these models the example kernel is primarily implemented to gain some practical experience with the programming models.

For all other models, build-scripts are created based on the build-script of the reference implementation.(see Listing 2) These scripts build program executables for different problem sizes, numbers of threads and implementation variants. Additional parameters that can influence

the performance, like grain size, are added for programming models where they are necessary. All executables can then be run by a run-script (see Listing 3) that iterates through all executables and runs them one after the other, aggregates the results and writes them to a results file. This file is then used for manual analysis of the data.

These results are then further analysed manually to find interesting characteristics of this programming model. The goal is to find out how good the performance is for the intuitive implementation, how much performance can be achieved by doing some simple optimizations and how much the choice of additional parameters, like grain size, influences the results.

To get more representative results, each kernel is run for a configured number of times and the run-time of each run is measured. Both the best and the worst run are dropped, and the average run-time of the other runs is output. All tests are done using double-precision floating point variables.

5.2 The matrix multiplication kernel

A matrix multiplication of the form $C = A \times B + C$ is a typical kernel that is widely used. It is also available as a part of the BLAS library [Blas, 2008] which provides mathematical kernels for scientific applications. Although in its simplest forms the sequential version is easy to implement, there is still a lot of room for optimization by increasing cache-locality. (An introduction into optimization for matrix multiplications can be found in [Wolfe, 1995].)

The simplest sequential reference implementation used in this evaluation is based on the standard ijk form which can be seen in Figure 5.1. Although it is the most intuitive form, it is far away from optimal performance. Assuming *column major order* (Column elements of a matrix are stored in continuous memory), the problem is that for matrix B the inner loop jumps between different columns of the matrix while reading only one element per column. For large matrices, which do not fully fit into the processor cache, this leads to memory lines being cached only for one element access and then being replaced by another memory line before they are needed again.

```
for(int i = 0; i < N; i++)
    for(int j = 0; j < N; j++)
        for(int k = 0; k < N; k++)
            C[i*N + j] += A[i*N + k] * B[k*N + j];
```

Figure 5.1: Sequential reference implementation of the ijk form of matrix multiplication.

A typical optimization, which is automatically applied by some compilers is loop reordering. In the case of matrix multiplication reordering the “j” and the “k” loop leads to more efficient behaviour. After reordering, the inner loop iterates through a column of matrix B, meaning that the whole cache line is utilized for the iteration. In addition to that, only one element of A is accessed in the inner loop so that it can be kept in the processor registers. This form of

matrix multiplication is typically called *ikj form* in the literature.

```
for(int i = 0; i < N; i++)
    for(int k = 0; k < N; k++)
        for(int j = 0; j < N; j++)
            C[i*N + j] += A[i*N + k] * B[k*N + j];
```

Figure 5.2: Sequential reference implementation of the ikj form of matrix multiplication.

Another typical optimization is dividing the given matrices into submatrices so that each submatrix fits into the cache. The matrix multiplication is then split into an inner and an outer multiplication. In the inner multiplication, two submatrices are multiplied and added to the corresponding submatrix of the result matrix. The outer multiplication multiplies the global matrix consisting of submatrices with each other. Figure 5.3 shows the reference implementation of this method. Both the outer and the inner multiplication are implemented using the *ikj* method.

```
for(int it = 0; it < N; it+=STRIDE)
    for(int kt = 0; kt < N; kt+=STRIDE)
        for(int jt = 0; jt < N; jt+=STRIDE)
            for(int i = it; i < (it + STRIDE) && i < N; i++)
                for(int k = kt; k < (kt + STRIDE) && k < N; k++)
                    for(int j = jt; j < (jt + STRIDE) && j < N; j++)
                        C[i*N + j] += A[i*N + k] * B[k*N + j];
```

Figure 5.3: Sequential reference implementation of the submatrix form of matrix multiplication.

Unfortunately, doing two comparison operations instead of one in the inner loop leads to some overhead compared to the ikj form. To solve this, one can only allow matrices the dimensions of which are divisible by the stride. If support for other matrices is needed, the only solution is to do the submatrix multiplication for the biggest submatrix that fulfils this criterion, and to handle the corner cases in additional loops where the other condition is used then. Although this optimization is actually already too low-level for the scope of this work, it has also been implemented as a sequential reference implementation and included as the fourth variant "unrolled submatrix form", to be able to determine how much the setting of a grain size, which is possible in some programming models, can utilize the potential of submatrix multiplications.

5.3 Testing environments

For the performance evaluation three different machines were used.

The first machine, *daisy*, is a *Sun Fire X4600 M2* system. It consists of eight *AMD Opteron 8218* dual-core processors that run at a frequency of 2.6 GHz. This system has been chosen as it allows sixteen-fold parallelization on x86 processors. The software stack consists of the

Sun Solaris 10 operating system, the *Sun Studio 12* compiler collection and the *GNU compiler collection* (*GCC*) version 3.4.3.

The second system, *rose*, is a *Sun Fire T5140* which consists of two Sun UltraSPARC T2 Plus processors. This system is especially interesting because of its unique processor architecture. Each of the processors consists of eight cores, which in turn are capable to do eight-way *Simultaneous Multithreading* (*SMT*). The *CoolThreads* technology used in the processor allows to do fast thread-switching whenever a high-latency operation is issued. This makes the processor well suitable for high-throughput applications, especially as the single-thread performance of the processor is fairly low compared to other processors. Like on *daisy* the software stack consists of *Sun Solaris 10*, *Sun Studio 12* and *GCC 3.4.3*.

Unfortunately, both *daisy* and *rose* are not suitable for doing all tests on. Most programming models are first developed for the most common platforms, which means that not all of them support the Sun Solaris operating system. The situation for *rose* is even worse as the SPARC architecture is less common as the x86 architecture, which is why only few programming models support it.

To overcome these limitations, a third system, *fleur*, has been chosen for the tests. This machine consists of two quad-core *AMD Opteron 2378* processors and, therefore, allows for eight-fold parallelization at least. Its software stack consists of the 64-bit version of Ubuntu 8.04 Server and *GCC 4.2.4*. The kernel version is 2.6.24.

An overview of the used systems can be found in Table 5.1

System	Daisy	Rose	Fleur
Type	Sun Fire X4600 M2	Sun Fire T5140	HP DL385 R05p
Processors	8x AMD Opteron 8218	2x Sun UltraSPARC T2+	2x AMD Opteron 2378
Threads	16	128	8
Compilers	SUN Studio 12	SUN Studio 12	GCC 4.2.4
	GCC 3.4.3	GCC 3.4.3	

Table 5.1: Testing environments.

5.4 Sequential version

The sequential version of the matrix multiplication is implemented in C++ and has been compiled using the compilers accessible on the machines. The results of the performance measurements can be found in the Tables 5.2-5.6. For the submatrix forms of the multiplication different strides have been tried out. Only the result that has provided the best performance is shown in the tables.

On fleur (Table 5.2) it can be seen that memory layout becomes a performance dominating factor on matrix dimensions higher than 256. On matrix sizes higher than 1024 it is advantageous to split the matrix into single blocks that fit into the processor cache. Some higher-level programming models should theoretically be able to automatically utilize these effects.

Variant	Dimensions	Stride	Avg. time (sec.)
ijk	256		0.10
ijk	512		0.95
ijk	1024		32.63
ijk	2048		456.29
ikj	256		0.08
ikj	512		0.26
ikj	1024		3.99
ikj	2048		29.34
submatrix	256	256	0.06
submatrix	512	512	0.26
submatrix	1024	512	2.70
submatrix	2048	128	23.60
submatrix_unroll	256	128	0.06
submatrix_unroll	512	512	0.26
submatrix_unroll	1024	512	2.19
submatrix_unroll	2048	256	19.23

Table 5.2: Performance results of the matrix multiplication code on fleur.

Variant	Dimensions	Stride	Avg. Time (sec.)
ijk	256		0.11
ijk	512		4.36
ijk	1024		48.36
ijk	2048		622.21
ikj	256		0.02
ikj	512		0.56
ikj	1024		4.48
ikj	2048		46.29
submatrix	256	32	0.04
submatrix	512	128	0.33
submatrix	1024	128	2.74
submatrix	2048	64	24.55
submatrix_unroll	256	32	0.02
submatrix_unroll	512	256	0.20
submatrix_unroll	1024	128	1.84
submatrix_unroll	2048	64	17.26

Table 5.3: Performance results of the matrix multiplication code on daisy (SunCC).

On daisy (Tables 5.3 and 5.4), the submatrix variant is already advantageous for matrix sizes bigger than 256. Also, the optimal strides are typically smaller on daisy. This can be explained by the smaller cache sizes compared to fleur. It should also be noted that even for this simple matrix multiplication example there is a big performance difference between the Sun and the GCC compiler. As the Sun compiler would be the compiler of choice for this system, the results

Variant	Dimensions	Stride	Avg. time (sec.)
ijk	256		0.13
ijk	512		5.55
ijk	1024		113.64
ijk	2048		952.82
ikj	256		0.04
ikj	512		0.72
ikj	1024		4.72
ikj	2048		37.46
submatrix	256	256	0.04
submatrix	512	128	0.38
submatrix	1024	128	3.06
submatrix	2048	64	29.76
submatrix_unroll	256	256	0.03
submatrix_unroll	512	256	0.29
submatrix_unroll	1024	128	2.61
submatrix_unroll	2048	64	23.39

Table 5.4: Performance results of the matrix multiplication code on daisy (GCC).

from Table 5.3 will be used as a reference for the evaluation of all programming models on this machine.

Variant	Matrix dimensions	Stride	Avg. time (sec.)
ijk	256		0.49
ijk	512		3.89
ijk	1024		48.23
ijk	2048		1389.37
ikj	256		0.34
ikj	512		2.77
ikj	1024		31.05
ikj	2048		248.21
submatrix	256	256	0.45
submatrix	512	256	3.70
submatrix	1024	512	29.04
submatrix	2048	256	235.33
submatrix_unroll	256	256	0.35
submatrix_unroll	512	256	2.78
submatrix_unroll	1024	256	22.24
submatrix_unroll	2048	256	176.83

Table 5.5: Performance results of the matrix multiplication code on rose (SunCC).

On rose (Tables 5.5 and 5.6) it can be noted that for matrix sizes bigger than 512 the submatrix multiplication performs better. Again, SunCC outperforms GCC, which means that the results from Table 5.5 will be used as a reference in the evaluation.

Variant	Matrix dimensions	Stride	Avg. time (sec.)
1	256		0.58
1	512		4.66
1	1024		55.81
1	2048		1623.46
1	256		0.51
1	512		4.09
1	1024		61.95
1	2048		496.92
1	256	256	0.77
1	512	256	5.98
1	1024	256	47.51
1	2048	128	382.86
1	256	256	0.51
1	512	512	4.09
1	1024	256	32.60
1	2048	256	261.48

Table 5.6: Performance results of the matrix multiplication code on rose (GCC).

5.5 Results

5.5.1 OpenMP

The OpenMP version of the matrix multiplication is relatively easy to implement. (see Listing 5) To parallelize the loops of the sequential version it is only necessary to add the compiler directive `#pragma omp parallel for` before the outmost loop over i or j. All four variants of the matrix multiplication have been parallelized in this way. Figures 5.4 and 5.5 depict the resulting code for the ijk and the submatrix form.

```
#pragma omp parallel for
for(int i = 0; i < N; i++)
    for(int j = 0; j < N; j++)
        for(int k = 0; k < N; k++)
            C[i*N + j] += A[i*N + k] * B[k*N + j];
```

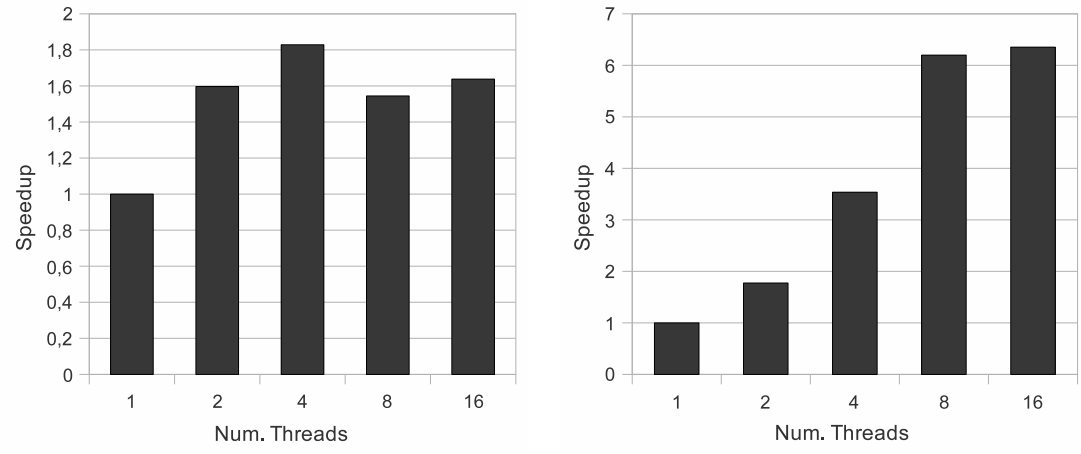
Figure 5.4: OpenMP implementation of the ijk form of matrix multiplication.

The performance measurement results gained on daisy show that potential speedup for the matrix multiplication kernel in OpenMP is limited by the way in which it is implemented. Figure 5.6(a) depicts the speedup gained for the ikj-variant of the matrix multiplication code. It can be seen that for this variant speedup is very limited. In fact, for the eight processor run the parallelization overhead seems to outweigh the gains compared to the four processor run so that execution is actually slower. Interestingly, performance increases again while raising the number of threads to sixteen. To explain this phenomenon, tests have been done using different

```
#pragma omp parallel for
for(int it = 0; it < N; it+=STRIDE)
    for(int kt = 0; kt < N; kt+=STRIDE)
        for(int jt = 0; jt < N; jt+=STRIDE)
            for(int i = it; i < (it + STRIDE) && i < N; i++)
                for(int k = kt; k < (kt + STRIDE) && k < N; k++)
                    for(int j = jt; j < (jt + STRIDE) && j < N; j++)
                        C[i*N + j] += A[i*N + k] * B[k*N + j];
```

Figure 5.5: OpenMP implementation of the submatrix form of matrix multiplication.

thread-affinity settings. It seems that on daisy OpenMP threads are first distributed between physical processors and then between processor cores. In case of the matrix multiplication this is a disadvantage as the communication overhead is higher between physical processors than between different processor cores.



(a) ikj (b) unrolled submatrix form

Figure 5.6: Speedup of the OpenMP implementation of the matrix multiplication code on daisy - matrix size: 2048

If we compare the speedup achieved by the ikj-variant of the kernel with the speedup achieved by the unrolled submatrix variant, it can be seen that the optimized version of the kernel also scales better. This variant scales well up to eight threads, and at least some speedup can be noticed when again doubling the number of threads to a total of sixteen.

Detailed run-times of the test-runs on daisy are depicted in the Tables 5.7 and 5.8. It should be noted that on daisy the one-processor-run of the OpenMP version was always faster than the sequential version. This behaviour might result from additional code optimizations that are only done by the compiler when the OpenMP flag is set. For this reason, the one-processor-run of the OpenMP version has been taken as a reference point for the calculation of speedup.

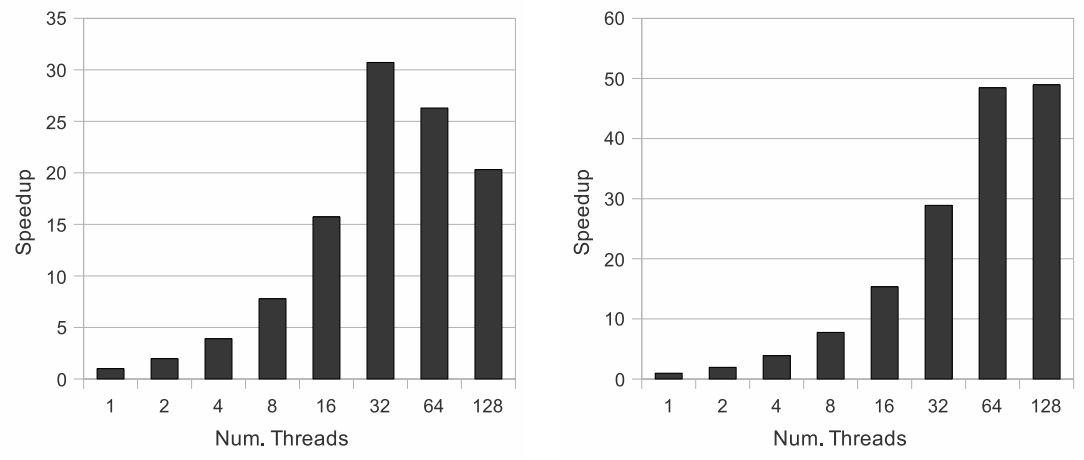
The observation that the submatrix variant can exhibit higher speedup can be reproduced on both rose and fleur, although the ikj-variant scales better on these platforms. Figure 5.7

Type	Threads	Avg. Time	Speedup
Sequential reference	1	46.29	0.70
OpenMP	1	32.37	1.00
OpenMP	2	20.28	1.60
OpenMP	4	17.70	1.83
OpenMP	8	20.96	1.54
OpenMP	16	19.77	1.64

Table 5.7: Performance results of the OpenMP implementation of the ikj variant of the matrix multiplication code on daisy - matrix size: 2048

Type	Threads	Stride	Avg. Time (sec.)	Speedup
Sequential reference	1	64	17.26	0.99
OpenMP	1	32	17.11	1.00
OpenMP	2	32	9.64	1.77
OpenMP	4	32	4.84	3.54
OpenMP	8	32	2.76	6.20
OpenMP	16	32	2.69	6.35

Table 5.8: Performance results of the OpenMP implementation of the unrolled submatrix variant of the matrix multiplication code on daisy - matrix size: 2048

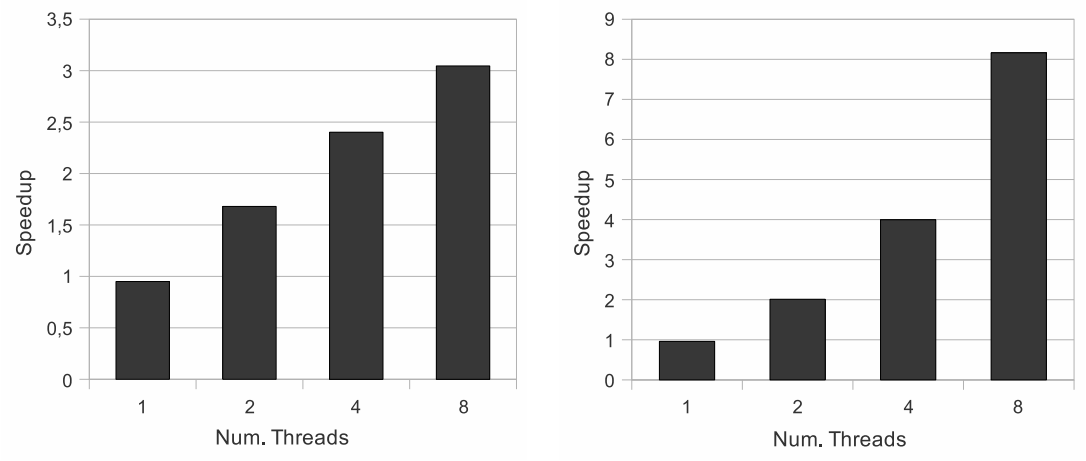


(a) ikj (b) unrolled submatrix form

Figure 5.7: Speedup of the OpenMP implementation of the matrix multiplication code on rose - matrix size: 2048

depicts that on rose both variants exhibit nearly linear speedup up to 32 threads. This means that the application is even able to efficiently utilize two-way SMT, as rose consists of only sixteen cores. When using more than 32 threads in the ikj-variant, the parallelization overhead seems to exceed the possible gains, which results in a slowdown compared to the 32-thread run. The unrolled submatrix variant scales up even further with a speedup-factor of up to 49.

On fleur (see Figure 5.8) the unrolled submatrix variant scales especially well. It can be



(a) ikj (b) unrolled submatrix form

Figure 5.8: Speedup of the OpenMP implementation of the matrix multiplication code on fleur - matrix size: 2048

noticed that a speedup better than linear was achieved during the test-run. Such an effect can be explained with increased cache-locality due to the split of the iteration space onto multiple processors.

The analysis of OpenMP's performance leads to the conclusion that OpenMP is able to efficiently utilize parallelism on shared-memory architectures, and to provide good speedup and good absolute performance. However, with OpenMP it becomes even more important to optimize memory access patterns compared to sequential code as they not only influence the absolute performance but also the possible speedup. This means that effectively OpenMP programs still have to be programmed on a very low level to run efficiently.

5.5.2 Threading Building Blocks

Due to the fact that TBB is only an object oriented library, the parallelization of the matrix multiplication is not as easy as it is with other programming models. (see Listing 7) Each matrix multiplication kernel has to be moved out into its own function object that can then be executed via the *parallel_for* function. Inside the kernel the iteration ranges for the loops which are to parallelize have to be replaced by TBB's range objects. An example implementation is shown in Figure 5.9. Parallelizing two loops gets even more complex than just parallelizing one as it requires the developer to use a *blocked_range2d* object to specify the iteration range. When converting an existing 1-dimensional implementation to two dimensions this means that not only the function object has to be modified but also all calls to *parallel_for* which use this function object. A 2-dimensional implementation of the ikj-variant is depicted in Figure 5.10.

Tests with TBB were both done using the 1-dimensional and the 2-dimensional implementation. The tests with two dimensions were made to see if the range-object *blocked_range2d* provides better cache-locality compared to the one-dimensional *blocked_range*. Since TBB re-

```
struct Submatrix {
    void operator() (const blocked_range<int>& range) const
    {
        for(int it = range.begin(); it < range.end(); it += STRIDE)
            for(int kt = 0; kt < N; kt += STRIDE)
                for(int jt = 0; jt < N; jt += STRIDE)
                    for(int i = it; i < (it + STRIDE) && i < range.end(); i++)
                        for(int k = kt; k < (kt + STRIDE) && k < N; k++)
                            for(int j = jt; j < (jt + STRIDE) && j < N; j++)
                                mat3[i*N j] += mat1[i*N k] * mat2[k*N j];

    }
};

void runSubmatrix()
{
    Submatrix submatrix;
    parallel_for(blocked_range<int>(0, N, GRAINSIZE), submatrix);
}
```

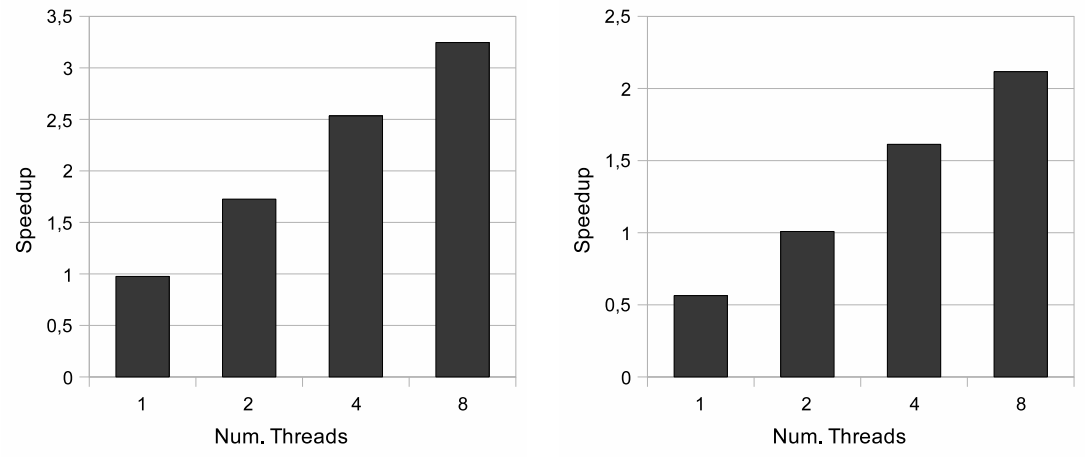
Figure 5.9: TBB implementation of the submatrix form of matrix multiplication.

```
struct Ikj {
    void operator() (const blocked_range2d<int, int>& range) const
    {
        for(int i = range.cols().begin(); i < range.cols().end(); i++)
            for(int k = 0; k < DIMENSIONS; k++)
                for(int j = range.rows().begin(); j < range.rows().end(); j++)
                    mat3[i*N + j] += mat1[i*N + k] * mat2[k*N + j];

    }
};

void runIkj()
{
    Ikj ikj;
    parallel_for(blocked_range2d<int, int>(0, N, GRAINSIZE, 0, N, GRAINSIZE), ikj);
}
```

Figure 5.10: TBB implementation of the ikj form of matrix multiplication using a 2-dimensional range object.

quires a grain size to be specified by the programmer, different settings have been tested with the ijk-variant and the ikj-variant to see which setting provides the best performance. For the submatrix variants a grain size of 1 was used because it would have made little sense to increase it there.

When looking at the performance of the example kernels, some interesting insights can be gained. Contrary to OpenMP, no performance can be gained using the unrolled submatrix form. Relative to the corresponding one-processor run, both kernels scale in a similar manner, but in absolute numbers the ikj-variant outperforms the unrolled submatrix variant (see Figure 5.11). For the grain size different settings were used for the ijk and ikj variants, but no correlation between performance and grain size has been found. The grain size that provided the best performance for a specific configuration can be found in Table 5.9.



(a) ikj (b) unrolled submatrix form

Figure 5.11: Speedup of the TBB implementation of the matrix multiplication code on fleur - matrix size: 2048

Type	Threads	Grain size	Avg. Time (sec.)	Speedup
Sequential reference	1		29.34	1.00
TBB	1	4	30.07	0.98
TBB	2	16	17.00	1.73
TBB	4	4	11.58	2.53
TBB	8	1	9.04	3.25

Table 5.9: Performance results of the TBB implementation of the ikj variant of the matrix multiplication code on fleur - matrix size: 2048

To see whether a 2-dimensional range can be used in TBB to replace the manual optimization of splitting the matrices into submatrices, tests have also been done using a *blocked_range2d* object. The corresponding code can be found in Figure 5.10. As can be seen in Table 5.11, the performance is only a bit worse than the performance of the unrolled submatrix variant in Table 5.10. Other than in the one-dimensional implementation (see Table 5.9) a correlation between grain size and performance can be noticed. The optimal grain size for fleur seems to be 256, which corresponds to the optimal stride of 256 in the unrolled submatrix variant. This shows that range objects are flexible enough to be used instead of the manual submatrix optimizations.

Type	Threads	Stride	Avg. Time (sec.)	Speedup
Sequential reference	1	256	19.23	1.00
TBB	1	256	34.12	0.56
TBB	2	256	19.06	1.01
TBB	4	128	11.92	1.61
TBB	8	256	9.08	2.12

Table 5.10: Performance results of the TBB implementation of the unrolled submatrix variant of the matrix multiplication code on fleur - matrix size: 2048

Type	Threads	Grain size	Avg. Time	Speedup
Sequential reference	1		29.34	1.00
TBB	1	256	36.52	0.80
TBB	2	256	19.67	1.49
TBB	4	256	12.93	2.27
TBB	8	256	11.28	2.60

Table 5.11: Performance results of the 2-dimensional TBB implementation of the ikj variant of the matrix multiplication code on fleur - matrix size: 2048

One major problem of TBB seems to be that although its high-level nature allows to abstain from some low-level optimizations, it cannot provide the same performance and scalability as OpenMP with low-level optimizations. The overhead created by TBB seems to be too high to be competitive. One reason might be that TBB relies on function calls inside parallel constructs, which may make it more difficult for the compiler to optimize. Apart from that, the compiler does not know about the parallelism in those constructs and, therefore, cannot use this information for optimizations. It is possible that compilers supporting more sophisticated optimizations than the GCC are able to optimize TBB code better, but for the platform it was tested on this is not the case.

The experience from parallelizing the matrix multiplication kernels has already shown that parallelizing existing libraries with TBB can become much more work than it is with most other programming models. The experience with converting a 1-dimensional implementation to two dimensions shows that the function objects cannot be used as an abstraction layer as the calls to the function object still contain some implementation specifics. And yet TBB has the advantage of being easily extensible because it is just a C++ library. This might prove to be an advantage during the transition from sequential to parallel models where no model is able to support all use-cases. One major drawback is that for applications needing very good performance the performance potential is limited. Although it might perform better using other compilers, the optimization potential is limited by the compiler's lack of understanding of the parallelism in TBB. The author believes that in the long term TBB will not be of significant importance because of its complexity compared to other models and the limitations imposed by being only a C++ library.

5.5.3 Cilk++

Parallelizing the matrix multiplication in *Cilk++* is only a matter of replacing the normal *for*-statement with a *cilk_for* statement. Like TBB Cilk++ allows to specify a grain size for its parallel for-loop, but unlike TBB Cilk++ can also calculate an optimal grain size setting. To see whether the grain size can influence the performance of the matrix multiplication kernel, different grain size settings were used in the ijk and ikj variants.

```
cilk_for(int it = 0; it < N; it += STRIDE)
    for(int kt = 0; kt < N; kt += STRIDE)
        for(int jt = 0; jt < N; jt += STRIDE)
            for(int i = it; i < (it + STRIDE) && i < N; i++)
                for(int k = kt; k < (kt + STRIDE) && k < N; k++)
                    for(int j = jt; j < (jt + STRIDE) && j < N; j++)
                        mat3[i*N + j] += mat1[i*N + k] * mat2[k*N + j];
```

Figure 5.12: Cilk++ implementation of the submatrix form of matrix multiplication.

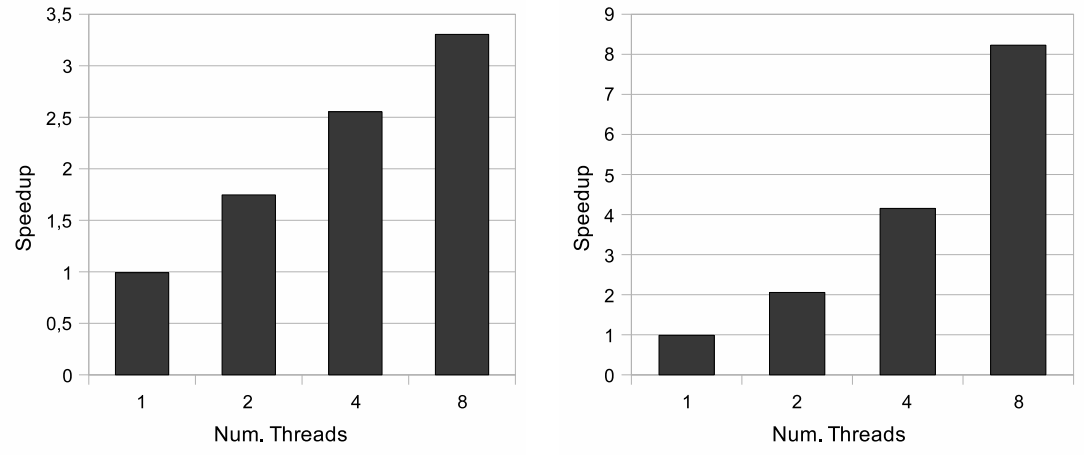
Type	Threads	Grain size	Avg. Time (sec.)	Speedup
Sequential reference	1		29.34	1.00
Cilk++	1	1	29.58	0.99
Cilk++	2	automatic	16.80	1.75
Cilk++	4	16	11.49	2.55
Cilk++	8	8	8.88	3.31

Table 5.12: Performance results of the Cilk++ implementation of the ikj variant of the matrix multiplication code on fleur - matrix size: 2048

The test results do not show any correlation between grain size and performance. Apart from that, the performance and scalability are similar to OpenMPs so that the same findings should apply. This means that the unrolled submatrix optimization not only influences absolute performance, but also scalability. When looking at Figure 5.13(b) a better than linear speedup can be noticed compared to the sequential reference implementation, whereas for the ikj variant in Figure 5.13(a) the speedup with 8 threads amounts to only 3.3.

Type	Threads	Stride	Avg. Time (sec.)	Speedup
Sequential reference	1	256	19.23	1.00
Cilk++	1	256	19.49	0.99
Cilk++	2	256	9.36	2.05
Cilk++	4	256	4.63	4.15
Cilk++	8	256	2.34	8.23

Table 5.13: Performance results of the Cilk++ implementation of the unrolled submatrix variant of the matrix multiplication code on fleur - matrix size: 2048



(a) ikj (b) unrolled submatrix form

Figure 5.13: Speedup of the Cilk++ implementation of the matrix multiplication code on fleur - matrix size: 2048

5.5.4 Chapel

Chapel's focus on high-level array operations allows for implementing the matrix multiplication example kernel in a more elegant way than in the sequential reference implementation. (see Listing 8) It can in fact be derived from the mathematical definition of a matrix multiplication. The following formula depicts how each element of the resulting matrix C is calculated:

$$C_{i,j} = \sum_{r=1}^{n} A_{i,r} B_{r,j} \tag{5.1}$$

Each element $C_{i,j}$ is calculated as the sum of products of the elements of the i^{th} row of matrix A and the j^{th} column of matrix B. In Figure 5.14 an array statement is used to calculate the product of the i^{th} row with the j^{th} column. The result of this multiplication is then reduced with the addition operator.

```
const ProblemSpace: domain(2) distributed(Block) = [1..n, 1..n];
var A, B, C: [ProblemSpace] real(64);

forall (i, j) in ProblemSpace do {
    C(i,j) = C(i,j)+ +reduce(A(i,1..n) * B(1..n, j));
}
```

Figure 5.14: Chapel implementation of a matrix multiplication.

Tests and performance measurements done using versions 0.8 and 0.9 of the prototype Chapel compiler showed that the Chapel compiler is still in an early stage of development. Data distributions and the interface to implement custom array distributions are still undocumented and are not recommended by the developers to be tested yet. Actual measurements were done using a

standard two-dimensional arithmetic array domain for the matrices. The measurements showed that neither the forall loop nor the array statements executed in parallel, therefore, there was no speedup when using more threads. Brad Chamberlain, one of the lead developers of Chapel, confirmed that this part has not been implemented yet. To test the actual autoparallelization capabilities of Chapel, a simple one-dimensional block distribution has been provided by the Chapel developers, which has been used by them to implement a Chapel version of the *stream* benchmark [Stream, 2009]. Measurements using the stream benchmark, in fact, showed a small speedup when using two threads.

As the prototype Chapel compiler is not recommended for productive use and as it does not aim to provide competitive performance, no detailed performance measurements have been done in the context of this work.

5.5.5 X10

Although X10 should provide more high-level means of implementing a matrix multiplication, the easiest way to implement it is similar to the sequential reference implementation and its counterparts in OpenMP, TBB and Cilk++. Figure 5.15 shows an example implementation for shared memory architectures. In this example a foreach loop iterates over both matrix dimensions, which might allow X10 to automatically apply the submatrix optimization to the iteration space.

```
finish foreach ((i,j) in [0,0]..[N-1,N-1])
    for(var k:int = 0; k < N; k++)
        C(i,j) += A(i,k) * B(k,j);
```

Figure 5.15: X10 implementation of a matrix multiplication for shared memory architectures.

While the shared memory example is very straightforward, the implementation becomes much more complex when trying to implement the matrix multiplication for distributed memory systems. The difficulty is that locality is explicit in X10, which leads to complex problems when actually trying to implement a matrix multiplication algorithm for distributed memory architectures. As there is still little documentation in this area, it has not been possible to implement the algorithm in a way that would actually have made sense performance-wise.

With future versions of X10 it might be possible to implement matrix multiplication in a more declarative syntax, which would also make it easier for the X10 compiler to optimize the code. A code example for calculating a dot product in X10 using declarative syntax can be found in [Saraswat and Nystrom, 2008, p123] and is also displayed in Figure 5.16. Unfortunately, this example has never been fully explained and no details on X10's declarative syntax can be found in the literature. Apart from that, the examples found in the literature do not seem to be consistent. The author believes that the declarative syntax is still not fully specified and cannot be used in the current implementation of the X10 compiler.

```
def dotProduct(a: Array[T](D), b: Array[T](D)): Array[Double](D) =
    (new Array[T]([1:D.places],
        (Point) => (new Array[T](D | here,
            (i): Point) => a(i)*b(i)).sum())).sum();
```

Figure 5.16: Dot product implementation using X10's declarative syntax. (Source: [Saraswat and Nystrom, 2008, p123])

Like in the evaluation of Chapel, no detailed performance measurements were done using the X10 compiler, as its developers have not aimed at providing competitive performance yet. It can be noted that during the test-runs of the matrix multiplication code some speedup could be achieved compared to a one processor run of the X10 code, but compared to other programming models X10 is still very slow.

Chapter 6

Conclusion

Even when ignoring heterogeneous architectures, parallel programming is still a challenge. For the time being, distributed memory architectures will be ignored by most programmers. Only in High Performance Computing the support of distributed memory architectures is essential, but there people will still rely on the "assembler of parallel programming", namely MPI. At least on shared memory platforms approaches on a higher level than multithreading work well. Both OpenMP and Cilk++ can already be used today and can provide good performance although they still require a lot of manual optimization. It is only a matter of time until the features found in these programming models are available in other programming languages than C++ and Fortran.

The problem is, that development will not stop at homogeneous multi-cores. In fact, specialized accelerator chips are already available. At least with OpenCL, a standardized interface will become available for them, but access to them is explicit and happens on a very low level. It still cannot be said whether it will be possible to integrate programming for accelerators into a parallel programming model on a high level.

Another problem will emerge when the limits of shared memory architectures are reached. If this is the case, a high-level distributed memory programming model will be necessary for distributed memory architectures to come to the mainstream. Without such a programming model, shared memory architectures would continue to be the standard, but as this would stop innovation, the CPU industry would become a replacement industry instead of an innovation industry.

6.1 About the tested programming models

The tested programming models can be split into two groups, namely, established programming models, which can be used in production environments, and experimental high-level programming models. Experimental high-level programming models aim at providing high performance and good scalability, which is realised using new high-level concepts. Although it is not sure yet whether these programming models will ever be used in productive environments, some of the

concepts will certainly be adopted by other programming models.

As shown in this work, X10's strengths can be found in the area of task-based parallelism and coordination. One such concept is X10's *rooted exception model*, where exceptions are propagated from child to parent tasks, which means that exceptions can never get lost. Another major contribution by X10 is the *clock* concept, which is able to completely replace barriers in modern parallel programming models. X10's regions and data distributions have the advantage of allowing static reasoning, which allows the application of set operations, such as intersections, to iteration spaces. Also, this concept is very likely to grant good performance for parallel operations. Besides, X10 provides extensions to object oriented programming to make it better suitable for parallel environments. These contributions will most certainly find their way into other object oriented programming languages.

The major problem of X10's domains and data distributions is the complexity of managing local and remote memory accesses. As such, the programmer has to explicitly specify whether a memory access references a local or a remote memory location, which increases code complexity. In fact, writing distributed memory code in X10 is still very low-level. Another concern with X10 is whether the concepts provided are flexible enough to support complex scenarios like irregular problems.

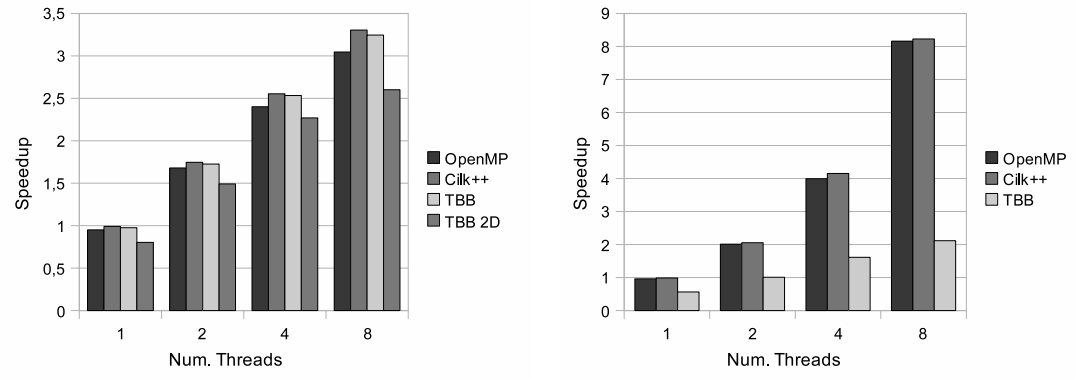
The complications associated with X10, however, cannot be found in Chapel: First, Chapel promises to provide arbitrary data distributions. In addition, Chapel abstracts away the difference between local and remote memory accesses. Chapel's array syntax allows to describe some algorithms on a higher level than classical sequential programming models do, which has the advantage of both making compiler optimizations easier to realize and reducing complexity for the programmer. In Chapel it is easy to derive work distribution from data distribution, though, where necessary, work distribution can also be specified manually. The question remaining open is whether Chapel's programming model has the potential of providing high performance. This question cannot be answered yet.

In terms of high-level concepts, Threading Building Blocks (TBB) is the programming model out of the tested established ones that comes closest to the experimental programming models. TBB's *range* class allows to define iteration spaces similar to those in Chapel's and X10's array domains. The range classes included in the TBB distribution provide only basic functionality, though, as custom ranges can be implemented by the user, it should be possible to create ranges for arbitrary problem classes. However, the problematic thing about ranges is that, unlike Chapel's domains and X10's regions, a range does not provide iterators. This means that the actual application of a range requires detailed understanding of the specific range's conceptual fundament, which leads to implementations on a lower level, compared to implementations using array domains.

The biggest disadvantage of TBB in comparison with the other evaluated programming models is its being just an object oriented C++ library. Due to this, code accessing TBB functionality is very bloated compared to other programming models and most of the code is just boilerplate code. Also, the fact that TBB is not a programming language becomes a

problem with regard to memory consistency models: TBB only provides memory consistency for operations on instances of the atomic template class.

Unlike TBB, OpenMP and Cilk++ provide only little support for data-parallelism and focus on task-based parallelism instead. However, this is compensated by their reduced complexity: Both programming models facilitate the parallelization of existing C++ code, with the parallelization requiring only little modifications of the source-code. Furthermore, both programming models allow to create a sequential C++ representation of a parallel program which can be used in code-analysis tools and for debugging. Though only Cilk++ guarantees that the sequential representation always has the same semantics as the parallel version, in most cases this should also apply to OpenMP programs. In general, since the release of OpenMP version 3.0, which supports tasks, Cilk++ and OpenMP have been very similar. The differences can mainly be found in the memory consistency models: Whereas Cilk++ tries to provide an intuitive model that is very relaxed, OpenMP's is more explicit, thus allowing more fine-grained control. Also, the performance behaviour may differ in some complex scenarios, which, however, are not in the scope of this work.



(a) ikj

(b) unrolled submatrix form

Figure 6.1: Comparison of the evaluated programming model's speedup for the matrix multiplication code on fleur - matrix size: 2048

When we look at the performance and scalability of OpenMP, Cilk++ and TBB, which can be seen in Figure 6.1, it can be noted that regarding the simple matrix multiplication example OpenMP and Cilk++ exhibit similar behaviour: Both programming models are able to achieve better than linear speedup, using the matrix multiplication code in the unrolled submatrix variant. With respect to the ikj-variant, which is easier to implement, neither OpenMP nor Cilk++ are able to achieve such results. TBB, though, behaves differently: Whereas it can deliver competitive performance for the ikj-variant of the code, for the unrolled submatrix variant the performance is actually worse than for the ikj-variant. The tests carried out for the ikj-variant using the *blocked_range2d* object, which are marked as *TBB 2D* in Figure 6.1(a), have not been able to provide better performance, either.

Based on this evaluation the following conclusions can be drawn:

1. Both OpenMP and Cilk++ are mature programming models, enabling the programmer to easily parallelize existing programs. Nevertheless, they are still very low-level and profit from manual optimization. OpenMP's advantage over Cilk++ is that it is an industry standard supported by most compiler-vendors. Cilk++, in turn, profits from a more sophisticated memory consistency model.

2. TBB tries to provide a more high-level approach, compared to OpenMP and Cilk++. However, its major design flaw is that it is only a C++ library, thus being not able to provide its own memory consistency model. Apart from that, compilers are not aware of TBB's parallelism, which reduces the potential for optimization. Finally, TBB requires a lot of boilerplate code, as a consequence of which code readability is reduced.

3. Chapel provides very interesting concepts for array operations. All the same, it cannot yet be said whether it will ever be able to provide competitive performance.

4. X10's innovation lies mainly in the area of task-based parallelism and coordination - especially the *clock* concept is very interesting and will almost certainly replace barriers in other programming models. Although X10's array concepts have the advantage of allowing set operations on array regions and data distributions, they are not as sophisticated as Chapel's: It is doubtful whether X10 arrays will be able to support any kind of problem.

	OpenMP	Cilk++	TBB	Chapel	X10
data distribution	unsupported	unsupported	only data access patterns	custom data distributions	supported
work distribution	task and data-parallel constructs	mainly task-parallel	mainly data-parallel	derived from data distribution or explicit	task and data-parallel
memory model	local view, synchronized with global view on sync ops	global view	global view	PGAS - mostly abstracted away	APGAS
memory consistency	weak ordering	dag-consistency	none, release consistency for atomic class	undecided	sequential if race-free, only local
coordination	barriers, mutexes, critical sections, taskwait	mutexes, sync	mutexes	sync variables, atomic blocks	finish, clocks, atomic blocks

Table 6.1: Overview over the evaluated programming models.

6.2 Promising concepts

The experimental programming models evaluated in this work lay the groundwork for future high-level programming models. There seems to be a consensus in the community that PGAS is the memory model future programming models will be built upon. The question, however, remains, whether it is necessary for the programmer to explicitly manage locality. The simplicity of Chapel's memory model from the programmers point of view would ease the adoption of distributed memory programming models. However, it is yet to be determined if this approach is scalable.

On the side of task management, X10's task management model, that builds upon Cilk's model and extends it with features like a rooted exception model, looks very promising. The *clock* concept is a good replacement for barriers as it works well together with tasks and has no disadvantages. It is less error-prone than mutex-based coordination, especially, as it can be guaranteed that a program using locks is deadlock-free. Clocks would be a good addition to existing task-based programming models.

Although clocks would reduce the need for the use of mutexes, they are still needed to secure critical sections. Another way to secure critical sections, which is less error-prone, is the use of transactions. Currently, transactional memory suffers from a high overhead, which makes it unattractive compared to locking techniques. Only time can tell if this problem will be solved.

The problem most relevant for scalability of a parallel program on distributed memory architectures is data distribution. Without a scalable high-level means of data distribution, distributed memory architectures will not be accepted in the mainstream. Both X10 and Chapel try to solve this problem in a different way. Chapel's concept is more elegant in that it supports arbitrary types of data distribution. The question, however, remains, whether this concept is scalable. X10 restricts itself to data distributions that allow static reasoning. Although this approach has a higher chance to scale well, it might not be accepted in the community if it is not able to support every type of problem.

6.3 Future developments

In the near future, chances are high that programming models for shared memory architectures will still be based on fork-join parallelism. They will differ from standard threading libraries by providing a task scheduler and a memory model for the programming language. A lot of productivity could already be gained today if some of the task management features of X10 were adapted to existing programming models.

For distributed memory architectures no new high-level programming model can be seen that might be able to replace MPI as the standard programming model. It seems that the PGAS model is a good starting point for developing a new programming model for homogeneous distributed memory systems, but PGAS is still a very low-level approach. Automatic data distribution still seems to be a good way to reduce the complexity of writing parallel programs,

but the difficulty is the necessity of providing arbitrary data distribution, while still maintaining good scalability.

The research regarding heterogeneous architectures is still at its beginning, and, for the time being, OpenCL is going to become the standard for accessing accelerator units. One concept that might be able to provide a higher-level approach for programming heterogeneous architectures is *stream programming*. Stream programming is especially interesting because it would even allow to work with accelerator hierarchies. Only time will show whether stream programming will be able to solve the challenges posed by heterogeneous architectures.

Although a lot of problems are still not solved on the software side, hardware development will further proceed in the direction of parallel computing. Although the number of transistors on processors still increases, they cannot be efficiently used to increase single processor performance. Therefore they are instead used to increase on-chip parallelism. Another paradigm change might occur when sometime in the future the cost per transistor still decreases, while the power consumption per transistor doesn't decrease any more. According to [Hofstee, 2009] performance can only be further improved on a fixed power budget, when efficiency is improved. And efficiency per computation can only be improved in two fundamental ways:

> "Running each thread slower or
> Specializing the core to the computation" [Hofstee, 2009]

As reducing single-thread performance will sooner or later limit scalability for most applications, the only remaining solution is to use cores specialized to the computation. General Purpose Graphics Processing Units (GPGPUs) are already used today to accelerate computations. The author believes that the functionality provided by GPGPUs will sooner or later be integrated into normal CPUs in the form of specialized cores, similar to the SPUs in the Cell processor. If we look at embedded processors, where power is more restricted than in desktop- and server-based computing, we can see that these processors contain a great variety of units specialized for specific tasks. This might also become necessary for standard CPUs.

Another way to increase efficiency per computation is through the use of FPGAs. With FPGAs, processing units specialized for a specific kernel can be created on the fly, and as soon as this unit is not needed any more, the FPGA can be reconfigured for another kernel. Of course, an FPGA cannot provide the same performance as a processor specialized for a specific computation, but the flexibility of reconfiguration might prove to be of advantage in the future.

Part I

Appendix

List of Figures

List of Tables

Bibliography

[Adve and Gharachorloo, 1996] Adve, S. V. and Gharachorloo, K. (1996). Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76.

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485. Reston, VA.

[Archibald and Baer, 1986] Archibald, J. and Baer, J.-L. (1986). Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298.

[ARMCI, 2009] (2009). ARMCI - Aggregate Remote Memory Copy Interface. Website. http://www.emsl.pnl.gov/docs/parsoft/armci/, (2009.03.19).

[Asanovic et al., 2006] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[Balaji et al., 2009] Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E., Thakur, R., and Träff, J. L. (2009). MPI on a Million Processors. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 20–30, Berlin, Heidelberg. Springer-Verlag.

[Barney, 2009] Barney, B. (2009). POSIX Threads Programming. Online Source. https://computing.llnl.gov/tutorials/pthreads/, (2009.03.19).

[Bell et al., 2006] Bell, C., Bonachea, D., Nishtala, R., and Yelick, K. (2006). Optimizing bandwidth limited problems using one-sided communication and overlap. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 10.

[Benkner, 1999] Benkner, S. (1999). Optimizing Irregular HPF Applications using Halos. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 1015–1024, London, UK. Springer-Verlag.

[Benkner and Zima, 1999] Benkner, S. and Zima, H. (1999). Compiling high performance Fortran for distributed-memory architectures. *Parallel Computing*, 25(13-14):1785–1825.

[Blas, 2008] (2008). BLAS Website. http://www.netlib.org/blas/, (2009.04.21).

[Blumofe et al., 1996] Blumofe, R. D., Frigo, M., Joerg, C. F., Leiserson, C. E., and Randall, K. H. (1996). Dag-Consistent Distributed Shared Memory. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 132–141, Washington, DC, USA. IEEE Computer Society.

[Blumofe and Leiserson, 1994] Blumofe, R. D. and Leiserson, C. E. (1994). Scheduling multithreaded computations by work stealing. pages 356–368.

[Blythe, 2008] Blythe, D. (May 2008). Rise of the Graphics Processor. *Proceedings of the IEEE*, 96(5):761–778.

[Callahan et al., 2004] Callahan, D., Chamberlain, B. L., and Zima, H. P. (2004). The Cascade High Productivity Language. In *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pages 52–60, Los Alamitos, CA, USA. IEEE Computer Society.

[Chamberlain et al., 2007] Chamberlain, B. L., Callahan, D., and Zima, H. P. (2007). Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312.

[Chandra et al., 1997] Chandra, R., Chen, D.-K., Cox, R., Maydan, D. E., Nedeljkovic, N., and Anderson, J. M. (1997). Data distribution support on distributed shared memory multiprocessors. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 334–345, New York, NY, USA. ACM.

[Chapel, 2009a] (2009a). Chapel Language Specification 0.780. Technical report, Cray Inc.

[Chapel, 2009b] (2009b). Chapel: The Cascade high productivity language. http://chapel.cs.washington.edu, (2009.03.19).

[Chapman et al., 1992] Chapman, B. M., Mehrotra, P., and Zima, H. P. (1992). Vienna Fortran—a Fortran language extension for distributed memory multiprocessors. pages 39–62.

[Charles et al., 2005] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA. ACM.

[Cilk, 2009a] (2009a). *Cilk++ Programmers Guide*. Cilk Arts, Inc.

[Cilk, 2009b] (2009b). Cilk++ Webpage. Cilk Arts Inc. http://www.cilk.com/, (2009.03.18).

[Co-Array, 2009] (2009). Co-Array Fortran. Website. http://www.co-array.org/, (2009.03.19).

[Codeplay, 2010] (2010). Codeplay Offload. http://offload.codeplay.com/, (2010.01.13).

[Denning, 2005] Denning, P. J. (2005). The locality principle. *Commun. ACM*, 48(7):19–24.

[Diaconescu and Zima, 2007] Diaconescu, R. E. and Zima, H. P. (2007). An Approach To Data Distributions in Chapel. *Int. J. High Perform. Comput. Appl.*, 21(3):313–335.

[Dijkstra, 1965] Dijkstra, E. W. (1965). Cooperating Sequential Processes, Technical Report EWD-123. Technical report. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html, (2010.01.13).

[Dijkstra, nd] Dijkstra, E. W. (n.d.). Hierarchical ordering of sequential processes. In *Operating Systems Techniques* , pages 72–93.

[Flynn, 1972] Flynn, M. J. (1972). Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960.

[Frigo, 1998] Frigo, M. (1998). The Weakest Reasonable Memory Model. Master's thesis, MIT Department of Electrical Engineering and Computer Science.

[Frigo, 1999] Frigo, M. (1999). *Portable High-Performance Programs*. PhD thesis, MIT Department of Electrical Engineering and Computer Science.

[Frigo et al., 1998] Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[GASNet, 2009] (2009). GASNet. Website. http://gasnet.cs.berkeley.edu/, (2009.03.19).

[Ghuloum et al., 2007a] Ghuloum, A., Smith, T., Wu, G., Zhou, X., Fang, J., Guo, P., So, B., Rajagopalan, M., Chen, Y., and Chen, B. (2007a). Future-Proof Data Parallel Algorithms and Software on Intel® Multi-Core Architecture. *Intel® Technology Journal*, 11(4).

[Ghuloum et al., 2007b] Ghuloum, A., Sprangle, E., Fang, J., Wu, G., and Zhou, X. (2007b). Ct: A Flexible Parallel Programming Model for Tera-scale Architectures. http://techresearch.intel.com/UserFiles/en-us/File/terascale/Whitepaper-Ct.pdf, (2009.03.19).

[Gosling et al., 2005] Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional.

[Gray and Reuter, 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

[Gummaraju and Rosenblum, 2005] Gummaraju, J. and Rosenblum, M. (2005). Stream Programming on General-Purpose Processors.

[Gustafson, 1988] Gustafson, J. L. (1988). Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533.

[Herlihy et al., 2003] Herlihy, M., Luchangco, V., and Moir, M. (2003). Obstruction-free synchronization: double-ended queues as an example. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 522–529.

[Hilfinger et al., 2001] Hilfinger, P. N., Bonachea, D., Gay, D., Graham, S., Liblit, B., Pike, G., and Yelick, K. (2001). Titanium Language Reference Manual. Technical report, Berkeley, CA, USA.

[HMPP, 2010] (2010). HMPP Workbench. http://www.caps-entreprise.com/hmpp.html, (2010.01.13).

[Hofstee, 2009] Hofstee, H. P. (2009). Heterogeneous Multi-core Processors: The Cell Broadband Engine. In Keckler, S. W., Olukotun, K., and Hofstee, H. P., editors, *Multicore Processors and Systems*. Springer Publishing Company, Incorporated.

[HPF, 1997] (1997). High Performance Fortran Language Specification. Technical report, High Performance Fortran Forum.

[Java, 2009] (2009). Lesson: Concurrency (The Java Tutorials - Essential Classes). Sun Microsystems, Inc. Online Source. http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html, (2009.03.19).

[Kennedy et al., 2007] Kennedy, K., Koelbel, C., and Zima, H. (2007). The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proceedings of the Proc.Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, San Diego, California.

[Koelbel and Mehrotra, 1991] Koelbel, C. and Mehrotra, P. (1991). Programming data parallel algorithms on distributed memory using Kali. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 414–423, New York, NY, USA. ACM.

[Kumar et al., 2007] Kumar, A., Senthilkumar, G., Krishna, M., Jayam, N., Baruah, P. K., Sharma, R., Srinivasan, A., and Kapoor, S. (2007). A Buffered-Mode MPI Implementation for the Cell BETM Processor. In *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part I*, pages 603–610, Berlin, Heidelberg. Springer-Verlag.

[Larus and Kozyrakis, 2008] Larus, J. and Kozyrakis, C. (2008). Transactional memory. *Commun. ACM*, 51(7):80–88.

[Lea, 2009] Lea, D. (2009). The JSR-133 Cookbook for Compiler Writers. http://gee.cs.oswego.edu/dl/jmm/cookbook.html, (2009.03.19).

[Lee, 2006] Lee, E. A. (2006). The problem with threads. *Computer*, 39(5):33–42.

[Leiserson and Mirman, 2008] Leiserson, C. W. and Mirman, I. B. (2008). How to Survive the Multicore Software Revolution (or at Least Survive the Hype). Online Source. http://www.cilk.com/ebook/download5643, (2009.03.19).

[Loveman, 1993] Loveman, D. B. (1993). High performance Fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):25–42.

[Massingill et al., 2005] Massingill, B. L., Mattson, T. G., and Sanders, B. A. (2005). *Patterns for parallel application programs*. The Software Pattern Series. Addison-Wesley.

[Mehrotra et al., 2002] Mehrotra, P., Van Rosendale, J., and Zima, H. (2002). High Performance Fortran - History, Status and Future. In *ISHPC '02: Proceedings of the 4th International Symposium on High Performance Computing*, page 490, London, UK. Springer-Verlag.

[Microsoft, 2009] (2009). Processes and Threads (Windows). Microsoft. Online Source. http://msdn.microsoft.com/en-us/library/ms684841(VS.85).aspx, (2009.03.19).

[Monteyne, 2008] Monteyne, M. (2008). Rapidmind Whitepaper. Online Source. http://www.rapidmind.com/pdfs/WP_RapidMindPlatform.pdf, (2009.03.19).

[Moore, 1965] Moore, G. E. (1965). Cramming More Components Onto Integrated Circuits. Electronics.

[Netzer and Miller, 1992] Netzer, R. H. B. and Miller, B. P. (1992). What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88.

[Nikolopoulos et al., 2000] Nikolopoulos, D. S., Papatheodorou, T. S., Polychronopoulos, C. D., Labarta, J., and Ayguadé, E. (2000). Is data distribution necessary in OpenMP? In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 47, Washington, DC, USA. IEEE Computer Society.

[Numrich and Reid, 1998] Numrich, R. W. and Reid, J. (1998). Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31.

[OpenCL, 2009] (2009). The OpenCL Specification. Technical report, Khronos OpenCL Working Group.

[OpenMP, 2008] (2008). OpenMP Application Program Interface. Technical report, OpenMP Architecture Review Board.

[Patterson and Hennessy, 2008] Patterson, D. A. and Hennessy, J. L. (2008). *Computer Organization and Design: The Hardware/Software Interface, Fourth Edition*. Morgan Kaufmann, 4 edition.

[Petit et al., 2006] Petit, E., Bodin, F., Papaure, G., and Dru, F. (2006). ASTEX: a hot path based thread extractor for distributed memory system on a chip. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 141, New York, NY, USA. ACM.

[Quinn, 2003] Quinn, M. J. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group.

[Randall, 1998] Randall, K. H. (1998). *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

[Reinders, 2007] Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly Media, Inc., First Edition edition.

[Saraswat and Nystrom, 2008] Saraswat, V. and Nystrom, N. (2008). Report on the Experimental Language X10. Technical Report Version 1.7.

[Saraswat et al., 2007] Saraswat, V. A., Jagadeesan, R., Michael, M., and von Praun, C. (2007). A theory of memory models. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 161–172, New York, NY, USA. ACM.

[Stream, 2009] (2009). STREAM Benchmark. http://www.cs.virginia.edu/stream/, (2009.06.18).

[Supercomputing, 2001] (2001). *Cilk 5.4.6 Reference Manual*. Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. http://supertech.lcs.mit.edu/cilk/manual-5.4.6.pdf.

[Titanium, 2009] (2009). Titanium Project. Website. http://titanium.cs.berkeley.edu/, (2009.03.19).

[Top, 2009] (2009). Architecture Share Over Time - Top500 Supercomputing Sites. Top500.org. Website. http://www.top500.org/overtime/list/32/archtype, (2009.04.21).

[Tullsen et al., 1995] Tullsen, D. M., Eggers, S. J., and Levy, H. M. (1995). Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403, New York, NY, USA. ACM.

[UPC, 2005] (2005). UPC language specifications, v1.2. Technical report, UPC Consortium.

[UPC, 2009] (2009). The Berkeley UPC Compiler. Website. http://upc.lbl.gov/, (2009.03.19).

[Willhalm and Popovici, 2008] Willhalm, T. and Popovici, N. (2008). Putting intel threading building blocks to work. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 3–4, New York, NY, USA. ACM.

[Wolfe, 1995] Wolfe, M. J. (1995). *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Wulf and McKee, 1995] Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24.

[X10, 2009] (2009). X10. Website. http://x10.codehaus.org/, (2009.03.19).

[Yelick et al., 2007] Yelick, K., Bonachea, D., Chen, W.-Y., Colella, P., Datta, K., Duell, J., Graham, S. L., Hargrove, P., Hilfinger, P., Husbands, P., Iancu, C., Kamil, A., Nishtala, R., Su, J., Welcome, M., and Wen, T. (2007). Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA. ACM.

Sourcecode

Listing 1: The Makefile used to make and run the sequential reference implementation.

```
LIBS =

TARGET =  MatMul

OBJS =

include config/make.def


$(TARGET):  $(OBJS)
  $(CXX) $(CXXFLAGS) -o bin/$(TARGET) MatMul.cpp $(OBJS) $(LIBS)

all: ijk ikj submatrix submatrix_unroll

clean:
  rm -f $(OBJS) $(TARGET)
  rm -f lib/*
  rm -f bin/*
  rm -f results/*

ijk:
  ./ijk_suite "$(DIMENSIONS)" "$(RUNS)" "$(DATA_TYPE)"

results/ijk:
  ./ijk_run "$(DIMENSIONS)" "$(RUNS)" "$(DATA_TYPE)"

ikj:
  ./ikj_suite "$(DIMENSIONS)" "$(RUNS)" "$(DATA_TYPE)"

results/ikj:
  ./ikj_run "$(DIMENSIONS)" "$(RUNS)" "$(DATA_TYPE)"

submatrix:
  ./submatrix_suite "$(DIMENSIONS)" "$(RUNS)" "$(DATA_TYPE)" "$(SUBMATRIX_STRIDE
      )"

submatrix_unroll:
  ./submatrix_unroll_suite "$(DIMENSIONS)" "$(RUNS)" "$(DATA_TYPE)" "$(
      SUBMATRIX_STRIDE)"

results/submatrix:
  ./submatrix_run "$(DIMENSIONS)" "$(RUNS)" "$(DATA_TYPE)" "$(SUBMATRIX_STRIDE)"

results/submatrix_unroll:
  ./submatrix_unroll_run "$(DIMENSIONS)" "$(RUNS)" "$(DATA_TYPE)" "$(
      SUBMATRIX_STRIDE)"
```

```
44
45 run: results/all
46 results/all: results/ijk results/ikj results/submatrix results/submatrix_unroll
47   ./gather_results.pl
```

Listing 2: The submatrix_suite script of the sequential reference implementation which is used to make a target for each combination of parameters.

```
1  for i in $1 # dimensions
2  do
3    for j in $4 # submatrix stride
4    do
5      if [ "$j" -le "$i" ]
6      then
7        echo "#define DIMENSIONS $i
8  #define RUNS $2
9  #define DATA_TYPE $3
10 #define DATA_TYPE_NAME  \"$3\"
11 #define METHOD    \"submatrix\"
12 #define SUBMATRIX_STRIDE  $j
13 #define RUN   runSubmatrix" > parameters.h
14 make TARGET=submatrix.$3.dim$i.stride$j
15     fi
16   done
17 done
```

Listing 3: The submatrix_run script of the sequential reference implementation which is used to run each variant of the submatrix multiplication.

```
1  for i in $1 # dimensions
2  do
3    for j in $4 # submatrix stride
4    do
5      if [ "$j" -le "$i" ]
6      then
7        bin/submatrix.$3.dim$i.stride$j | tee results/submatrix.$3.dim$i.stride$j
8      fi
9    done
10 done
```

Listing 4: Sequential reference implementation of the matrix multiplication code. (see Section 5.2)

```
1  //============================================================================
2  // Name        : MatMul.cpp
3  // Description : Matrix multiplication. To be consistent with literature,
4  //               column major order is used.
5  //============================================================================
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <sys/time.h>
10 #include <string.h>
11 #include <algorithm>
12 #include "parameters.h"
13
14 using namespace std;
15
16 #define VERIFY
17
18 DATA_TYPE mat1[DIMENSIONS * DIMENSIONS];
```

```
19 DATA_TYPE mat2[DIMENSIONS * DIMENSIONS];
20 DATA_TYPE mat3[DIMENSIONS * DIMENSIONS];
21
22 inline void clearC()
23 {
24   memset(mat3, 0, sizeof(DATA_TYPE) * DIMENSIONS * DIMENSIONS);
25 }
26
27 double timing(struct timeval start, struct timeval stop)
28 {
29   double dStart = 1.0e-6 * start.tv_usec;
30   double dStop = (stop.tv_sec - start.tv_sec) + 1.0e-6 * stop.tv_usec;
31
32   return dStop - dStart;
33 }
34
35 // Prints out the matrices if the parameter "SHOW_MATRICES" is set. (For
       debugging purposes.)
36 void printMatrix(const char name[], DATA_TYPE * mat)
37 {
38 #ifdef SHOW_MATRICES
39   printf("\nMatrix: %s\n", name);
40   for(int j = 0; j < DIMENSIONS; j++)
41   {
42     for(int i = 0; i < DIMENSIONS; i++)
43     {
44       printf("%d\t", (int)mat[j*DIMENSIONS + i]);
45     }
46     printf("\n");
47   }
48 #endif
49 }
50
51 // Generates the matrices.
52 void generateMatrices()
53 {
54   for(int j = 0; j < DIMENSIONS; j++)
55   {
56     for(int i = 0; i < DIMENSIONS; i++)
57     {
58       mat1[i + j * DIMENSIONS] = i + j;
59       if(abs(i - j) == 1)
60       {
61         mat2[j * DIMENSIONS + i] = 1;
62       }
63       else
64       {
65         mat2[j * DIMENSIONS + i] = 0;
66       }
67       mat3[j * DIMENSIONS + i] = 0;
68     }
69   }
70 }
71
72 // Runs a verification of the results using a sequential ijk version of the
       matrix multiplication.
73 #ifdef VERIFY
74 DATA_TYPE mat4[DIMENSIONS * DIMENSIONS];
75 inline void verifyData()
76 {
```

```
77   for(int i = 0; i < DIMENSIONS; i++)
78     for(int k = 0; k < DIMENSIONS; k++)
79       for(int j = 0; j < DIMENSIONS; j++)
80         mat4[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
               ];
81   printMatrix("C (verify)", mat4);
82   for(int i = 0; i < DIMENSIONS; i++)
83   {
84     for(int j = 0; j < DIMENSIONS; j++)
85     {
86       if(mat3[i*DIMENSIONS + j] != mat4[i*DIMENSIONS + j])
87       {
88         printf("Correct:\t\t0\n");
89         return;
90       }
91     }
92   }
93   printf("Correct:\t\t1\n");
94 }
95 #endif
96
97 // Ijk variant of the multiplication.
98 inline double runIjk(const bool verify)
99 {
100   struct timeval start, stop;
101
102   gettimeofday(&start, NULL);
103
104   for(int i = 0; i < DIMENSIONS; i++)
105     for(int j = 0; j < DIMENSIONS; j++)
106       for(int k = 0; k < DIMENSIONS; k++)
107         mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
               ];
108   gettimeofday(&stop, NULL);
109   printMatrix("C (ijk)", mat3);
110 #ifdef VERIFY
111   if(verify)
112   {
113     verifyData();
114   }
115 #endif
116   clearC();
117
118   return timing(start, stop);
119 }
120
121 // Ikj variant of the multiplication.
122 inline double runIkj(const bool verify)
123 {
124   struct timeval start, stop;
125
126   gettimeofday(&start, NULL);
127
128   for(int i = 0; i < DIMENSIONS; i++)
129     for(int k = 0; k < DIMENSIONS; k++)
130       for(int j = 0; j < DIMENSIONS; j++)
131         mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
               ];
132   gettimeofday(&stop, NULL);
133   printMatrix("C (ikj)", mat3);
```

```
134 #ifdef VERIFY
135   if(verify)
136   {
137     verifyData();
138   }
139 #endif
140   clearC();
141 
142   return timing(start, stop);
143 }
144 
145 // Submatrix variant of the matrix multiplication.
146 #ifdef SUBMATRIX_STRIDE
147 inline double runSubmatrix(const bool verify)
148 {
149   struct timeval start, stop;
150 
151   gettimeofday(&start, NULL);
152 
153   for(int it = 0; it < DIMENSIONS; it+=SUBMATRIX_STRIDE)
154     for(int kt = 0; kt < DIMENSIONS; kt+=SUBMATRIX_STRIDE)
155       for(int jt = 0; jt < DIMENSIONS; jt+=SUBMATRIX_STRIDE)
156         for(int i = it; i < (it + SUBMATRIX_STRIDE) && i < DIMENSIONS; i++)
157           for(int k = kt; k < (kt + SUBMATRIX_STRIDE) && k < DIMENSIONS; k++)
158             for(int j = jt; j < (jt + SUBMATRIX_STRIDE) && j < DIMENSIONS; j++)
159               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
160   gettimeofday(&stop, NULL);
161   printMatrix("C (submatrix)", mat3);
162 #ifdef VERIFY
163   if(verify)
164   {
165     verifyData();
166   }
167 #endif
168   clearC();
169 
170   return timing(start, stop);
171 }
172 
173 // Unrolled submatrix variant of the matrix multiplication.
174 inline double runSubmatrixUnroll(const bool verify)
175 {
176   struct timeval start, stop;
177 
178   gettimeofday(&start, NULL);
179 
180   int dim = DIMENSIONS - (DIMENSIONS % SUBMATRIX_STRIDE);
181 
182   // i, j, k
183   for(int it = 0; it < dim; it+=SUBMATRIX_STRIDE)
184     for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
185       for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
186         for(int i = it; i < it + SUBMATRIX_STRIDE; i++)
187           for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
188             for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
189               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
190 
191   if(DIMENSIONS != dim)
```

```
192   {
193     // -i, j, k
194     for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
195       for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
196         for(int i = dim; i < DIMENSIONS; i++)
197           for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
198             for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
199               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
200
201     // -i, j, -k
202     for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
203       for(int i = dim; i < DIMENSIONS; i++)
204         for(int k = dim; k < DIMENSIONS; k++)
205           for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
206             mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS
                    + j];
207
208     // -i, -j, -k
209     for(int i = dim; i < DIMENSIONS; i++)
210       for(int k = dim; k < DIMENSIONS; k++)
211         for(int j = dim; j < DIMENSIONS; j++)
212           mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS +
                  j];
213
214     // i, j, -k
215     for(int it = 0; it < dim; it+=SUBMATRIX_STRIDE)
216       for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
217         for(int i = it; i < it + SUBMATRIX_STRIDE; i++)
218           for(int k = dim; k < DIMENSIONS; k++)
219             for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
220               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
221
222     // i, -j, -k
223     for(int it = 0; it < dim; it+=SUBMATRIX_STRIDE)
224       for(int i = it; i < it + SUBMATRIX_STRIDE; i++)
225         for(int k = dim; k < DIMENSIONS; k++)
226           for(int j = dim; j < DIMENSIONS; j++)
227             mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS
                    + j];
228
229     // i, -j, k
230     for(int it = 0; it < dim; it+=SUBMATRIX_STRIDE)
231       for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
232         for(int i = it; i < it + SUBMATRIX_STRIDE; i++)
233           for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
234             for(int j = dim; j < DIMENSIONS; j++)
235               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
236
237     // -i, -j, k
238     for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
239       for(int i = dim; i < DIMENSIONS; i++)
240         for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
241           for(int j = dim; j < DIMENSIONS; j++)
242             mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS
                    + j];
243   }
244   gettimeofday(&stop, NULL);
```

```
245   printMatrix("C (submatrix_unroll)", mat3);
246 #ifdef VERIFY
247   if(verify)
248   {
249     verifyData();
250   }
251 #endif
252   clearC();
253 
254   return timing(start, stop);
255 }
256 #endif
257 
258 // Drops the highest and lowest timing and calculates the average of the rest.
259 double getAvg(double *timings)
260 {
261   double avg = 0.0;
262   double min, max;
263 
264   if(timings[0] > timings[1])
265   {
266     max = timings[0];
267     min = timings[1];
268   }
269   else
270   {
271     min = timings[0];
272     max = timings[1];
273   }
274 
275   for(int i = 2; i < RUNS; i++)
276   {
277     if(timings[i] > max)
278     {
279       avg += max;
280       max = timings[i];
281     }
282     else if(timings[i] < min)
283     {
284       avg += min;
285       min = timings[i];
286     }
287     else
288     {
289       avg += timings[i];
290     }
291   }
292   return avg / (RUNS - 2);
293 }
294 
295 int main(void) {
296   generateMatrices();
297 
298   printf("Method:\t\t\t%s\n", METHOD);
299   printf("Dimensions:\t\t%d\n", DIMENSIONS);
300   printf("Num. Threads:\t\t%d\n", 1);
301   printf("DataType:\t\t%s\n", DATA_TYPE_NAME);
302   printf("Grainsize:\t\t%d\n", DIMENSIONS);
303   printf("Runs:\t\t\t%d\n", RUNS);
304 #ifdef SUBMATRIX_STRIDE
```

```
305   printf("Submatrix Stride:\t%d\n", SUBMATRIX_STRIDE);
306 #endif
307   printMatrix("A", mat1);
308   printMatrix("B", mat2);
309
310   double timings[RUNS];
311
312   for(int i = 0; i < RUNS; i++)
313   {
314     timings[i] = RUN(false);
315   }
316   printf("Avg. Time:\t\t%f\n", getAvg(timings));
317 #ifdef VERIFY
318   RUN(true);
319 #endif
320 }
```

Listing 5: OpenMP implementation of the matrix multiplication code. (see Sections 4.1.2 and 5.5.1)

```
1  //============================================================================
2  // Name        : MatMul.cpp
3  // Description : Matrix multiplication arallelized using OpenMP. To be
4  //           consistent with literature, column major order is used.
5  //============================================================================
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <sys/time.h>
10 #include <string.h>
11 #include <algorithm>
12 #include <omp.h>
13
14 #include "parameters.h"
15
16 using namespace std;
17
18 #define VERIFY
19
20
21 DATA_TYPE mat1[DIMENSIONS * DIMENSIONS];
22 DATA_TYPE mat2[DIMENSIONS * DIMENSIONS];
23 DATA_TYPE mat3[DIMENSIONS * DIMENSIONS];
24
25 inline void clearC()
26 {
27   memset(mat3, 0, sizeof(DATA_TYPE) * DIMENSIONS * DIMENSIONS);
28 }
29
30 double timing(struct timeval start, struct timeval stop)
31 {
32   double dStart = 1.0e-6 * start.tv_usec;
33   double dStop = (stop.tv_sec - start.tv_sec) + 1.0e-6 * stop.tv_usec;
34
35   return dStop - dStart;
36 }
37
38 // Prints out the matrices if the parameter "SHOW_MATRICES" is set. (For
       debugging purposes.)
39 void printMatrix(const char name[], DATA_TYPE * mat)
```

```
40 {
41 #ifdef SHOW_MATRICES
42   printf("\nMatrix: %s\n", name);
43   for(int j = 0; j < DIMENSIONS; j++)
44   {
45     for(int i = 0; i < DIMENSIONS; i++)
46     {
47       printf("%d\t", (int)mat[j*DIMENSIONS + i]);
48     }
49     printf("\n");
50   }
51 #endif
52 }
53
54 // Generates the matrices.
55 void generateMatrices()
56 {
57   for(int j = 0; j < DIMENSIONS; j++)
58   {
59     for(int i = 0; i < DIMENSIONS; i++)
60     {
61       mat1[i + j * DIMENSIONS] = i + j;
62       if(abs(i - j) == 1)
63       {
64         mat2[j * DIMENSIONS + i] = 1;
65       }
66       else
67       {
68         mat2[j * DIMENSIONS + i] = 0;
69       }
70       mat3[j * DIMENSIONS + i] = 0;
71     }
72   }
73 }
74
75 // Runs a verification of the results using a sequential ijk version of the
       matrix multiplication.
76 #ifdef VERIFY
77 DATA_TYPE mat4[DIMENSIONS * DIMENSIONS];
78 inline void verifyData()
79 {
80   // Uses a sequential ijk version for verification.
81   for(int i = 0; i < DIMENSIONS; i++)
82     for(int k = 0; k < DIMENSIONS; k++)
83       for(int j = 0; j < DIMENSIONS; j++)
84         mat4[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
              ];
85   printMatrix("C (verify)", mat4);
86   for(int i = 0; i < DIMENSIONS; i++)
87   {
88     for(int j = 0; j < DIMENSIONS; j++)
89     {
90       if(mat3[i*DIMENSIONS + j] != mat4[i*DIMENSIONS + j])
91       {
92         printf("Correct:\t\t0\n");
93         return;
94       }
95     }
96   }
97   printf("Correct:\t\t1\n");
```

```
98 }
99 #endif
100
101 // Ijk variant of the multiplication.
102 inline double runIjk(const bool verify)
103 {
104   struct timeval start, stop;
105
106   gettimeofday(&start, NULL);
107
108 #pragma omp parallel for
109   for(int i = 0; i < DIMENSIONS; i++)
110     for(int j = 0; j < DIMENSIONS; j++)
111       for(int k = 0; k < DIMENSIONS; k++)
112         mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
              ];
113   gettimeofday(&stop, NULL);
114   printMatrix("C (ijk)", mat3);
115 #ifdef VERIFY
116   if(verify)
117   {
118     verifyData();
119   }
120 #endif
121   clearC();
122
123   return timing(start, stop);
124 }
125
126 // Ikj variant of the multiplication.
127 inline double runIkj(const bool verify)
128 {
129   struct timeval start, stop;
130
131   gettimeofday(&start, NULL);
132
133 #pragma omp parallel for
134   for(int i = 0; i < DIMENSIONS; i++)
135     for(int k = 0; k < DIMENSIONS; k++)
136       for(int j = 0; j < DIMENSIONS; j++)
137         mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
              ];
138   gettimeofday(&stop, NULL);
139   printMatrix("C (ikj)", mat3);
140 #ifdef VERIFY
141   if(verify)
142   {
143     verifyData();
144   }
145 #endif
146   clearC();
147
148   return timing(start, stop);
149 }
150
151 // Submatrix variant of the matrix multiplication.
152 #ifdef SUBMATRIX_STRIDE
153 inline double runSubmatrix(const bool verify)
154 {
155   struct timeval start, stop;
```

```
156
157   gettimeofday(&start, NULL);
158
159 #pragma omp parallel for
160   for(int it = 0; it < DIMENSIONS; it+=SUBMATRIX_STRIDE)
161     for(int kt = 0; kt < DIMENSIONS; kt+=SUBMATRIX_STRIDE)
162       for(int jt = 0; jt < DIMENSIONS; jt+=SUBMATRIX_STRIDE)
163         for(int i = it; i < (it + SUBMATRIX_STRIDE) && i < DIMENSIONS; i++)
164           for(int k = kt; k < (kt + SUBMATRIX_STRIDE) && k < DIMENSIONS; k++)
165             for(int j = jt; j < (jt + SUBMATRIX_STRIDE) && j < DIMENSIONS; j++)
166               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
167   gettimeofday(&stop, NULL);
168   printMatrix("C (submatrix)", mat3);
169 #ifdef VERIFY
170   if(verify)
171   {
172     verifyData();
173   }
174 #endif
175   clearC();
176
177   return timing(start, stop);
178 }
179
180 // Unrolled submatrix variant of the matrix multiplication.
181 inline double runSubmatrixUnroll(const bool verify)
182 {
183   struct timeval start, stop;
184
185   gettimeofday(&start, NULL);
186
187   int dim = DIMENSIONS - (DIMENSIONS % SUBMATRIX_STRIDE);
188
189   // i, j, k
190 #pragma omp parallel for
191   for(int it = 0; it < dim; it+=SUBMATRIX_STRIDE)
192     for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
193       for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
194         for(int i = it; i < it + SUBMATRIX_STRIDE; i++)
195           for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
196             for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
197               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
198
199   // -i, j, k
200   for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
201 #pragma omp parallel for
202     for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
203       for(int i = dim; i < DIMENSIONS; i++)
204         for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
205           for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
206             mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS
                    + j];
207
208   // -i, j, -k
209 #pragma omp parallel for
210   for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
211     for(int i = dim; i < DIMENSIONS; i++)
212       for(int k = dim; k < DIMENSIONS; k++)
```

```
213         for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
214           mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS +
                j];
215
216   // -i, -j, -k
217 #pragma omp parallel for
218   for(int i = dim; i < DIMENSIONS; i++)
219     for(int k = dim; k < DIMENSIONS; k++)
220       for(int j = dim; j < DIMENSIONS; j++)
221         mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
              ];
222
223   // i, j, -k
224 #pragma omp parallel for
225   for(int it = 0; it < dim; it+=SUBMATRIX_STRIDE)
226     for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
227       for(int i = it; i < it + SUBMATRIX_STRIDE; i++)
228         for(int k = dim; k < DIMENSIONS; k++)
229           for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
230             mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS
                  + j];
231
232   // i, -j, -k
233 #pragma omp parallel for
234   for(int it = 0; it < dim; it+=SUBMATRIX_STRIDE)
235     for(int i = it; i < it + SUBMATRIX_STRIDE; i++)
236       for(int k = dim; k < DIMENSIONS; k++)
237         for(int j = dim; j < DIMENSIONS; j++)
238           mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS +
                j];
239
240   // i, -j, k
241 #pragma omp parallel for
242   for(int it = 0; it < dim; it+=SUBMATRIX_STRIDE)
243     for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
244       for(int i = it; i < it + SUBMATRIX_STRIDE; i++)
245         for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
246           for(int j = dim; j < DIMENSIONS; j++)
247             mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS
                  + j];
248
249   // -i, -j, k
250   for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
251 #pragma omp parallel for
252     for(int i = dim; i < DIMENSIONS; i++)
253       for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
254         for(int j = dim; j < DIMENSIONS; j++)
255           mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS +
                j];
256
257
258   gettimeofday(&stop, NULL);
259   printMatrix("C (submatrix_unroll)", mat3);
260 #ifdef VERIFY
261   if(verify)
262   {
263     verifyData();
264   }
265 #endif
266   clearC();
```

```
267
268   return timing(start, stop);
269 }
270 #endif
271
272 // Drops the highest and lowest timing and calculates the average of the rest.
273 double getAvg(double *timings)
274 {
275   double avg = 0.0;
276   double min, max;
277
278   if(timings[0] > timings[1])
279   {
280     max = timings[0];
281     min = timings[1];
282   }
283   else
284   {
285     min = timings[0];
286     max = timings[1];
287   }
288
289   for(int i = 2; i < RUNS; i++)
290   {
291     if(timings[i] > max)
292     {
293       avg += max;
294       max = timings[i];
295     }
296     else if(timings[i] < min)
297     {
298       avg += min;
299       min = timings[i];
300     }
301     else
302     {
303       avg += timings[i];
304     }
305   }
306   return avg / (RUNS - 2);
307 }
308
309 int main(void) {
310   generateMatrices();
311
312   omp_set_num_threads(NUM_THREADS);
313
314   printf("Method:\t\t\t%s\n", METHOD);
315   printf("Dimensions:\t\t%d\n", DIMENSIONS);
316   printf("Num. Threads:\t\t%d\n", NUM_THREADS);
317   printf("DataType:\t\t%s\n", DATA_TYPE_NAME);
318   printf("Grainsize:\t\t%d\n", DIMENSIONS);
319   printf("Runs:\t\t\t%d\n", RUNS);
320 #ifdef SUBMATRIX_STRIDE
321   printf("Submatrix Stride:\t%d\n", SUBMATRIX_STRIDE);
322 #endif
323   printMatrix("A", mat1);
324   printMatrix("B", mat2);
325
326   double timings[RUNS];
```

```
327
328   for(int i = 0; i < RUNS; i++)
329   {
330     timings[i] = RUN(false);
331   }
332   printf("Avg. Time:\t\t%f\n", getAvg(timings));
333 #ifdef VERIFY
334   RUN(true);
335 #endif
336 }
```

Listing 6: Cilk++ implementation of the matrix multiplication code. (see Sections 4.1.3 and 5.5.3)

```
1  //===================================================================================
2  // Name        : MatMul.cpp
3  // Description : Matrix multiplication parallelized using Cilk++. To be
4  //           consistent with literature, column major order is used.
5  //===================================================================================
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <sys/time.h>
10 #include <string.h>
11 #include "parameters.h"
12 #include <cilk.h>
13
14 using namespace std;
15
16 #define VERIFY
17
18 DATA_TYPE mat1[DIMENSIONS * DIMENSIONS];
19 DATA_TYPE mat2[DIMENSIONS * DIMENSIONS];
20 DATA_TYPE mat3[DIMENSIONS * DIMENSIONS];
21
22 inline void clearC()
23 {
24   memset(mat3, 0, sizeof(DATA_TYPE) * DIMENSIONS * DIMENSIONS);
25 }
26
27 double timing(struct timeval start, struct timeval stop)
28 {
29   double dStart = 1.0e-6 * start.tv_usec;
30   double dStop = (stop.tv_sec - start.tv_sec) + 1.0e-6 * stop.tv_usec;
31
32   return dStop - dStart;
33 }
34
35 // Prints out the matrices if the parameter "SHOW_MATRICES" is set. (For
       debugging purposes.)
36 void printMatrix(const char name[], DATA_TYPE * mat)
37 {
38 #ifdef SHOW_MATRICES
39   printf("\nMatrix: %s\n", name);
40   for(int j = 0; j < DIMENSIONS; j++)
41   {
42     for(int i = 0; i < DIMENSIONS; i++)
43     {
44       printf("%d\t", (int)mat[j*DIMENSIONS + i]);
45     }
```

```
46      printf("\n");
47    }
48  #endif
49  }
50
51  // Generates the matrices.
52  void generateMatrices()
53  {
54    for(int j = 0; j < DIMENSIONS; j++)
55    {
56      for(int i = 0; i < DIMENSIONS; i++)
57      {
58        mat1[i + j * DIMENSIONS] = i + j;
59        if(abs(i - j) == 1)
60        {
61          mat2[j * DIMENSIONS + i] = 1;
62        }
63        else
64        {
65          mat2[j * DIMENSIONS + i] = 0;
66        }
67        mat3[j * DIMENSIONS + i] = 0;
68      }
69    }
70  }
71
72  // Runs a verification of the results using a sequential ijk version of the
        matrix multiplication.
73  #ifdef VERIFY
74  DATA_TYPE mat4[DIMENSIONS * DIMENSIONS];
75  inline void verifyData()
76  {
77    for(int i = 0; i < DIMENSIONS; i++)
78      for(int k = 0; k < DIMENSIONS; k++)
79        for(int j = 0; j < DIMENSIONS; j++)
80          mat4[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
              ];
81    printMatrix("C (verify)", mat4);
82    for(int i = 0; i < DIMENSIONS; i++)
83    {
84      for(int j = 0; j < DIMENSIONS; j++)
85      {
86        if(mat3[i*DIMENSIONS + j] != mat4[i*DIMENSIONS + j])
87        {
88          printf("Correct:\t\t0\n");
89          return;
90        }
91      }
92    }
93    printf("Correct:\t\t1\n");
94  }
95  #endif
96
97  // Ijk variant of the multiplication.
98  inline double runIjk(const bool verify)
99  {
100   struct timeval start, stop;
101
102   gettimeofday(&start, NULL);
103
```

```
104 #ifdef GRAINSIZE
105 #pragma cilk_grainsize=GRAINSIZE
106   cilk_for(int i = 0; i < DIMENSIONS; i++)
107     for(int j = 0; j < DIMENSIONS; j++)
108       for(int k = 0; k < DIMENSIONS; k++)
109         mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
              ];
110 #else
111   cilk_for(int i = 0; i < DIMENSIONS; i++)
112     for(int j = 0; j < DIMENSIONS; j++)
113       for(int k = 0; k < DIMENSIONS; k++)
114         mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
              ];
115 #endif
116   gettimeofday(&stop, NULL);
117   printMatrix("C (ijk)", mat3);
118 #ifdef VERIFY
119   if(verify)
120   {
121     verifyData();
122   }
123 #endif
124   clearC();
125
126   return timing(start, stop);
127 }
128
129 // Ikj variant of the multiplication.
130 inline double runIkj(const bool verify)
131 {
132   struct timeval start, stop;
133
134   gettimeofday(&start, NULL);
135
136 #ifdef GRAINSIZE
137 #pragma cilk_grainsize=GRAINSIZE
138   cilk_for(int i = 0; i < DIMENSIONS; i++)
139     for(int k = 0; k < DIMENSIONS; k++)
140       for(int j = 0; j < DIMENSIONS; j++)
141         mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
              ];
142 #else
143   cilk_for(int i = 0; i < DIMENSIONS; i++)
144     for(int k = 0; k < DIMENSIONS; k++)
145       for(int j = 0; j < DIMENSIONS; j++)
146         mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
              ];
147 #endif
148   gettimeofday(&stop, NULL);
149   printMatrix("C (ikj)", mat3);
150 #ifdef VERIFY
151   if(verify)
152   {
153     verifyData();
154   }
155 #endif
156   clearC();
157
158   return timing(start, stop);
159 }
```

```
160
161 // Submatrix variant of the matrix multiplication.
162 #ifdef SUBMATRIX_STRIDE
163 inline double runSubmatrix(const bool verify)
164 {
165   struct timeval start, stop;
166
167   gettimeofday(&start, NULL);
168
169   cilk_for(int it = 0; it < DIMENSIONS; it+=SUBMATRIX_STRIDE)
170     for(int kt = 0; kt < DIMENSIONS; kt+=SUBMATRIX_STRIDE)
171       for(int jt = 0; jt < DIMENSIONS; jt+=SUBMATRIX_STRIDE)
172         for(int i = it; i < (it + SUBMATRIX_STRIDE) && i < DIMENSIONS; i++)
173           for(int k = kt; k < (kt + SUBMATRIX_STRIDE) && k < DIMENSIONS; k++)
174             for(int j = jt; j < (jt + SUBMATRIX_STRIDE) && j < DIMENSIONS; j++)
175               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                    DIMENSIONS + j];
176   gettimeofday(&stop, NULL);
177   printMatrix("C (submatrix)", mat3);
178 #ifdef VERIFY
179   if(verify)
180   {
181     verifyData();
182   }
183 #endif
184   clearC();
185
186   return timing(start, stop);
187 }
188
189 // Unrolled submatrix variant of the matrix multiplication.
190 inline double runSubmatrixUnroll(const bool verify)
191 {
192   struct timeval start, stop;
193
194   gettimeofday(&start, NULL);
195
196   int dim = DIMENSIONS - (DIMENSIONS % SUBMATRIX_STRIDE);
197
198   // i, j, k
199   cilk_for(int it = 0; it < dim; it+=SUBMATRIX_STRIDE)
200     for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
201       for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
202         for(int i = it; i < it + SUBMATRIX_STRIDE; i++)
203           for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
204             for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
205               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                    DIMENSIONS + j];
206
207   if(DIMENSIONS != dim)
208   {
209     // -i, j, k
210     for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
211       cilk_for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
212         for(int i = dim; i < DIMENSIONS; i++)
213           for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
214             for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
215               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                    DIMENSIONS + j];
216
```

```
217     // -i, j, -k
218     cilk_for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
219       for(int i = dim; i < DIMENSIONS; i++)
220         for(int k = dim; k < DIMENSIONS; k++)
221           for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
222             mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS
                    + j];
223
224     // -i, -j, -k
225     cilk_for(int i = dim; i < DIMENSIONS; i++)
226       for(int k = dim; k < DIMENSIONS; k++)
227         for(int j = dim; j < DIMENSIONS; j++)
228           mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS +
                  j];
229
230     // i, j, -k
231     cilk_for(int it = 0; it < dim; it+=SUBMATRIX_STRIDE)
232       for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
233         for(int i = it; i < it + SUBMATRIX_STRIDE; i++)
234           for(int k = dim; k < DIMENSIONS; k++)
235             for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
236               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
237
238     // i, -j, -k
239     cilk_for(int it = 0; it < dim; it+=SUBMATRIX_STRIDE)
240       for(int i = it; i < it + SUBMATRIX_STRIDE; i++)
241         for(int k = dim; k < DIMENSIONS; k++)
242           for(int j = dim; j < DIMENSIONS; j++)
243             mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS
                    + j];
244
245     // i, -j, k
246     cilk_for(int it = 0; it < dim; it+=SUBMATRIX_STRIDE)
247       for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
248         for(int i = it; i < it + SUBMATRIX_STRIDE; i++)
249           for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
250             for(int j = dim; j < DIMENSIONS; j++)
251               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
252
253     // -i, -j, k
254     for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
255       cilk_for(int i = dim; i < DIMENSIONS; i++)
256         for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
257           for(int j = dim; j < DIMENSIONS; j++)
258             mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS
                    + j];
259   }
260   gettimeofday(&stop, NULL);
261   printMatrix("C (submatrix_unroll)", mat3);
262 #ifdef VERIFY
263   if(verify)
264   {
265     verifyData();
266   }
267 #endif
268   clearC();
269
270   return timing(start, stop);
```

```
271 }
272 #endif
273
274 // Drops the highest and lowest timing and calculates the average of the rest.
275 double getAvg(double *timings)
276 {
277   double avg = 0.0;
278   double min, max;
279
280   if(timings[0] > timings[1])
281   {
282     max = timings[0];
283     min = timings[1];
284   }
285   else
286   {
287     min = timings[0];
288     max = timings[1];
289   }
290
291   for(int i = 2; i < RUNS; i++)
292   {
293     if(timings[i] > max)
294     {
295       avg += max;
296       max = timings[i];
297     }
298     else if(timings[i] < min)
299     {
300       avg += min;
301       min = timings[i];
302     }
303     else
304     {
305       avg += timings[i];
306     }
307   }
308   return avg / (RUNS - 2);
309 }
310
311 int my_cilk_main(void *unused) {
312   generateMatrices();
313
314   printf("Method:\t\t\t%s\n", METHOD);
315   printf("Dimensions:\t\t%d\n", DIMENSIONS);
316   printf("Num. Threads:\t\t%d\n", NUM_THREADS);
317   printf("DataType:\t\t%s\n", DATA_TYPE_NAME);
318 #ifdef GRAINSIZE
319   printf("Grainsize:\t\t%d\n", GRAINSIZE);
320 #endif
321   printf("Runs\t\t\t%d\n", RUNS);
322 #ifdef SUBMATRIX_STRIDE
323   printf("Submatrix Stride:\t%d\n", SUBMATRIX_STRIDE);
324 #endif
325   printMatrix("A", mat1);
326   printMatrix("B", mat2);
327
328   double timings[RUNS];
329
330   for(int i = 0; i < RUNS; i++)
```

```
331   {
332     timings[i] = RUN(false);
333   }
334   printf("Avg. Time:\t\t%f\n", getAvg(timings));
335 #ifdef VERIFY
336   RUN(true);
337 #endif
338   return 0;
339 }
340
341 int main(void) {
342   cilk::context ctx;
343   ctx.set_worker_count(NUM_THREADS);
344   return ctx.run(my_cilk_main, NULL);
345 }
```

Listing 7: TBB implementation of the matrix multiplication code. (see Sections 4.1.4 and 5.5.2)

```
1  //============================================================================
2  // Name        : MatMul.cpp
3  // Description : Matrix multiplication parallelized using TBB. To be
4  //           consistent with literature, column major order is used.
5  //============================================================================
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <sys/time.h>
10 #include <string.h>
11 #include <algorithm>
12 #include "parameters.h"
13
14 #include "tbb/task_scheduler_init.h"
15 #include "tbb/parallel_for.h"
16 #include "tbb/blocked_range.h"
17
18 using namespace std;
19 using namespace tbb;
20
21 #define VERIFY
22
23 DATA_TYPE mat1[DIMENSIONS * DIMENSIONS];
24 DATA_TYPE mat2[DIMENSIONS * DIMENSIONS];
25 DATA_TYPE mat3[DIMENSIONS * DIMENSIONS];
26
27 inline void clearC()
28 {
29   memset(mat3, 0, sizeof(DATA_TYPE) * DIMENSIONS * DIMENSIONS);
30 }
31
32 double timing(struct timeval start, struct timeval stop)
33 {
34   double dStart = 1.0e-6 * start.tv_usec;
35   double dStop = (stop.tv_sec - start.tv_sec) + 1.0e-6 * stop.tv_usec;
36
37   return dStop - dStart;
38 }
39
40 // Prints out the matrices if the parameter "SHOW_MATRICES" is set. (For
       debugging purposes.)
41 void printMatrix(const char name[], DATA_TYPE * mat)
```

```
42 {
43 #ifdef SHOW_MATRICES
44   printf("\nMatrix: %s\n", name);
45   for(int j = 0; j < DIMENSIONS; j++)
46   {
47     for(int i = 0; i < DIMENSIONS; i++)
48     {
49       printf("%d\t", (int)mat[j*DIMENSIONS + i]);
50     }
51     printf("\n");
52   }
53 #endif
54 }
55
56 // Generates the matrices.
57 void generateMatrices()
58 {
59   for(int j = 0; j < DIMENSIONS; j++)
60   {
61     for(int i = 0; i < DIMENSIONS; i++)
62     {
63       mat1[i + j * DIMENSIONS] = i + j;
64       if(abs(i - j) == 1)
65       {
66         mat2[j * DIMENSIONS + i] = 1;
67       }
68       else
69       {
70         mat2[j * DIMENSIONS + i] = 0;
71       }
72       mat3[j * DIMENSIONS + i] = 0;
73     }
74   }
75 }
76
77 // Runs a verification of the results using a sequential ijk version of the
       matrix multiplication.
78 #ifdef VERIFY
79 DATA_TYPE mat4[DIMENSIONS * DIMENSIONS];
80 inline void verifyData()
81 {
82   // Uses a sequential ijk version for verification.
83   for(int i = 0; i < DIMENSIONS; i++)
84     for(int k = 0; k < DIMENSIONS; k++)
85       for(int j = 0; j < DIMENSIONS; j++)
86         mat4[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS + j
              ];
87   printMatrix("C (verify)", mat4);
88   for(int i = 0; i < DIMENSIONS; i++)
89   {
90     for(int j = 0; j < DIMENSIONS; j++)
91     {
92       if(mat3[i*DIMENSIONS + j] != mat4[i*DIMENSIONS + j])
93       {
94         printf("Correct:\t\t0\n");
95         return;
96       }
97     }
98   }
99   printf("Correct:\t\t1\n");
```

```
100 }
101 #endif
102
103 struct Ijk {
104   void operator() (const blocked_range<int>& range) const
105   {
106     for(int i = range.begin(); i < range.end(); i++)
107       for(int j = 0; j < DIMENSIONS; j++)
108         for(int k = 0; k < DIMENSIONS; k++)
109           mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS +
                  j];
110   }
111 };
112
113 // Ijk variant of the multiplication.
114 inline double runIjk(const bool verify)
115 {
116   struct timeval start, stop;
117
118   gettimeofday(&start, NULL);
119   Ijk ijk;
120   parallel_for(blocked_range<int>(0, DIMENSIONS, GRAINSIZE), ijk);
121   gettimeofday(&stop, NULL);
122   printMatrix("C (ijk)", mat3);
123 #ifdef VERIFY
124   if(verify)
125   {
126     verifyData();
127   }
128 #endif
129   clearC();
130
131   return timing(start, stop);
132 }
133
134 struct Ikj {
135   void operator() (const blocked_range<int>& range) const
136   {
137     for(int i = range.begin(); i < range.end(); i++)
138       for(int k = 0; k < DIMENSIONS; k++)
139         for(int j = 0; j < DIMENSIONS; j++)
140           mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*DIMENSIONS +
                  j];
141   }
142 };
143
144 // Ikj variant of the multiplication.
145 inline double runIkj(const bool verify)
146 {
147   struct timeval start, stop;
148
149   gettimeofday(&start, NULL);
150   Ikj ikj;
151   parallel_for(blocked_range<int>(0, DIMENSIONS, GRAINSIZE), ikj);
152   gettimeofday(&stop, NULL);
153   printMatrix("C (ikj)", mat3);
154 #ifdef VERIFY
155   if(verify)
156   {
157     verifyData();
```

```
158   }
159 #endif
160   clearC();
161
162   return timing(start, stop);
163 }
164
165 #ifdef SUBMATRIX_STRIDE
166
167 struct Submatrix {
168   void operator() (const blocked_range<int>& range) const
169   {
170     for(int it = range.begin(); it < range.end(); it+=SUBMATRIX_STRIDE)
171       for(int kt = 0; kt < DIMENSIONS; kt+=SUBMATRIX_STRIDE)
172         for(int jt = 0; jt < DIMENSIONS; jt+=SUBMATRIX_STRIDE)
173           for(int i = it; i < (it + SUBMATRIX_STRIDE) && i < range.end(); i++)
174             for(int k = kt; k < (kt + SUBMATRIX_STRIDE) && k < DIMENSIONS; k++)
175               for(int j = jt; j < (jt + SUBMATRIX_STRIDE) && j < DIMENSIONS; j
                      ++)
176                 mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
177
178   }
179 };
180
181 // Submatrix variant of the matrix multiplication.
182 inline double runSubmatrix(const bool verify)
183 {
184   struct timeval start, stop;
185
186   gettimeofday(&start, NULL);
187
188   Submatrix submatrix;
189   parallel_for(blocked_range<int>(0, DIMENSIONS, GRAINSIZE), submatrix);
190
191   gettimeofday(&stop, NULL);
192   printMatrix("C (submatrix)", mat3);
193 #ifdef VERIFY
194   if(verify)
195   {
196     verifyData();
197   }
198 #endif
199   clearC();
200
201   return timing(start, stop);
202 }
203
204 struct SubmatrixUnroll {
205   void operator() (const blocked_range<int>& range) const
206   {
207     int dim = DIMENSIONS - (DIMENSIONS % SUBMATRIX_STRIDE);
208     int dimi = range.end() - ((range.end() - range.begin()) % SUBMATRIX_STRIDE);
209     int stepi = SUBMATRIX_STRIDE;
210     bool unrolli = true;
211     if(stepi > (range.end() - range.begin()))
212     {
213       stepi = range.end() - range.begin();
214       dimi = range.end();
215     }
```

```
216     if(dimi == range.end())
217     {
218       unrolli = false;
219     }
220
221     for(int it = range.begin(); it < dimi; it+=stepi)
222       for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
223         for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
224           for(int i = it; i < (it + stepi); i++)
225             for(int k = kt; k < (kt + SUBMATRIX_STRIDE); k++)
226               for(int j = jt; j < (jt + SUBMATRIX_STRIDE); j++)
227                 mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
228
229     if(dim != DIMENSIONS)
230     {
231       if(unrolli)
232       {
233         // -i, j, k
234         for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
235           for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
236             for(int i = dimi; i < range.end(); i++)
237               for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
238                 for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
239                   mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                        DIMENSIONS + j];
240
241         // -i, j, -k
242         for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
243           for(int i = dimi; i < range.end(); i++)
244             for(int k = dim; k < DIMENSIONS; k++)
245               for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
246                 mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
247
248         // -i, -j, -k
249         for(int i = dimi; i < range.end(); i++)
250           for(int k = dim; k < DIMENSIONS; k++)
251             for(int j = dim; j < DIMENSIONS; j++)
252               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                    DIMENSIONS + j];
253       }
254       // i, j, -k
255       for(int it = range.begin(); it < dimi; it+=stepi)
256         for(int jt = 0; jt < dim; jt+=SUBMATRIX_STRIDE)
257           for(int i = it; i < it + stepi; i++)
258             for(int k = dim; k < DIMENSIONS; k++)
259               for(int j = jt; j < jt + SUBMATRIX_STRIDE; j++)
260                 mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
261
262       // i, -j, -k
263       for(int it = range.begin(); it < dimi; it+=stepi)
264         for(int i = it; i < it + stepi; i++)
265           for(int k = dim; k < DIMENSIONS; k++)
266             for(int j = dim; j < DIMENSIONS; j++)
267               mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                    DIMENSIONS + j];
268
269       // i, -j, k
```

```
270       for(int it = range.begin(); it < dimi; it+=stepi)
271         for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
272           for(int i = it; i < it + stepi; i++)
273             for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
274               for(int j = dim; j < DIMENSIONS; j++)
275                 mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
276
277       if(unrolli)
278       {
279         // -i, -j, k
280         for(int kt = 0; kt < dim; kt+=SUBMATRIX_STRIDE)
281           for(int i = dimi; i < range.end(); i++)
282             for(int k = kt; k < kt + SUBMATRIX_STRIDE; k++)
283               for(int j = dim; j < DIMENSIONS; j++)
284                 mat3[i*DIMENSIONS + j] += mat1[i*DIMENSIONS + k] * mat2[k*
                      DIMENSIONS + j];
285       }
286     }
287   }
288 };
289
290 // Unrolled submatrix variant of the matrix multiplication.
291 inline double runSubmatrixUnroll(const bool verify)
292 {
293   struct timeval start, stop;
294
295   gettimeofday(&start, NULL);
296
297   SubmatrixUnroll submatrixUnroll;
298   parallel_for(blocked_range<int>(0, DIMENSIONS, GRAINSIZE), submatrixUnroll);
299
300   gettimeofday(&stop, NULL);
301   printMatrix("C (submatrix_unroll)", mat3);
302 #ifdef VERIFY
303   if(verify)
304   {
305     verifyData();
306   }
307 #endif
308   clearC();
309
310   return timing(start, stop);
311 }
312 #endif
313
314 // Drops the highest and lowest timing and calculates the average of the rest.
315 double getAvg(double *timings)
316 {
317   double avg = 0.0;
318   double min, max;
319
320   if(timings[0] > timings[1])
321   {
322     max = timings[0];
323     min = timings[1];
324   }
325   else
326   {
327     min = timings[0];
```

```
328     max = timings[1];
329   }
330
331   for(int i = 2; i < RUNS; i++)
332   {
333     if(timings[i] > max)
334     {
335       avg += max;
336       max = timings[i];
337     }
338     else if(timings[i] < min)
339     {
340       avg += min;
341       min = timings[i];
342     }
343     else
344     {
345       avg += timings[i];
346     }
347   }
348   return avg / (RUNS - 2);
349 }
350
351 int main(void) {
352   generateMatrices();
353
354   printf("Method:\t\t\t%s\n", METHOD);
355   printf("Dimensions:\t\t%d\n", DIMENSIONS);
356   printf("Num. Threads:\t\t%d\n", NUM_THREADS);
357   printf("DataType:\t\t%s\n", DATA_TYPE_NAME);
358   printf("Grainsize:\t\t%d\n", GRAINSIZE);
359   printf("Runs\t\t\t%d\n", RUNS);
360 #ifdef SUBMATRIX_STRIDE
361   printf("Submatrix Stride:\t%d\n", SUBMATRIX_STRIDE);
362 #endif
363   printMatrix("A", mat1);
364   printMatrix("B", mat2);
365
366   // Initialize the task scheduler.
367   task_scheduler_init init(NUM_THREADS);
368
369   double timings[RUNS];
370
371   for(int i = 0; i < RUNS; i++)
372   {
373     timings[i] = RUN(false);
374   }
375   printf("Avg. Time:\t\t%f\n", getAvg(timings));
376 #ifdef VERIFY
377   RUN(true);
378 #endif
379 }
```

Listing 8: Chapel implementation of the matrix multiplication code. (see Sections 4.2.5 and 5.5.4)

```
1 use Time;
2
3 config var n = 16;
4 config var runs = 7;
```

```
5
6  def main() {
7    const ProblemSpace: domain(2) distributed(Block) = [1..n, 1..n];
8
9    var A, B, C: [ProblemSpace] real(64) = 0.0;
10
11   var timings: [1..runs] real;
12   var myTime: Timer();
13
14   // Generate matrices.
15   forall (i, j) in ProblemSpace do
16   {
17     A(i, j) = i + j - 2;
18     if(abs(i - j) == 1)
19     {
20       B(i, j) = 1;
21     }
22   }
23
24   for run in 1..runs
25   {
26     myTime.start();
27
28     // Matrix multiplication Kernel
29     forall (i, j) in ProblemSpace do
30     {
31       C(i, j) = C(i, j) + + reduce (A(i, 1..n) * B(1..n, j));
32     }
33     myTime.stop();
34     timings[run] = myTime.elapsed();
35     myTime.clear();
36   }
37
38   var avgTime = ((+ reduce timings) - (max reduce timings) - (min reduce timings
         )) / (runs - 2);
39
40   writeln("Dimensions:\t\t", n);
41   writeln("Num. Threads:\t\t", maxThreads);
42   writeln("DataType:\t\tdouble");
43   writeln("Runs:\t\t\t", runs);
44   writeln("Avg. Time:\t\t", avgTime);
45 }
```

Lebenslauf

Name:	Martin Wimmer
Geburtsdatum:	03.08.1982
Geburtsort:	Wien

Ausbildung

1988-1992	Volksschule Alt-Erlaa
1992-2000	BG Mödling, Untere Bachgasse 8
2000-2005	Bakkalaureatsstudium Wirtschaftsinformatik, Universität Wien
2006	Zivildienst
seit 2007	Masterstudium Scientific Computing, Universität Wien