A User-Centric Perspective on Parallel Programming with Focus on OpenMP

DISSERTATION

zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
im Fachbereich 16 (Elektrotechnik/Informatik)
der Universität Kassel

vorgelegt von

Michael Suess

Kassel im September 2007

To Claudia

Summary

The process of developing software that takes advantage of multiple processors is commonly referred to as parallel programming. For various reasons, this process is much harder than the sequential case. For decades, parallel programming has been a problem for a small niche only: engineers working on parallelizing mostly numerical applications in High Performance Computing. This has changed with the advent of multi-core processors in mainstream computer architectures. Parallel programming in our days becomes a problem for a much larger group of developers.

The main objective of this thesis was to find ways to make parallel programming easier for them. Different aims were identified in order to reach the objective: research the state of the art of parallel programming today, improve the education of software developers about the topic, and provide programmers with powerful abstractions to make their work easier.

To reach these aims, several key steps were taken. To start with, a survey was conducted among parallel programmers to find out about the state of the art. More than 250 people participated, yielding results about the parallel programming systems and languages in use, as well as about common problems with these systems. Furthermore, a study was conducted in university classes on parallel programming. It resulted in a list of frequently made mistakes that were analyzed and used to create a programmers' checklist to avoid them in the future.

For programmers' education, an online resource was setup to collect experiences and knowledge in the field of parallel programming – called the Parawiki. Another key step in this direction was the creation of the Thinking Parallel weblog, where more than 50.000 readers to date have read essays on the topic.

For the third aim (powerful abstractions), it was decided to concentrate on one parallel programming system: OpenMP. Its ease of use and high level of abstraction were the most important reasons for this decision. Two different research directions were pursued. The first one resulted in a parallel library called AthenaMP. It contains so-called generic components, derived from design patterns for parallel programming. These include functionality to enhance the locks provided by OpenMP, to perform operations on large amounts of data (data-parallel programming), and to enable the implementation of irregular algorithms using task pools. AthenaMP itself serves a triple role: the components are well-documented and can be used directly in programs, it enables developers to study the source code and learn from it, and it is possible for compiler writers to use it as a testing ground for their OpenMP compilers.

The second research direction was targeted at changing the OpenMP specification to make the system more powerful. The main contributions here were a proposal to enable thread-cancellation and a proposal to avoid busy waiting. Both were implemented in a research compiler, shown to be useful in example applications, and proposed to the OpenMP Language Committee.

Zusammenfassung

Der Software-Entwicklungsprozess für Programme, die mehrere Prozessoren ausnutzen wird üblicherweise als parallele Programmierung bezeichnet. Aus mehreren Gründen ist parallele deutlich schwieriger als sequentielle Programmierung. Während der letzten Jahrzehnte war von diesem Problem nur eine schmale Randgruppe betroffen: Ingenieure, die an der Parallelisierung von hauptsächlich numerischen Anwendungen im Hochleistungsrechnen arbeiteten. Seit Multi-Core Computerarchitekturen weite Verbreitung gefunden haben, hat sich das geändert. Heutzutage ist die parallele Programmierung ein Problem für eine bedeutend größere Gruppe von Entwicklern.

Ziel dieser Arbeit war das Erkunden von Wegen, um die parallele Programmierung für diese Gruppe einfacher zu gestalten. Dazu wurden verschiedene Teilziele verfolgt: den Stand der Technik zu erfassen, die Kenntnisse der Entwickler über dieses Thema zu verbessern und den Programmierern mächtige Abstraktionen zur Erleichterung ihrer Arbeit an die Hand zu geben.

Zunächst wurde eine Umfrage unter parallelen Programmierern durchgeführt. Aus den Antworten von über 250 Teilnehmern konnten Erkenntnisse über die verwendeten Systeme, Sprachen und deren Probleme gewonnen werden. Weiterhin wurde auf Basis von Lehrveranstaltungen zum Thema Parallelverarbeitung an der Universität Kassel eine Studie zu häufigen Fehlern durchgeführt. Daraus konnte später eine Checkliste für Programmierer erstellt werden, damit die Fehler in Zukunft vermieden werden können.

Um die Kenntnisse der Programmierer auszubauen und Erfahrungen und Wissen auf dem Gebiet der parallelen Programmierung an einer Stelle zu vereinen, wurde im Internet eine Ressource erstellt – das Parawiki. Ein weiterer wichtiger Schritt in diese Richtung war der Aufbau des Thinking Parallel Weblogs, in dem bereits mehr als 50.000 Interessierte Aufsätze zum Thema gelesen haben.

Bezüglich des dritten Ziels (mächtige Abstraktionen) konzentriert sich die Arbeit auf ein einziges paralleles Programmiersystem: OpenMP. Die Hauptgründe für diese Entscheidung waren seine leichte Bedienbarkeit und seine hohe Abstraktionsebene. Auch hier wurden zwei unterschiedliche Forschungsansätze verfolgt. Aus dem Ersten ging die parallele Bibliothek AthenaMP hervor. Sie enthält so genannte generische Komponenten, welche von Entwurfsmustern der parallelen Programmierung abgeleitet wurden und stellt Funktionalität zur Verfügung, um z.B. die von OpenMP bereit gestellten Locks zu erweitern, Operationen auf großen Datenmengen auszuführen (datenparallele Programmierung) und die Implementierung irregulärer Algorithmen mit Hilfe von Taskpools zu erleichtern. AthenaMP ermöglicht den direkten Einsatz der Komponenten in Programmen und unterstützt dies durch gute Dokumentation. Darüber hinaus können Entwickler die Quellen einsehen und daraus lernen und Compilerhersteller sind in der Lage, die Bibliothek als Testfall für ihre OpenMP-Compiler einzusetzen.

Der zweite Ansatz war die Erweiterung der OpenMP-Spezifikation. Die wichtigsten Beiträge sind Erweiterungen zum Abbruch von Threads und zum Vermeiden aktiven Wartens. Beide wurden in einem Compiler implementiert, ihr Nutzen wurde anhand von Anwendungen belegt und sie wurden dem OpenMP-Sprachausschuss vorgeschlagen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The density of transistors on chips doubles every 24 months.

(Gordon Moore, 1965)

1.1 Motivation

The world of computer hardware and architectures is going through a revolution today. For the past decades, Moore's Law has been translated into faster processors by the chip-industry. As a side-effect, the performance of sequential programs has increased without additional effort on the side of the programmers, as well. When a program did not show a satisfactory performance, it was often sufficient to wait for six months and the next generation of hardware to make its performance acceptable.

The situation is different today. Heat-problems have forced the chip-makers to abandon the MHz-race. Yet Moore's Law still holds, providing the industry with an increase of approximately 100 percent in available transistors per chip every 24 months. Since they cannot use these transistors to increase execution speeds (because of excess heat), they have decided to use them differently. Multiple processors have been put on a single chip side-by-side (called *multi-core processors*) and sometimes each of these processors is even able to run more than one thread simultaneously – commonly referred to as *Chip-Multithreading* (CMT). An example of this is the `Sun T1 processor`, which has eight cores, each of which is able to run four threads at the same time. While this approach has great potential for performance increases, it also has a big problem: existing programs will not run any faster on these architectures, except if they are parallelized.

Parallelizing Programs (often referred to as *parallel programming*) is still a difficult task. Great expectations have been raised for automatic parallelization in the past, yet today these efforts are not up to par with what can be achieved by the experienced programmer. In addition to all the problems encountered by the developers of sequential programs, a parallel programmer has to face concurrency and synchronization issues, data distribution and task mapping difficulties, as well as timing and debugging problems, to name just a few (a more thorough discussion can be found in Section 2.2). Dealing with them requires great care and skill.

In the past, the main emphasis when developing languages and other systems for parallel programming has been performance. Performance is defined in this context by terms like *execution speed* or *throughput*. This seems natural, because one of the main reasons to develop parallel programs is speeding them up. From the world of sequential programming and software engineering, different topics are slowly making their way into the parallel processing community. Topics like *readability of code*, *ease of use*, or *decreasing the possibility of programming mistakes* are examples. In *High Performance Computing* (HPC), these have not been as important, but during the parallel revolution described above, where many more software developers have to cope with parallel programming, they will start to play increasing roles.

In 1997, a new specification for programming shared memory multiprocessors called *OpenMP* [Ope05] has been created with some of these topics in mind. Shared memory systems avoid many of the difficulties associated with programming distributed memory architectures, but have a whole class of their own problems. OpenMP tries to make it as easy as possible for programmers to deal with them. What OpenMP does not do at the moment, is to provide the parallel programmer with all the flexibility and power of other, more traditional (thread-based) parallel programming systems.

Because of the reasons stated in the last few paragraphs, the main objective of this thesis is to *make parallel programming easier* for the *normal* programmer. Different aims were identified in order to reach that objective.

The first one was to understand the situation and difficulties associated with parallel programming better by analyzing the current *state of affairs* regarding the field in general. It became necessary to understand which parallel programming systems are in actual use by programmers today, what languages they are based on and what their problems are.

The second aim I pursued in this thesis was to research how to *improve the knowledge and education* of software developers on the topic of parallel programming. The logic behind that aim was that experienced developers with good background knowledge on parallel programming will find creating parallelized programs easier.

The third and most important aim identified was to provide software developers with *powerful abstractions* to make their work easier. I concentrated on one parallel programming system for this part: OpenMP. These abstractions were both a library for parallel programming and changes in the OpenMP specification to make it more powerful and expressive. By encapsulating at least some of the difficulties associated with parallel programming in either a library or the language itself, work becomes easier for the developer using these abstractions.

All aims and objectives of this work are depicted in Figure 1.1, with the main objective shown in red color and the aims painted in yellow. The concrete steps taken to pursue these aims are shown in green and blue and are shortly introduced in Section 1.2. I will be using this picture at the beginning of each advanced chapter throughout this thesis to show how the topic of the respective chapter fits into the context of this work.

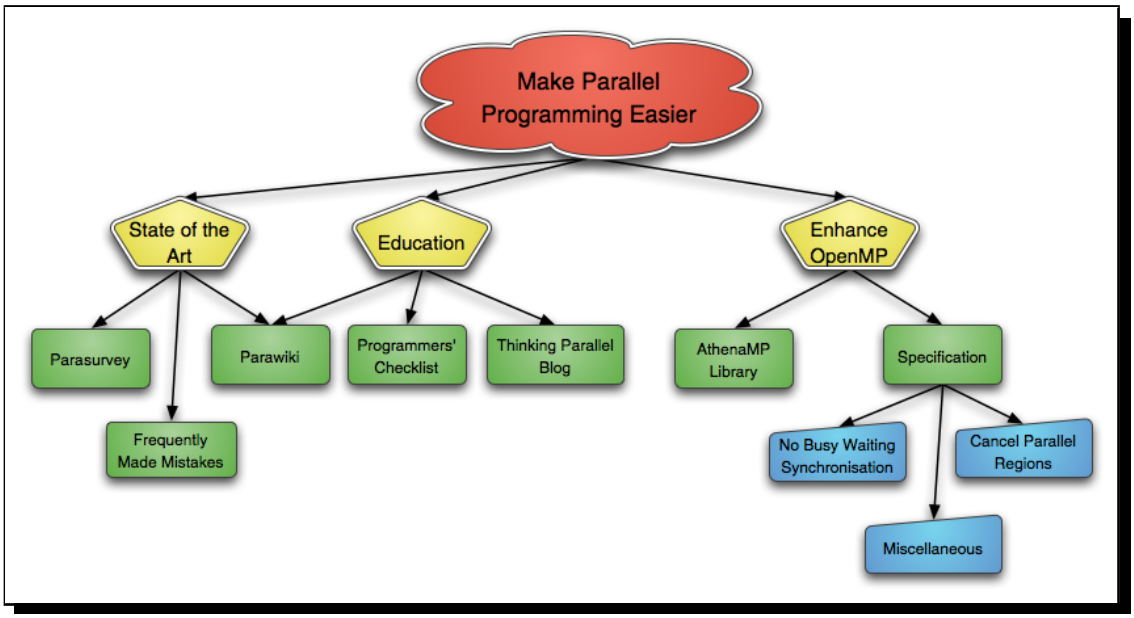


Figure 1.1: Objectives, aims and contributions of this thesis

1.2 Contributions

This section highlights the contributions of this thesis. They are shown in green and blue in Figure 1.1. The first aim I pursued was to get a feeling for the current *state of the art* with regards to parallel programming. My main contribution in this field is a survey I have conducted among parallel programmers, called *Parasurvey* in the figure. More than 250 people participated and enabled me to form hypotheses on the actual popularity and usage of languages and systems in the field, along with information about target architectures and encountered problems, among others.

The second contribution is a *list of frequently made mistakes* by novice parallel programmers when using OpenMP. This list was created by observing my students in multiple projects. It is useful, because knowing about the mistakes helps to avoid them, leading to better programs. Furthermore, some of the mistakes are made less likely to occur by solutions presented later in this thesis, which justifies using those solutions even more.

To find information about different parallel programming systems, especially their strengths, weaknesses or applicability is a difficult task. The information from the creators of these systems is of course biased and reports on experiences are difficult to find, if available at all. To change this situation, I have setup a single place to find these kinds of information on parallel programming systems – the *Parawiki*. It was supposed to provide me with information about parallel programming by actual users of these systems and to help educate programmers, who can learn from the experiences described there as well.

The Parawiki has already brought us into the territory of my second aim: to find ways to *improve the education* of programmers about parallel programming. The main con-

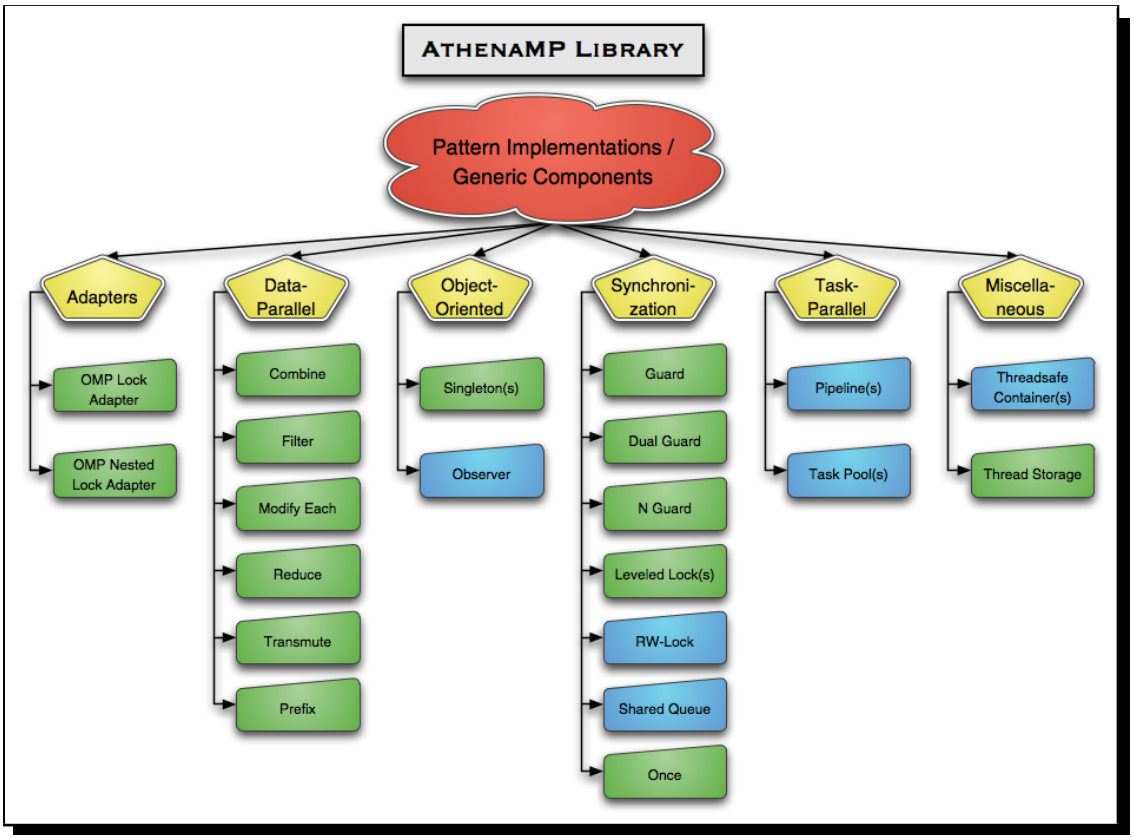


Figure 1.2: The AthenaMP library in a nutshell

tribution for this aim is a *programmers checklist* for OpenMP. It has been derived from experiences with my students and their mistakes and from my own programming experiences.

The last contribution to the aim of education is not exactly classifyable as research, yet still valuable. I am talking about the *Thinking Parallel* Weblog. Read by more than 50.000 readers to date and with more than 1.200 regular subscribers, this has probably had a bigger impact than all of my other efforts combined.

My last and most important aim was to make OpenMP easier to use and more powerful. I have taken two main steps here: providing a powerful library for OpenMP/C++ and preparing to enhance the specification. The library I am talking about is called *AthenaMP* and contains implementations of various parallel programming patterns in the form of generic components. They allow to specify advanced programming constructs with a few simple library calls, even to the point that parallelism is totally hidden from the programmer for a few cases. A quick overview over the functionality included in AthenaMP is presented in Figure 1.2.

AthenaMP is a big project with its own objectives. The generic components included in the library are well-documented and can be used as is in parallel programs. The library

is also useful as a learning aid for programmers interested in OpenMP/C++, since it is published under an open-source license. Compiler writers are also able to take advantage of it as a testing ground for their compilers, as not much code using OpenMP/C++ is available to test those. The library contains more than one hundred tests at this point that expose quite a few errors in compilers.

During my experiments with OpenMP, it became clear that some problems could only be solved by changing the specification. These problems include the inability to cancel threads in a parallel region and to avoid busy waiting, among others. To solve them, I have researched, implemented, tested and proposed to the OpenMP language committee changes to the OpenMP specification. All implementations were done in a research compiler called OMPi [DGLT03]. For a few other problems, I have worked with others on solutions while serving in the OpenMP language committee. My contributions in this regard are also described in this thesis.

Another type of contribution frequently present throughout this thesis are descriptions of workarounds. Before extending the specification or providing library functions/components, I have always tried to work around the problem first. These approaches are also described here.

The work done on improving OpenMP by providing powerful abstractions is mostly based on sample programs. In my experience, working with the system is the best way to find the areas where it can be improved. I have therefore implemented a variety of small example applications (e. g. `quicksort`, `labyrinth-search`, and `parallel prefix`). While they can be considered interesting to some readers, I have chosen not to include their sources, because for the context of my work they play a minor role. Of course, they are available on request.

Lastly, the whole thesis can also be seen as a contribution in a different regard: it serves as an evaluation of OpenMP, showing problems, workarounds, solutions, common mistakes and program examples in a comprehensive way. It should give a good estimation of what can be done with OpenMP and where its limits are.

1.3 Structure

The structure of this thesis closely follows the aims and contributions sketched in the last section. After this introduction, Chapter 2 lays the foundations necessary to understand the rest of this thesis. Of course that can only be an overview, as the field of parallel programming is too big to treat comprehensively in a few pages. The first aim, evaluating the state of the art of parallel programming, is described along with my contributions for this topic in Chapter 3. They include the parallel programming survey, the list of frequently made mistakes and the Parawiki. Chapter 4 describes my contributions to improve the education of programmers with regards to parallel programming. In this chapter, the programmers' checklist and the Thinking Parallel Weblog are described.

Until this point, the work has an emphasis on OpenMP, but can still be applied easily to other parallel programming systems. This changes with Chapters 5 and 6, as they are specific to OpenMP. In Chapter 5, a collection of parallel patterns and the generic components associated with them are described. These include task pools (useful for implementing irregular algorithms), generic locks, deadlock-avoidance functionality in locks, thread-safe singletons, and several data-parallel patterns. Most of those are available in the AthenaMP library. My work to change the OpenMP specification to make it more powerful and easier to use is presented in Chapter 6. The main topics of this chapter are proposals to enable thread-cancellation and to avoid busy waiting. Chapter 7 closes the thesis with a summary and possible directions for future work.

Most of the contributions presented in this thesis are in themselves structured according to what could be expected in a research paper. This is no surprise, as many of them have already been published in papers (see the list of publications at the end of this thesis for details). Therefore you will often find subsections for motivation, performance/benchmarks, and related work, allowing you to skip around in this thesis relatively freely if you are interested in specific parts only.

1.4 Acknowledgments

Above all, I am deeply in debt to my adviser Prof. Dr. Claudia Leopold. She has managed to create the best working environment imagineable for me and my research. Always available when I had problems or when I needed guidance, she also gave me the freedom to make mistakes and find my own way. I am glad I can not only call her an adviser, but a true friend. Thank you, Claudia!

I feel fortunate that Prof. Dr. Barbara Chapman has agreed to become my second adviser. Her enlightening talks in conferences, as well as our personal conversations have allowed me to enhance my knowledge of the field considerably. Special thanks to her for enabling me to join the OpenMP language committee through Compunity (an interest group on OpenMP that Barbara was one of the original founders), an experience that has enabled me to work with and learn from the creators of OpenMP. Thanks also goes to the Professors Wegner and Geihs for agreeing to join my thesis committee.

I thank my wife Claudia for her love and support. I consider myself lucky I have found a wife that endured the countless hours I have spent in front of the computer at home – and for sometimes forcing me to switch it off and do something different. Always willing to listen to my problems, her impact on the success of this work is greatly appreciated. Without you, none of this would have been possible!

Once again, I am deeply in dept to my family, especially but not limited to my parents, for supporting me with advice, guidance and open ears during this amazing time of my live.

I also thank my colleague Björn for guiding me through the darkest corners of C++ when I was getting lost. Moreover, thank you for letting me use your tremendous library and showing me how to use it without waisting too much time skimming through books.

There are numerous other people who deserve a big thank you, starting from my colleagues at work (Raffaele and yet another Claudia, to name just a few) to my students who have done an amazing amount of work to support my research. Thanks Beliz, Tobias, Alex, Christopher, Alex, Florian and the many others who have helped me during my time at the university. I hope I was a good adviser to you as well!

Last but not least I thank the University of Kassel for giving me the chance to do my dissertation here. I also thank the University Computing Centers at RWTH Aachen, TU Darmstadt, University of Frankfurt/Main and University of Kassel for providing the computing facilities used to test my sample applications on different compilers and hardware.

1.5 General Remarks

Although this is my dissertation, the work presented here was not done by me exclusively, as can be seen in the last paragraph. For this reason, I have chosen to stick to the more common *we* and *us* also found in many research papers for the rest of this publication to honor the people who have worked with me on the contributions presented here.This does not apply when talking about the Thinking Parallel Weblog in Section 4.2 and my work in the OpenMP language committee, which is why I am using the singular form then.

For readability, different font styles have been used to denote names, terms and identifiers, as shown in Table 1.1.

Style	Usage
`type writer`	products, companies, computers, architectures, programs, files, (member) functions, variables, data structures, data types, classes, clauses, constants, commands, environment variables
emphasize	special accentuation, terms

Table 1.1: Typographical conventions

1.6 Disclaimer

Trademarks and brand names have been used without explicitly indicating them. The absence of trademark symbols does not infer that a name or a product is not protected. All trademarks are the property of their respective owners.

Chapter 2

Foundations

In this chapter, the basics required to understand the rest of this thesis are explained. Since parallel programming itself is a huge topic, merely an overview can be provided here, along with references for further reading.

The chapter starts with a general introduction on how to parallelize a program in Section 2.1. A birds-view on the basic steps that need to be carried out for most successful parallelization efforts are sketched there. Afterwards, some of the problems and pitfalls commonly encountered while parallelizing code are highlighted in Section 2.2. This is necessary to understand how we have approached our main goal of making parallel programming easier, and it will also explain why we have chosen OpenMP as the basis of our research.

In the following Section 2.3, a short overview over the most commonly used types of parallel programming systems is provided. Once again, only a very broad generalization can be given, but it should be enough to gain an understanding of the systems in use today. As already explained in the introduction, a large part of this thesis attempts to enhance the parallel programming system OpenMP. Therefore a deeper understanding of this system is necessary to understand the contents of Chapters 5 and 6. To make up for this, a short tutorial on OpenMP is provided in Section 2.4. A summary closes the chapter in Section2.5.

2.1 An Introduction to Program Parallelization

This section presents the general steps involved in parallelizing a program. While for many programmers this is still a purely intuitive process, guided mostly by their own experiences, there are recipes of basic steps that can be followed to parallelize an algorithm/program.

Probably the most well-known methodology is presented by Grama et al. [Gra03]. It describes the process of building a parallel algorithm as a series of five steps:

- *Identifying portions of work that can be performed concurrently*
- *Mapping the concurrent pieces of work onto multiple execution entities running in parallel*

- *Distributing the input, output, and intermediate data associated with the program*
- *Managing accesses to data shared by multiple processors*
- *Synchronizing the processors at various stages of the parallel program execution*

([Gra03, p. 85])

We are going to use this methodology as the basis for this section. Its biggest advantage is that it defers architecture-specific decisions until the very last steps. A similar methodology is described by Quinn [Qui03] and Foster [Fos95]. It has four basic steps: *partitioning*, *communication*, *agglomeration* and *mapping*. Both methodologies are similar enough to not include the latter here.

2.1.1 Decomposition

The first step in Grama's methodology is called *decomposition* (also: *task decomposition*). The goal for this step is to divide the problem into several smaller subproblems, called *tasks*, that can be computed in parallel later on. The tasks can be of different size and must not necessarily be independent.

If many tasks are created, we talk about *fine-grained decomposition*. If few tasks are created, it is called *coarse-grained decomposition*. There is no sharp line in-between the two, and what is better heavily depends on the problem. Many tasks allow for more concurrency and better scalability, while fewer tasks usually have less communication / synchronization-overhead.

There are several ways to do problem decompositions, the most well-known probably being *recursive decomposition*, *data decomposition*, *functional decomposition*, *exploratory decomposition* and *speculative decomposition*. The next few paragraphs will shortly explain how they are carried out in practice.

Recursive Decomposition: *Divide-and-Conquer* is a widely deployed strategy for sequential algorithms. A problem is divided into subproblems here, which are again divided into subproblems recursively until a trivial solution can be calculated. Afterwards, the results of the subproblems are merged together as needed. This strategy is equivalent to a recursive problem decomposition. As the smaller tasks are often independent of one another, they can be calculated in parallel, often leading to well-scaling parallel algorithms. The parallel quicksort described in Section 5.4 is an example of recursive decomposition.

Data Decomposition: When data structures with large amounts of similar data need to be processed, data decomposition is usually a well-performing strategy. The tasks in this strategy correspond to groups of data. These can be either input data, output data or even intermediate data. All processors perform the same operations on these data, which

are usually independent from each other, once again allowing for well-scaling algorithms. The data-parallel patterns described in Section 5.3 all employ data decomposition.

Functional Decomposition: For functional decomposition, the functions to be performed are split into multiple tasks. These tasks can then be performed concurrently by different execution entities on different data. This often leads to so-called *pipelines*, which are described in more detail in Section 5.5.5.

Exploratory Decomposition: Exploratory decomposition is a special case for algorithms, that search through a predefined space of solutions. In this case, the tasks correspond to a partition of the search space, which can be processed concurrently. An example of exploratory decomposition is the breadth-first tree search used in Section 6.2.

Speculative Decomposition: Another special-purpose decomposition technique is called *speculative decomposition*. In the case when only one of several functions is carried out depending on a condition, these functions are turned into tasks and started before the condition needs to be evaluated. As soon as the condition has been evaluated, only the results of one task are used, all others are thrown away. This decomposition technique is quite wasteful on resources and seldom used.

The different decomposition methods described above can also be combined. As soon as the tasks have been generated, in the second step of Grama's methodology they are mapped onto *execution entities*. An execution entity in this context can either be a thread or a process.

2.1.2 Mapping

Now that the tasks have been created using one of the techniques described in the last section, they must be grouped and distributed across execution entities - appropriately called *mapping*. The target here is to find a mapping that minimizes the overhead associated with concurrent execution. There are two main sources of overhead in this context: the cost of interaction (which includes synchronization and communication) between execution entities and the cost of idle execution entities. Unfortunately, optimizing to bring both costs down at the same time is difficult, as they are conflicting goals.

There are two main classes of mapping techniques for parallel algorithms: *static mappings* and *dynamic mappings*. They are explained in the next two paragraphs.

Static Mapping: When all tasks are known before the algorithm starts, static mapping techniques can be employed. For functional decompositions, it can be employed frequently, yet mapping there is trivial since most of the time there are not many tasks to distribute and in many cases one execution entity per task is available anyways.

A more difficult problem is static mapping for data decompositions. There are techniques for many data structures available in this case, for the sake of brevity we concentrate on two-dimensional arrays here. These techniques can be trivially converted to the one- or multi-dimensional case. Common examples are:

- *Block Distribution:* The array is divided into as many blocks of the same size as there are execution entities. Different shapes of the blocks are possible (e. g. a block can be a row in the array, a column in the array, or a rectangular region spanning multiple rows and columns), but of course the shapes are not allowed to overlap. This is usually the easiest way to map data to execution entities.

- *Block-Cyclic Distribution:* When the work is not distributed equally in the array (e. g. when the calculations for the higher-indexed rows need substantially more work) a block-cyclic distribution may be in order. For this mapping, more blocks than execution entities are created, which are then mapped to execution entities using a round-robin algorithm.

- *Randomized Block Distribution:* When the amount of work to be done is even more irregularly spread across the array, a randomized block distribution can be employed. Like for the former technique, many more blocks than execution entities are created, which are then mapped randomly to execution entities.

Static mapping induces the least overhead and should therefore be used whenever applicable – i. e. whenever the exact amount of tasks and the amount of work done is known before the algorithm starts. However, for many problems it is not applicable and therefore dynamic mapping as described in the next paragraph needs to be used.

Dynamic Mapping: Things are more complicated when the number or size of tasks is not known prior to carrying out the algorithm. This is the case for most of the task decomposition techniques described above. Algorithms with this property are also called *irregular algorithms* and are described in more detail in Section 5.4. There are two ways to manage this case: first by providing a central resource that distributes tasks on demand. Depending on the architecture, this can be either a separate execution entity or a data-structure. Each time an execution entity runs out of work, it queries the resource for another task. If an execution entity generates more tasks than it can handle, it pushes these back to the resource. The second approach is more complicated, as it involves a distributed way to exchange tasks, which is usually difficult to implement correctly.

Dynamic mapping schemes are more costly to implement with regards to programming overhead and also more difficult to program correctly. Often, they are the only sensible way to map tasks to execution entities, though, which is why it becomes important to have easy to use abstractions in place (e. g. the task pools described in Section 5.4).

2.1.3 Synchronization/Communication

The last three points in Grama's methodology can be combined into a single phase: *communication/synchronization*. When task decomposition and mapping to execution entities have been done, the next step is to actually distribute the data to the appropriate execution entities, if that has not been done in the last step already (e. g. for data decompositions). This step is not necessary when working on shared memory architectures, but it may be a good idea to think about data-distribution to increase locality on these architectures as well.

It also involves setting up synchronization points in the program to obey task dependencies or setting up communication operations to exchange (intermediate) results. How this is done depends on the parallel programming system and architecture, therefore we are not going to show it here. This is the point in the methodology shown, where the target architecture can no longer be abstracted away. This is also the point that usually makes the most problems, some of which we are going to look at in the next section.

2.2 Why Parallel Programming is Hard

In this section, some of the reasons why parallel programming is considered difficult are described. We also point to the appropriate sections in this thesis, where our attempts to ease these difficulties are shown.

Added Complexity: In the last subsection, we have described a five-step approach to creating a parallel algorithm. None of these steps are necessary for a sequential program. For most of them, there is little support from tools available. The only thing that really helps is experience. Naturally, these additional and difficult steps are the main reason why parallel programming is considered difficult.

Parallel Programming is Error-Prone: Not only are there additional steps involved when creating parallel programs, but these steps are prone to errors. If the wrong task-decomposition is chosen, you might not see any performance increases from parallel programming. If the wrong mapping is chosen, synchronization overhead may kill performance. And if mistakes are made during the communication/synchronization phase, not only performance may suffer, but the program might not run or produce incorrect results altogether. We are trying to help for this particular problem in Section 3.2, where common mistakes when programming in OpenMP are described, and in Section 4.1, where we show techniques to make some of the mistakes less likely to occur.

Too Little Knowledge of New, Innovative Parallel Programming Systems: There are some parallel programming systems out there that promise to be easier to use. OpenMP is the primary example, but also e. g. functional languages like Erlang or Haskell. The

problem here is that most programmers do not even know about these systems or their strengths and weaknesses. Our efforts to level off this problem are explained in Chapters 3 and 4.

Parallel Programming is Not Yet Mainstream: A lot of information can be found about mainstream languages in books or on the internet, where a lot of tutorials or solutions to common problems can be found. Programming is easier, when there is a lot of help available. The situation is different for most parallel programming systems. There are some books available, but most of them concentrate on numerical algorithms and problems found in High Performance Computing. The number of resources on the internet is also quite thin, when compared to the more general topics of programming and software development. Our efforts to change this situation and make parallel programming easier are described in Sections 3.3 and 4.2.

Compilers Are Not as Well-Tested as Their Sequential Counterparts: Working with a compiler that is not well-tested can be a huge problem. A programmer can spend large amounts of time searching for mistakes in code, when really the compiler is to blame. In our experience, many compilers for parallel code are less well-tested than their sequential counterparts, therefore we have attempted to provide a testing ground for them with our work on AthenaMP described in Chapter 5.

No Good Platform for Parallel Programming Available: There are many tools available today to help programmers of sequential programs. IDEs, debuggers, profilers, correctness tools are all important to make programming easier, and together with the abstractions provided by the language and the libraries available they form a so-called *platform*. For the parallel case, there are tools available for some systems, but they are usually not as good as their sequential counterparts.

Not Enough Powerful Abstractions Available: Programming in assembler is more difficult than programming in a higher-level language. The languages used for parallel programming are often on a very low level, as each communication or synchronization operation needs to be managed in every detail by the programmer. OpenMP helps to raise the level of abstraction, as it hides many details from the programmer. However, it could be even more powerful in some regards, a topic that is explained further in Chapter 6.

A second aspect here is the lack of good libraries. Programming becomes much easier, when the programmer can rely on powerful libraries to encapsulate complex behavior. There are some numerical libraries available for many parallel programming systems, yet others are largely missing. Our contribution to this problem is the AthenaMP library described in Chapter 5.

Lack of Standards/Portability: We have discussed earlier that parallel programming systems are not yet mainstream. Related to this point, there are few standards available that programmers can rely on, especially when changing platforms. OpenMP and Java threads are a big win in this regard, since they enable portable parallel programming, even including the Microsoft Windows platform.

It Is Difficult to Move from a Sequential Program to a Parallel one: In most cases, parallelizing a sequential program is not an incremental process. For many systems, the programmer has to start refactoring code for parallelism, and until that is completely done, there is no way to check any intermediate steps. This is of course difficult. OpenMP has a good approach to this problem, as it is quite easy to add pragmas one at a time without having to refactor the whole program at once.

It Is Hard to Test Parallel Programs: Related to the last paragraph, testing parallel programs is a problem. Most errors can be classified as problems related to parallelism and problems with the actual algorithm. Unfortunately, finding a mistake is difficult when the culprit could be in any of the two sources. OpenMP once again helps, because it enables programmers to turn off parallelism for many programs and still get a correct, sequential one, that can be debugged for errors using the advanced sequential debugging tools available.

It should be clear from the last few paragraphs why OpenMP was chosen as basis for our research. Many of the problems inherent in other parallel programming systems have been solved there, others could be leveled off by our work when building on OpenMP.

2.3 An Overview of Available Parallel Programming Systems

In this section, the two most common classes of parallel programming systems are introduced: *message passing systems* and *threading systems*. We are going to look at three important properties of these systems: how execution entities (processes or threads, respectively) are created, how work is distributed to these entities and how communication/synchronization is achieved. This will help to further motivate our choice for OpenMP as basis for our research. OpenMP is described in the next Section 2.4.

2.3.1 Message Passing Systems

Message passing systems operate under the premise that communication between different execution entities happens only through explicit message exchange. They are also commonly called *Shared-Nothing Systems*, because contrary to threading systems they do not share state between execution entities. The most well-known message passing system

is the *Message Passing Interface* (MPI) [GNL98]. An older one found in high performance computing is the *Parallel Virtual Machine* (PVM) [Gei94]. Recently, a functional message passing system called Erlang is also talked about often [Arm07].

The term *execution entity* in this context does not necessarily mean a process as defined for operating systems. Quite often, message passing systems use operating system processes, but there are also systems out there using threads (e. g. Erlang).

How execution entities are created is usually dependent on the system used. For MPI, you may have to specify the number of processes to use in a startup file for your program or use a command line parameter. In MPI-2, PVM or Erlang it is also possible to start execution entities at runtime using explicit API-calls.

Work is distributed in message passing systems manually. The programmer has to take care of decomposing the problem into tasks and mapping the tasks to execution entities explicitly, e. g. by sending work to be done out to different execution entities using explicit messages. The same is true for communication and synchronization: everything is done through explicit messages.

Here is a short list of advantages that are often claimed for Message Passing Systems:

- they are less prone to errors
- they are well suited to architectures without shared memory (such as clusters), but can also take advantage of architectures, where shared memory is available
- programmers must manage data distribution manually, resulting in programs that exploit locality well
- result in well-scaling programs

And here are the disadvantages:

- programmers must manage data distribution manually, which is difficult for beginners
- there is usually no way to do incremental parallelization

Managing data-distribution by hand is both an advantage and a disadvantage: it is an additional step that needs to be done in the program, but once the programmer has done it, usually leads to better locality, scalability and performance.

2.3.2 Threading Systems

Threading Systems are sometimes called *Shared-Everything* systems, because memory and therefore the state of the program is mostly shared between the execution entities. In most cases, execution entities are threads for these kinds of systems. This makes communication appear easy, because all that needs to be done is to change a shared variable or

data-structure and all other threads can see the result. Unfortunately, it is not that easy, as concurrent accesses to the same location in memory need to be protected. This can be done in various ways (e. g. using locks), but if it is forgotten leads to unspecified behavior in most threading systems (a very frequent mistake, see Section 3.2 for details). The most widely-known threading systems are probably POSIX threads [But97], Java threads [GPB+06] and .NET threads. We are not counting OpenMP here, as it is on a semantically higher level than the others (e. g. there is no need to manage threads explicitly, see Section 2.4 for details).

In threading systems, threads are created via explicit API calls at runtime. Work distribution used to be a largely manual process, but library support starts to become available for some systems (e. g. through the `Executor`-class in Java threads). Communication happens through shared data structures that need to be explicitly protected from concurrent access by the programmer. Constructs for synchronization are usually available, e. g. barriers, mutex variables, different versions of locks and condition variables.

Here are some commonly cited advantages of threading systems:

- natural model for systems with shared memory
- no need to distribute data manually
- usually easier to get started with

And here are the disadvantages:

- not suited for distributed memory architectures, except through distributed shared memory systems
- communication through shared memory appears easy, but is error-prone

In the next section, we are going to look at OpenMP and how it differs from threading systems.

2.4 An Introduction to OpenMP

OpenMP is a parallel programming system that provides a specification for shared memory communication. In order to understand the contents of Chapters 5 and 6, it is necessary to know the basics of this system, therefore a short introduction is presented here. Further coverage is provided by a tutorial from Lawrence Livermore National Laboratory [Lab03], in the specification [Ope05], or in the books by Chapman et al. [CJP07] and Chandra et al. [CDK00].

This section is structured as follows: We start with general information on OpenMP in Section 2.4.1. Then, Section 2.4.2 highlights how threads are created. In Section 2.4.3, we explain how to synchronize threads and Section 2.4.4 is about sharing work between

threads. The data environment is explained in Section 2.4.5. Section 2.4.6 gives a short overview of the runtime library routines provided by OpenMP and Section 2.4.7 describes how environment variables can be used to alter the behavior of the runtime system. In Section 2.4.8, the memory model is described, while Section 2.4.9 describes some issues with regards to thread safety.

2.4.1 General

The first OpenMP specification was released in 1997 for Fortran. At a time, when many hardware vendors had their own directives for exploiting data-parallelism on their machines, this was a big step forward. One year later, the specification for C and C++ was released. At the time of this writing, the specification is at version 2.5, released in 2005, with version 3.0 expected in early 2008. Although the specification supports three languages, for the rest of this introduction only C++ is used.

OpenMP is an Application Programming Interface (API) that consists of directives, runtime library routines and environment variables. Some of the advantages of OpenMP have already been described in Section 2.2, the major ones are in no particular order:

- high level of abstraction: although OpenMP builds on threads, the user does not need to create or manage them explicitly
- portability: runs on a variety of architectures, including most UNIX platforms and various kinds of Windows operating systems and is specified for C/C++ and Fortran
- performance: performs about equal to lower-level systems most of the time
- standardization: specification supported by more than 15 vendors
- simplicity: employs only a small set of directives, runtime routines and environment variables
- allows incremental parallelization: adding parallelism step by step is possible
- allows to turn parallelism off: most OpenMP-programs can be compiled without OpenMP-support and result in a valid sequential program
- tool support: correctness tools, debuggers, and profilers are available

Of course, the system also has some disadvantages. OpenMP can neither provide facilities for distributed memory programming, nor is a well performing program on one architecture necessarily going to perform well on a different one. In some areas, it also lacks the level of control that low-level threading systems usually offer.

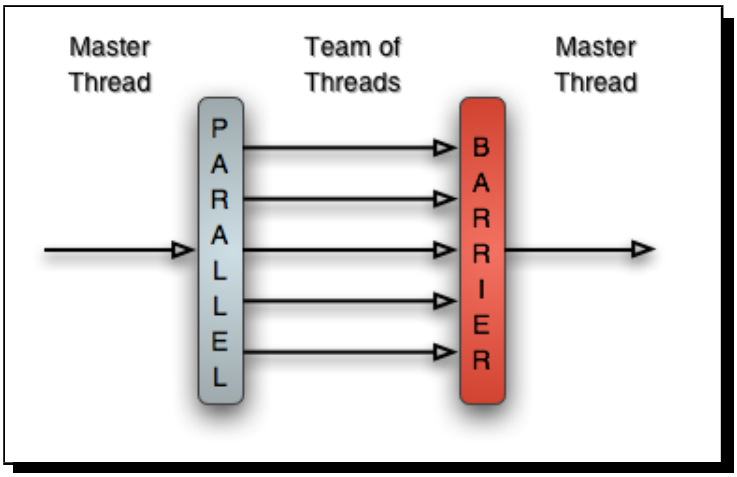


Figure 2.1: Program execution scheme for `OpenMP` programs

```
1  #include <iostream>
2  #include <omp.h>
3
4  int main(int argc, char**argv)
5  {
6    /* Spawn a team of 5 threads */
7    #pragma omp parallel num_threads (5)
8    {
9      std::cout << "Hello, I am a thread!" << std::endl;
10   } /* All threads join master thread here and terminate */
11 }
```

Figure 2.2: A parallel region in OpenMP

2.4.2 Creating Threads

In contrast to simple thread-based programming models, OpenMP imposes a structure on all threads. The program starts its life with a single thread, the *master thread*. Parallelism is expressed using a so-called *parallel region*. A parallel region is a block of code that is simultaneously executed by a *team of threads*. The team consists of one master thread and many *worker threads* and is spawned by a `parallel` directive. At the end of the parallel region, all threads in the team wait for each other in an implied *barrier*. This way of expressing parallelism is called *fork/join-model* and is represented graphically in Figure 2.1. A very basic program showing how to create threads is shown in Figure 2.2.

The program will print a message to the standard output device five times when executed. The `num_threads` clause is one way to specify, how many threads should be part of a team, other ways to influence this number are described later.

```
1 /* Spawn a team of 5 threads */
2 #pragma omp parallel num_threads (5)
3 {
4   /* Each worker thread spawns a new team of two threads */
5   #pragma omp parallel num_threads (2)
6   {
7     std::cout << "Hello, I am a thread!" << std::endl;
8   } /* implicit barrier for inner teams */
9 } /* implicit barrier for outer team */
```

Figure 2.3: Nested parallel regions in OpenMP

```
1 #pragma omp parallel num_threads (5)
2 {
3   #pragma omp critical
4   {
5     std::cout << "Hello, I am a thread!" << std::endl;
6   }
7 } /* All threads join master thread here and terminate */
```

Figure 2.4: The `critical` directive

Parallel regions in OpenMP can also be nested inside each other. If a new `parallel` directive is encountered by a team of threads, each worker-thread becomes the master of a new team. This is especially useful for libraries. Unfortunately, nested parallelism is not a required part of the OpenMP-specification as of the time of this writing and there are still compilers out there that serialize the inner regions. An example of nested parallelism is shown in Figure 2.3, where a total of ten threads should be spawned if nested parallelism is supported in the compiler.

Although the two programs shown in this section are very small, both have an error inside: `std::cout` is a resource that is shared among all threads. When multiple threads access a shared resource at the same time, it must be protected from concurrent access (see Grama et al. [Gra03] for the reasons why this is the case). In the next few paragraphs, we are going to describe ways to guarantee that.

2.4.3 Synchronization

Several constructs are available to synchronize threads in OpenMP. The most important one is the `critical` directive. It makes sure that a block of code is only executed by one thread at a time. This makes sure that shared resources can be accessed safely by multiple threads, as can be seen in our modified example in Figure 2.4.

```
1 int i = 0;
2
3 #pragma omp parallel num_threads (5)
4 {
5   #pragma omp atomic
6   ++i;
7 }
8
9 std::cout << i << " threads were created!" << std::endl;
```

Figure 2.5: The `atomic` directive

```
1  omp_lock_t my_lock;             /* define a lock */
2  omp_init_lock (&my_lock);       /* initialize lock */
3
4  #pragma omp parallel num_threads (5)
5  {
6    omp_set_lock (&my_lock);    /* set the lock */
7    std::cout << "Hello, I am a thread!" << std::endl;
8    omp_unset_lock (&my_lock); /* release the lock */
9  }
10
11 omp_destroy_lock (&my_lock);    /* destroy lock */
```

Figure 2.6: Locks in OpenMP

With the changes applied in this figure, the program is safe. We show another example of how synchronization can be employed in Figure 2.5. The program snippet shown in this figure counts the number of threads that were created and stores the result in the variable `i`. Access to `i` is protected by an `atomic` clause, which is a second synchronization clause in OpenMP. It guarantees that the update of the protected memory location happens in a single machine-step, thereby making sure no other thread can interfere. Atomic operations are usually faster than employing a critical section, yet they can only be used on a very limited set of arithmetic operations.

Locks are a very versatile synchronization method in OpenMP. Similar in functionality to the `critical` directive, they offer more flexibility. For example, where critical sections can only be using inside one function, locks can be employed across function borders. Locks also come in a nested variety that is useful if you need to lock the same lock multiple times in a row, e. g. from multiple functions. This usefulness comes at a cost, though: locks need to be initialized (`omp_init_lock`) before use and destroyed (`omp_destroy_lock`) after use, which is frequently forgotten by programmers, see Section 5.1 for a solution. An example of how to use locks is shown in Figure 2.6.

```
1 #pragma omp parallel num_threads (5)
2 {
3   #pragma omp for schedule (static)
4   for (int i=0; i<20; ++i) {
5     do_work (i);
6   }
7 }
```

Figure 2.7: The `for` work-sharing construct

We have mentioned the fourth synchronization construct in OpenMP already: the *barrier*. When a thread reaches a barrier, it has to wait until all other threads in his team have reached this point in the program as well. Only after then, all threads in the team can continue. In OpenMP, there are implicit barriers like the one at the end of the parallel region and there are explicit barriers that are specified using the `barrier` directive. There are even more synchronization constructs in OpenMP, e. g. the `master` and `ordered` directives, but they are beyond the scope of this introduction.

2.4.4 Work-sharing

In the last sample programs shown, all threads carried out the same work. Most of the time, this is not desired, as work should be divided among threads. This brings us into the realm of work-sharing. The most important work-sharing directive in OpenMP is the `for` directive. It is put before a for-loop and makes sure that all iterations of this loop are distributed among all threads in a team. An example will make this clearer in Figure 2.7.

In the example, each threads carries out exactly four iterations of the loop and calls the `do_work` method exactly four times, with a different `i` each time. We have set the `schedule` clause to static in this case to use static mapping of the iterations to threads, dynamic mapping is also available.

A second work-sharing construct available in OpenMP is the `sections` directive. In the block of code associated with it, one or multiple `section` directives can be listed. The code inside the block of code associated with each `section` construct is carried out by exactly one thread (not necessarily a different one for each section), making this the perfect directive for functional decomposition. An example is provided in Figure 2.8.

In the example, the `do_task_one` and `do_task_two` functions are carried out exactly once. For the program shown, they will most likely be carried out by two different threads, but this is not required by the specification.

The last work-sharing construct available for C++ is called `single`. The block of code associated with each `single` directive is carried out by exactly one thread. Figure 2.9 shows an example.

```
1  #pragma omp parallel num_threads (2)
2  {
3    #pragma omp sections
4    {
5      #pragma omp section
6      {
7        do_task_one ();
8      }
9      #pragma omp section
10     {
11       do_task_two ();
12     }
13   }
14 }
```

Figure 2.8: The `sections` work-sharing construct

```
1  #pragma omp parallel num_threads (5)
2  {
3    do_some_work ();
4
5    #pragma omp single
6    {
7      do_cleanup ();
8    }
9  }
```

Figure 2.9: The `single` work-sharing construct

```
1  #pragma omp parallel num_threads (2)
2  {
3    if (omp_get_thread_num () == 0)
4    {
5      do_task_one (); /* the master thread does this */
6    } else if (omp_get_thread_num () == 1)
7    {
8      do_task_two (); /* one worker thread does this */
9    }
10 }
```

Figure 2.10: Manual work-sharing

In the example, the `do_some_work` function is carried out by five threads. Only one thread executes the `do_cleanup` function. All work-sharing constructs have an implicit barrier at the end. It can be omitted by specifying the `nowait` clause after the work-sharing construct.

Besides using directives for work-sharing, it is also possible to use library routines. The `omp_get_thread_num` function returns a unique id for each thread in a team that can be used to map tasks to threads. This is called *manual work-sharing* and is shown in Figure 2.10.

2.4.5 Data Environment

OpenMP is a parallel programming system for shared memory, therefore some considerations are in order about which variables are shared between the threads, and which are private to each thread. The default is shared. Variables are private to each thread only, if they are:

- defined locally inside a parallel region
- defined locally inside a function called from within a parallel region
- loop index variables of loops parallelized with a `for` work-sharing construct
- made private by using a `private`, `firstprivate`, or `lastprivate` clause on a `parallel` directive or a work-sharing directive
- made private by using a `threadprivate` directive

2.4.6 Runtime Library Routines

There are several runtime library routines in OpenMP. The most commonly used are the following:

- `omp_set_num_threads`: sets the number of threads to use in future parallel regions
- `omp_get_num_threads`: returns the number of threads currently in the team
- `omp_get_thread_num`: returns a unique id of a thread in the team
- `omp_get_max_threads`: returns the number of threads used if a new parallel region was created at this point in the program
- `omp_get_num_procs`: returns the number of processors available to the program

More routines have already been introduced earlier in this section (e. g. the ones for locks). There are more functions available that are beyond the scope of this introduction.

2.4.7 Environment Variables

Besides the directives and runtime library routines introduced earlier in this section, the third part of OpenMP consists of environment variables. They can be used to influence the behavior of the OpenMP runtime system. An example is the `OMP_NUM_THREADS` environment variable, that is used to control how many threads are used in a parallel region in the absence of the `num_threads` clause and the `omp_set_num_threads` function. They are not used in this thesis, though, therefore they are not further explained here.

2.4.8 The Memory Model

The memory model of OpenMP is complicated and therefore frequently skipped in tutorials and introductory texts. To discuss it fully is beyond the scope of this section, see the excellent paper by Hoeflinger and de Supinski [HdS05] for details. We are going to concentrate on one aspect of the memory model here that has particular importance for this thesis. The specification states:

> *If multiple threads write to the same shared variable without synchronization, the resulting value of the variable in memory is unspecified. If at least one thread reads from a shared variable and at least one thread writes to it without synchronization, the value seen by any reading thread is unspecified.* ([Ope05, p. 11])

This has the implication, that even to read a shared variable from memory, synchronization is necessary between threads. The `flush` directive that is frequently employed to achieve this is not sufficient, except in rare edge-cases.

2.4.9 Thread Safety

As soon as functions are called from multiple threads at the same time, the concept of thread safety becomes important. Basically, a function is thread-safe, if it can be called from multiple threads concurrently and still returns correct results. The concept becomes even more important, as soon as functions from libraries are called. The OpenMP specification states with regards to thread safety:

> *All library, intrinsic and built-in routines provided by the base language must be thread-safe in a compliant implementation. In addition, the implementation of the base language must also be thread-safe (e.g., ALLOCATE and DEALLOCATE statements must be thread-safe in Fortran). Unsynchronized concurrent use of such routines by different threads must produce correct results (though not necessarily the same as serial execution results, as in the case of random number generation routines).* ([Ope05, p. 13])

These are far-reaching guarantees, that are unfortunately not kept by today's implementations, see Section 6.3 for details.

2.5 Chapter Summary

In this chapter, we have described some foundations for our work. We have shown the general steps involved in parallelizing an algorithm in Section 2.1: task decomposition, mapping and synchronization/communication. Common problems encountered in the field have been described in Section 2.2. Also in this section, we have provided reasons why we have chosen OpenMP as the basis for our research.

An overview over the two main classes of parallel programming systems in actual use today has been given in Section 2.3, which described the differences between message passing systems and threading systems. The chapter closed with an overview of OpenMP in Section 2.4. This system is relied on heavily in the rest of this thesis.

Chapter 3

Evaluating the State of the Art

> *It's not what you don't know that hurts you, it's what you know that just ain't so.* (Satchel Paige)

Before work was started to improve an existing parallel programming system, we took the time to conduct some research on what is the state of the art in the field today. We felt that only by knowing the systems in use today and the problems associated with them, we could achieve our goal to make parallel programming easier.

Unfortunately, the state of the art is difficult to grasp entirely and in a timely fashion, since it is constantly evolving and changing. Therefore, we have tried to improve our knowledge about the topic using three main instruments: a survey, a study on frequently made mistakes and a wiki for parallel programmers.

Section 3.1 describes a survey carried out among parallel programmers during most of the year 2005. Although not statistically significant, data could be gathered e. g. about what parallel programming systems and languages were used and known.

In Section 3.2, data on common programming mistakes made by our students while learning their first parallel programming system were collected over the course of two semesters. These data were used to compile a list of best practices for OpenMP-programmers, which is described later in Section 4.1. Although the list of mistakes is specific to OpenMP, at least some of them are common in other systems, as well.

Finally, in Section 3.3, an attempt to create a resource on the web to gather experiences in the field of parallel programming is described – the *Parawiki*. This collaborative resource has been setup in 2005 as well, as a single meeting and learning point about the strengths and weaknesses of the available parallel programming systems.

The chapter is closed with a short summary of what has been accomplished in Section 3.4. The relevance of this chapter in the context of this thesis is highlighted in Figure 3.1.

3.1 A Survey on Parallel Programming

The work presented in this section is derived from a technical report [SPL07]. Very few data seem to exist on the usage of parallel programming systems by application programmers. Some presumptions are floating around the community, e.g.:

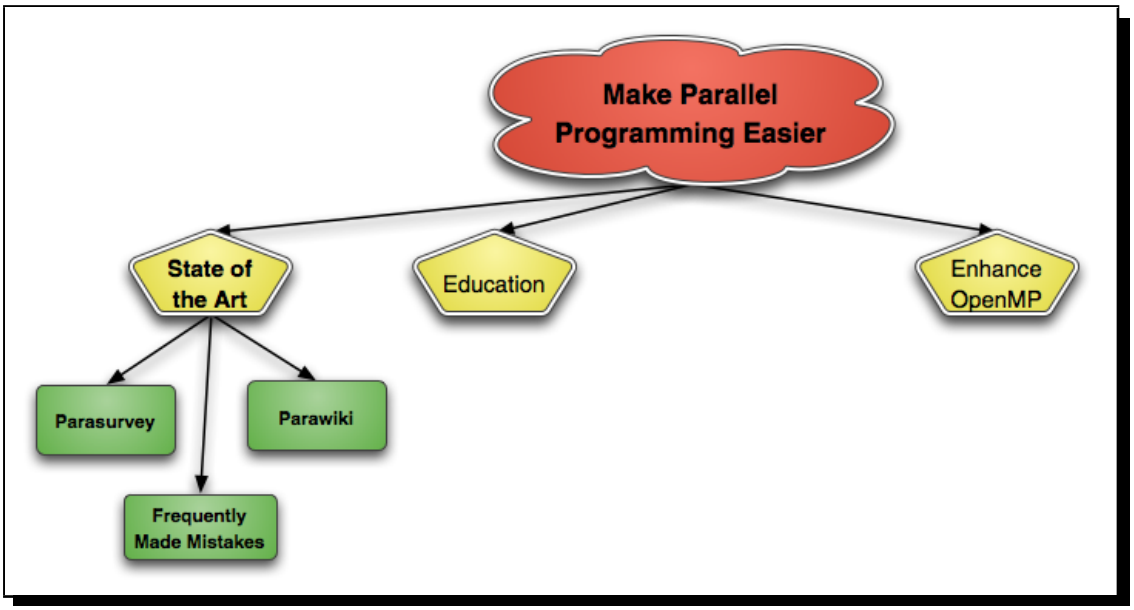


Figure 3.1: Objectives, aims and contributions of this thesis (evaluation)

- The majority of parallel applications uses MPI nowadays
- Java threads is growing strong

There are few real data available to back up any of these claims, though. Studies with a limited number of participants have been conducted in the past (e.g. by Sodan [Sod05]), yet their intentions were different from ours. Books about the different parallel programming systems, such as by Leopold [Leo01b], show which systems are available, but have no data about which ones are in actual use.

Furthermore, different approaches to parallel programming like algorithmic skeletons or parallelizing compilers are raising a lot of interest in the scientific community, but no data about how many application programmers actually know and/or use these systems are available.

Therefore, we have conducted a non-representative survey among the programmers of parallel applications. Section 3.1.1 explains how the survey was carried out. Section 3.1.2, the main part of this section, goes through the questions, shows the submitted answers, and comments on their relevance. We also wanted to combine several questions with each other to show even more detailed analysis, but decided against it due to the limited amount of participants.

It is important to point out, that each and every observation we sketch out in this part of the work is merely a hypothesis. Our findings are not backed up statistically and should therefore be treated merely as data points, not as a scientific proof. We believe, however, that any data is better than no data on the subject at all, as it can at least be used to come up with hypotheses, which can be proved later, possibly using a more representative study.

3.1.1 Survey Methodology

The survey was carried out online, with a simple HTML form, and a PHP script to put the submissions into a MySQL database. It was hosted on the web-server of our research group, where it can still be viewed for reference [Sue05a]. Submissions were accepted from February 2nd to November 7th, 2005. Visitors of the site were asked to fill out the survey only if they were developing parallel applications. Of course, we had no way to verify that. If participants submitted their answers along with a working e-mail address, they could win one of two $50 gift certificates from amazon.com. We also promised to deliver a report of the results, which was done in February 2006 as a technical report [SPL07] and mailed out to all participants who had expressed interest in being notified.

To raise more interest for the survey and generate submissions, we have sent messages to several discussion groups, once shortly after the start of the survey and once shortly before the end:

- **Usenet:** comp.parallel, comp.parallel.mpi, comp.parallel.pvm, comp.sys.super, aus.computers.parallel, comp.programming.threads
- **Mailing Lists:** beowulf@beowulf.org, lam@lam-mpi.org, omp@openmp.org, compunity@lists.rwth-aachen.de, bspall@bsp-worldwide.org
- **Forums:** CodeGuru Forum, Intel Developer Services Forum
- **Websites:** slashdot.org

It was necessary to send out these messages, because otherwise our survey would not have been noticed by enough members of the parallel programming community to make the results meaningful. The distribution of these messages influenced our results, though. An example: we did not find a discussion forum for Java threads or for algorithmic skeletons at the time. If we had found one, the results for these systems probably would have been higher. For this reason alone, all results of this survey should be considered merely as data points, as they are not in any way statistically sound. For statistical significance, it would have been necessary to sample a proportional part of the parallel programming population, and we know of no way to do so (at least not within our budget). It is for this reason, that no statistical measures were applied to the data in this section.

Other sites might have reported on the survey as well. On the Erlang mailing list, a link to the survey has been posted, which may have significantly increased the participation from this community. We got over 30 responses from Erlang programmers, whereas few other communities not specifically targeted by the survey were mentioned more than three times.

Before the data was processed and evaluated, some entries had to be disqualified. Firstly, we had four double submissions, identifiable by the submitted name and email

address. Only the last submission was taken into account in these cases. Secondly, two of our students participated in the survey, although all they know about parallelization was from our classes. Since we did not want to influence the results ourselves, these submissions were taken out as well. This left us with 256 out of 262 submissions.

Of course, we cannot rule out the possibility entirely that more of our students participated or that there were more double submissions, since a valid e-mail address or name was not required on the submission form. Judging from the IP-addresses of our participants, this has at least not happened on a noticeable scale, though.

3.1.2 Survey Analysis

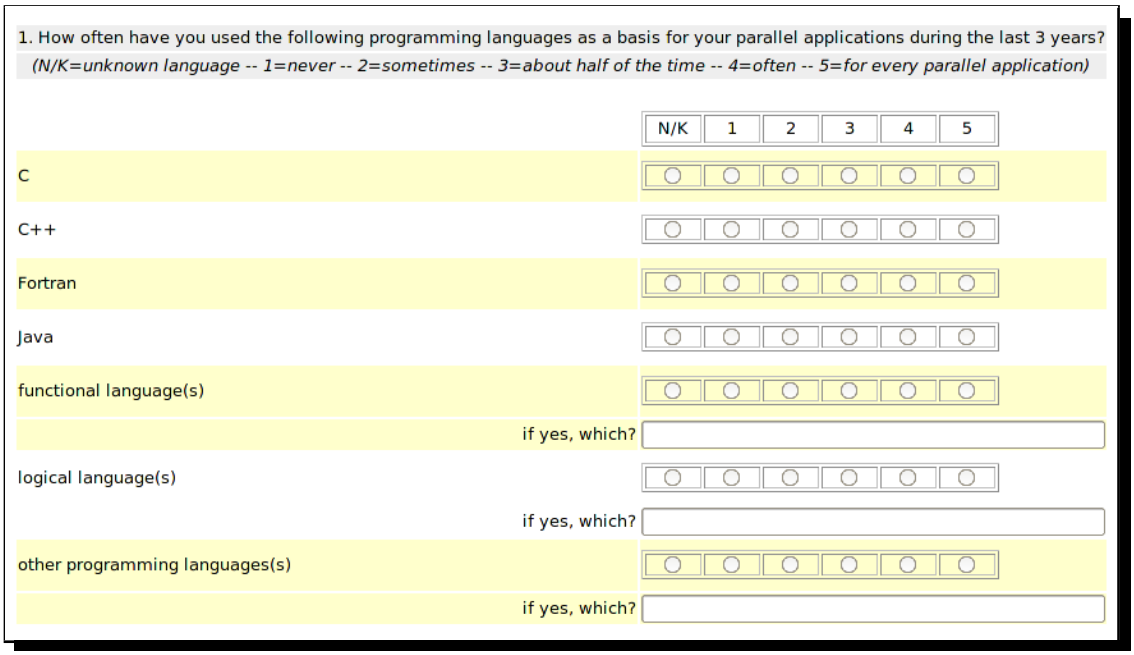
In this section, we present the results of the survey and add some analysis to each question. The first question considered languages for parallel programming, while the second question was about parallel programming systems. Question three evaluated which operating systems parallel applications were developed for, question four shows our data about which hardware platforms were targeted. In order to learn more about the participants of the survey, we have collected data on the organizations they worked in (question five), the average time they spent on parallel programming (question six), and the specific field they worked in (question seven). Finally, question eight highlights the problems people had with parallel programming. The last question of the survey, which provided the possibility to add remarks, was seldom used by our participants, and is therefore omitted here.

Question 1 – Programming Languages

The first question was formulated as shown in Figure 3.2. Results are depicted in Figure 3.3. The left graph labeled *Usage* is calculated by weighting the submissions like this: *N/K* and *never* were not counted, *sometimes* was counted as is, *about half the time* with a factor of 2, *often* with a factor of 3 and *for every parallel application* with a factor of 4. The height of the bar for each language was the sum of the weighted submissions. This graph presents a view on which languages were actually used by parallel programmers. We stick to this definition of *Usage* throughout this section, although it is of course somewhat arbitrary.

In the right graph in Figure 3.3 labeled *Publicity*, all submissions except the votes for *N/K (not known)* were counted. It shows, how many members of our survey group knew the language.

C came out as the winner in both graphs for our specific survey group, followed by C++ and Fortran. Parallel programming with logical languages was obviously not very widespread (or if it was, we have not managed to reach this group). The numbers for Java are interesting. As can be seen in the publicity-graph, many participants knew Java, almost as many as Fortran and more than the functional languages, yet when it comes to actually using it, Java fell behind Fortran by a great margin and was even surpassed by the


1. How often have you used the following programming languages as a basis for your parallel applications during the last 3 years?
(N/K=unknown language -- 1=never -- 2=sometimes -- 3=about half of the time -- 4=often -- 5=for every parallel application)

	N/K	1	2	3	4	5
C	○	○	○	○	○	○
C++	○	○	○	○	○	○
Fortran	○	○	○	○	○	○
Java	○	○	○	○	○	○
functional language(s)	○	○	○	○	○	○
if yes, which?						
logical language(s)	○	○	○	○	○	○
if yes, which?						
other programming languages(s)	○	○	○	○	○	○
if yes, which?						

Figure 3.2: Survey question 1

functional languages. This trend can also be observed in the first part of Figure 3.4, where the distribution of votes for Java is shown. Note the large number of votes for *never*, meaning the programmer has heard about the language but was not using it for parallel programming. Obviously the hype around Java has managed to make the language known to many programmers, but has not convinced too many of them to actually use it for parallel programming.

The second part of Figure 3.4 is interesting as well. It is the distribution of votes for functional languages. Obviously there were two camps of parallel programmers: the ones who did not use functional languages at all (this is the larger camp and it contains the votes for *N/K* and *never*), and the camp that used them *often* or even *for every application*. As can be seen in the graph, there were not many votes in-between the two (for *sometimes* or *half the time*), which seems to confirm the common belief that once a programmer starts using functional languages, he will never go back to using any other language.

The outcome of these graphs could be explained in part by the discussion groups we contacted for this survey, as many of the parallel programming systems for which we found and contacted mailing lists (e.g. MPI, OpenMP, PVM or POSIX threads) have C/C++/Fortran as their base languages. We could, however, compare the results of these languages, and C emerged as the clear winner between the three.

But what about the other languages? As can be seen in Figure 3.2, participants had the ability to enter their own choices of languages into various input fields. Figures 3.5 and 3.6 show, which languages have been entered. We only present languages that were submitted more than three times, since otherwise the graph would have become too big. We have

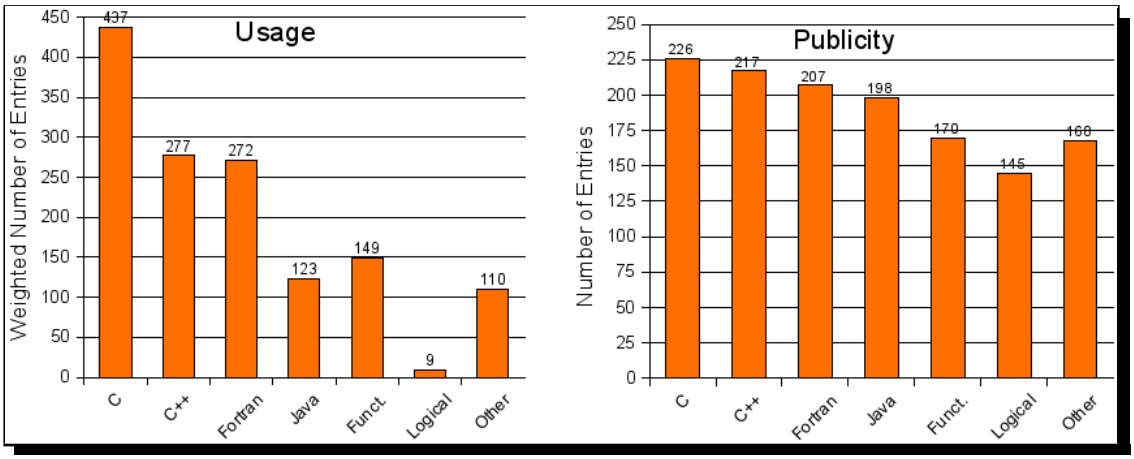


Figure 3.3: Parallel programming languages

also decided against dividing the graph into categories (functional, logical, others), for the sake of clarity and because the distinction does not lead to any new insights here.

We have already told you a possible reason for the relatively high numbers for Erlang when compared to the other languages in Section 3.1.1. What could be deduced from these numbers, though, is that Erlang had an active user community.

It would be unfair to compare the languages explicitly mentioned in question one with the other languages entered in textboxes. If we would have explicitly asked for e.g. Python in this question, it would have been voted higher. Therefore, the languages depicted in Figures 3.3 and 3.5 can only be compared fairly to the languages in their respective figures.

All of these graphs together gave us at least some indications about our initial question, which languages are known and in use for parallel programming today. All of C/C++/Fortran and Java were used and known to the community at large, while at least a few other languages (starting with Erlang, but also including languages like Perl and Python) were on the radar of some parallel programmers.

Question 2 – Parallel Programming Systems

The second question was phrased as:

How often have you used the following parallel programming systems during the last 3 years?

The possible answers as well as their respective number of submissions (weighted exactly as explained in Section 3.1.2) are shown in Figures 3.7 and 3.8.

The results of the usage graph seemed pretty clear: MPI won by a great margin, for our survey group, followed by POSIX threads and OpenMP. We suspect that 5 years ago, the numbers for PVM would have been much higher, but for now the battle for the predominant message passing system seems to have been settled in favor of MPI. The high

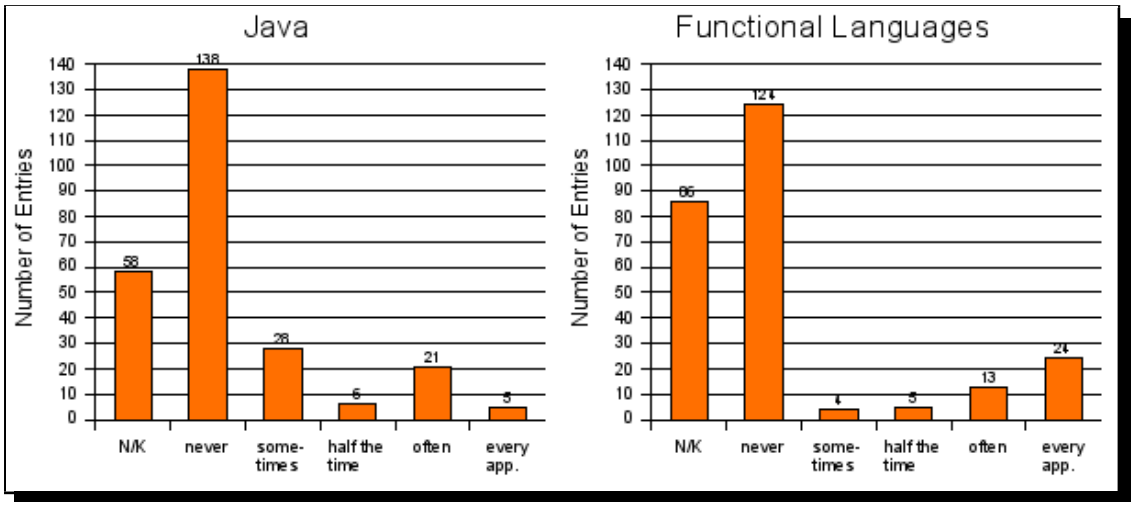


Figure 3.4: Parallel programming languages in detail

usage numbers for MPI were due to a large number of exclusive users, as can be seen in Figure 3.9. There were other systems which are also widely known (see Figure 3.8), but all of them fell behind in actual usage.

For shared memory programming, POSIX threads and OpenMP were close together, and it will be interesting to see, whether or not one of the systems manages to become the dominant one, especially since their usage distribution was about equal (see the bottom of Figure 3.9). High Performance Fortran (HPF) and BSP did not appear to play a significant role for shared memory programming, for our survey group. The picture for Java threads was similar to what has been observed in Section 3.1.2. The system was widely known, yet seldom used.

We were somewhat surprised by the low numbers for algorithmic skeletons. These are generating a lot of attention in the research community, yet their publicity and even more usage were low, in our survey group. We were not even able to create a hypothesis about which system is the most widely known, because all three submitted systems (P3L, DatTeL, FreePOOMA) have been mentioned only once!

A similar observation could be made for automatically parallelizing compilers. Compared to other systems, their publicity and usage were relatively low. This was surprising, since they were commonly available and all it took to use them was setting a switch in the command line. A detailed list of the parallelizing compilers submitted is shown in Figure 3.10. Note that, once again, only compilers that were submitted more than three times are depicted.

Distributed shared memory systems seemed to suffer a similar fate in our survey group. They were heavily researched, yet not widely known or used. Only SGI Altix and IBM pSeries have been mentioned more than three times, and both are actually hardware architectures with built-in support for distributed shared memory. No pure software solutions

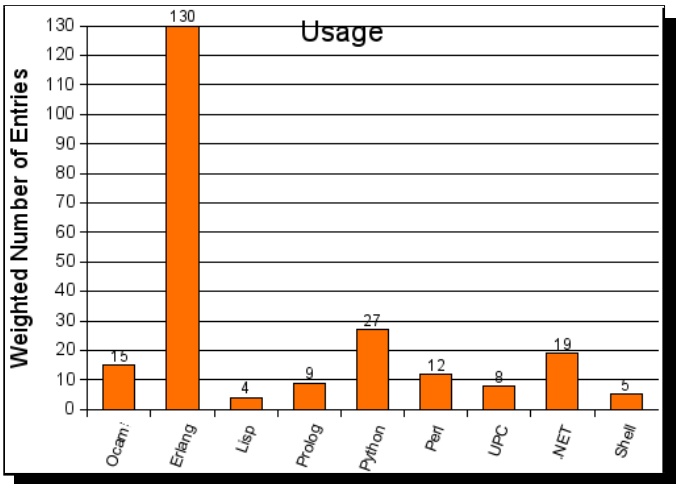


Figure 3.5: Usage of other languages for parallel programming

were mentioned more than three times, which indicates that they were not widely accepted or used in our survey group.

The other parallel programming systems submitted included a wide variety of systems, yet only Erlang (26 submissions, accumulating to a usage of 89) and .NET (four submissions with a usage of 10) managed to be mentioned more than three times. Noteworthy is the fact that Erlang is one of the very few programming languages for which parallelism is an integral part of the language, and it therefore had high submissions for both questions one and two. When interpreting the numbers, the systems entered by users can only be compared fairly among themselves, and not to systems explicitly asked for in question two.

Using these figures and observations, let us reconsider our initial question about the usage and publicity of parallel programming systems: It is safe to conclude that MPI, POSIX threads, OpenMP, and Java threads were widely known and used for parallel programming, with MPI being the most popular programming system from our submissions (but remember that we have not sampled a proportional part of the parallel programming population!). Algorithmic skeletons, parallelizing compilers and distributed shared memory systems did not seem to be widely accepted and used (or, if they were, we have not managed to reach their user communities). Erlang had significant usage numbers, but we can not fairly judge its popularity, from our survey methodology.

Question 3 – Operating Systems

Our next question was:

- What operating systems are your parallel applications intended for?

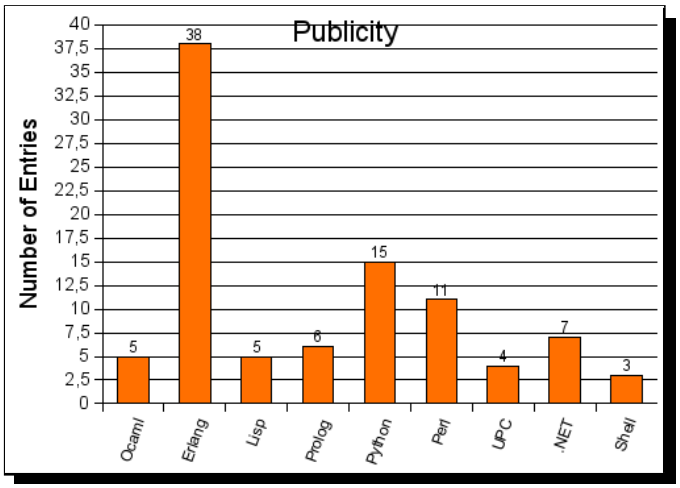


Figure 3.6: Publicity of other languages for parallel programming

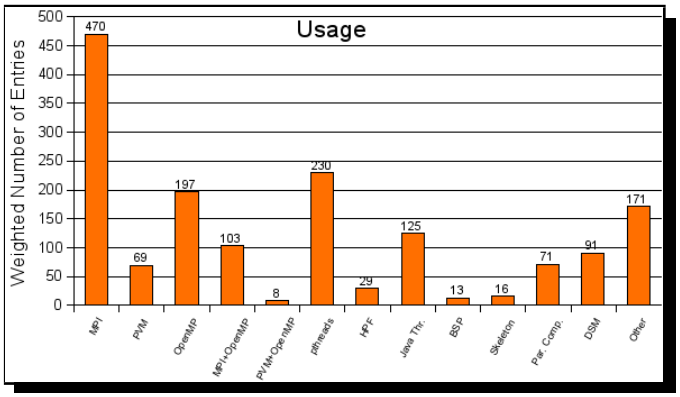


Figure 3.7: Usage of parallel programming systems

The possible answering options and the submissions are shown in Figure 3.11. Note that only radio buttons were provided for the answers, therefore it was not possible to distinguish between usage and publicity here as for questions one and two.

Linux was the dominant operating system for this survey by a wide margin, followed by Solaris and Windows. It seems safe to conclude that most parallel applications were developed for a flavor of UNIX, as even the operating systems that were hand-submitted consist mainly of variants of UNIX (as can be seen in the right part of Figure 3.11). It would be interesting to see if these numbers changed in time with the introduction of Windows for Supercomputers and the increase in parallel systems on the desktop due to the advent of multi-core processors.

Question 4 – Hardware Platforms

The next question we wanted to investigate was:

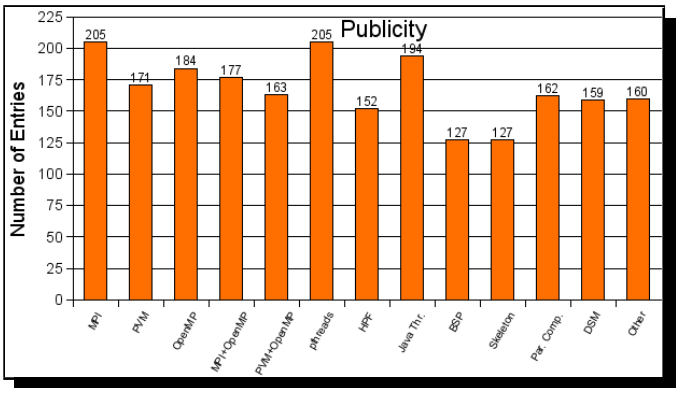


Figure 3.8: Publicity of parallel programming systems

- What hardware platforms are your parallel applications intended for?

Possible answers were shared memory architectures (e.g. SMPs, NUMAs), distributed memory architectures (e.g. clusters) and grids. Multiple answers were allowed. Distributed memory architectures won our survey with 199 submissions, followed by shared memory architectures with 160 submissions. Grids came in at a distant third place with only 47 submissions.

Question 5 – Organization

In this question we wanted to know:

- In what kind of organization are you developing parallel applications?

Answers were: University (122 submissions), Company (88 submissions), Other Research Institute and Other Organization (6 submissions total). This question was most useful to put the results we have gathered so far into perspective, as obviously our survey was biased in favor of systems used at universities.

Question 6 – Time spent

Our next question was:

- How much of your programming time is spent on the development of parallel applications?

Answers were: $0-20\%$ (81 submissions), $20-40\%$ (48 submissions), $40-60\%$ (46 submissions), $60-80\%$ (37 submissions) and $80-100\%$ (no submission). Multiple answers were not allowed. There either were no people who are employed to work on parallel projects full-time, or we have not managed to reach them.

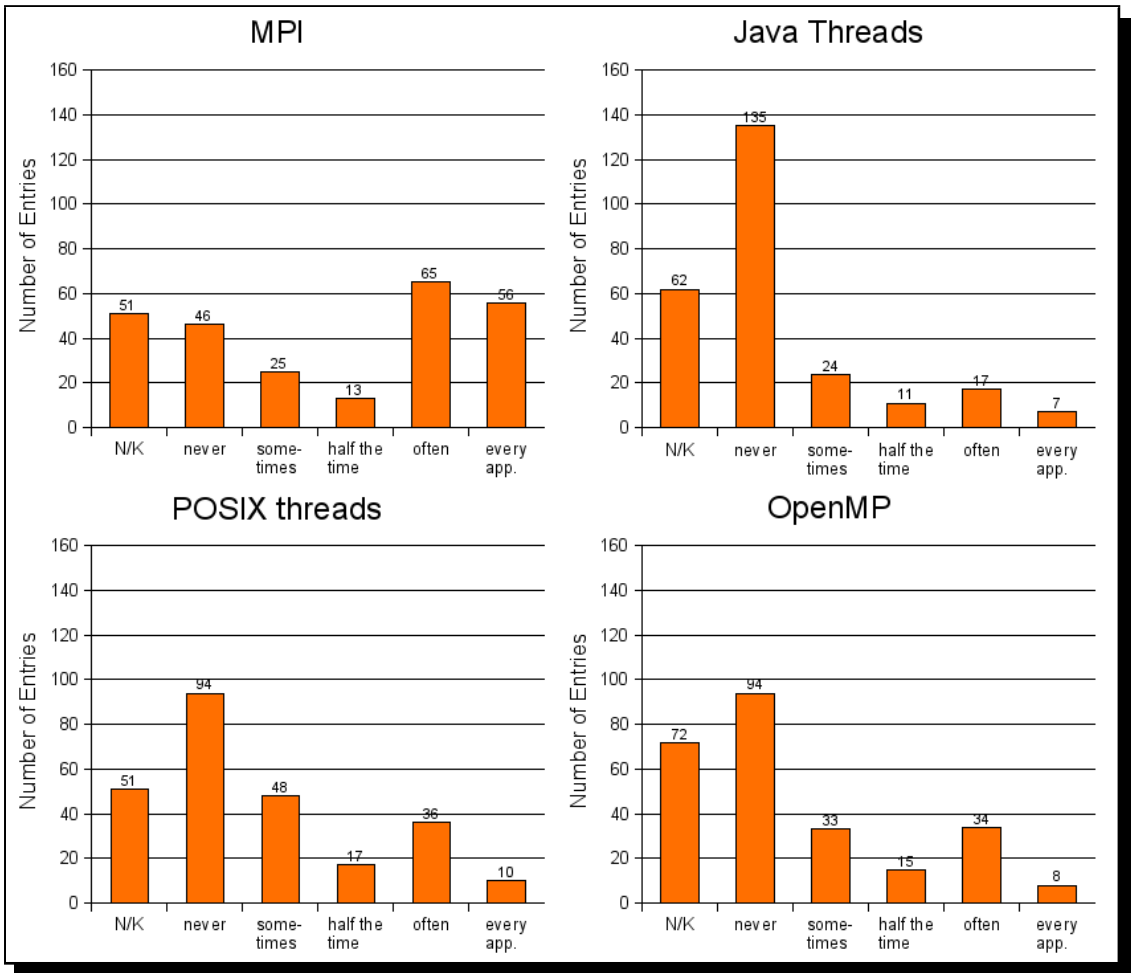


Figure 3.9: Parallel programming systems in detail

Question 7 – Fields

In this question we wanted to know:

- For which scientific or economic fields are you developing parallel applications?

We did not define these fields beforehand, but rather evaluated what people have entered in a text-field. This of course led to a wide variety of fields. We therefore classified them into bigger categories and came up with the following list: Physics (41 submissions), Computer Science (31 submissions), Biology (20 submissions), Chemistry (19 submissions), Mathematics (17 submissions), Engineering (14 submissions) and Astronomy (10 submissions). All other fields had less than ten submissions and were therefore left out. Once again, our survey results are obviously very biased in favor of the systems used for either engineering or the natural sciences.

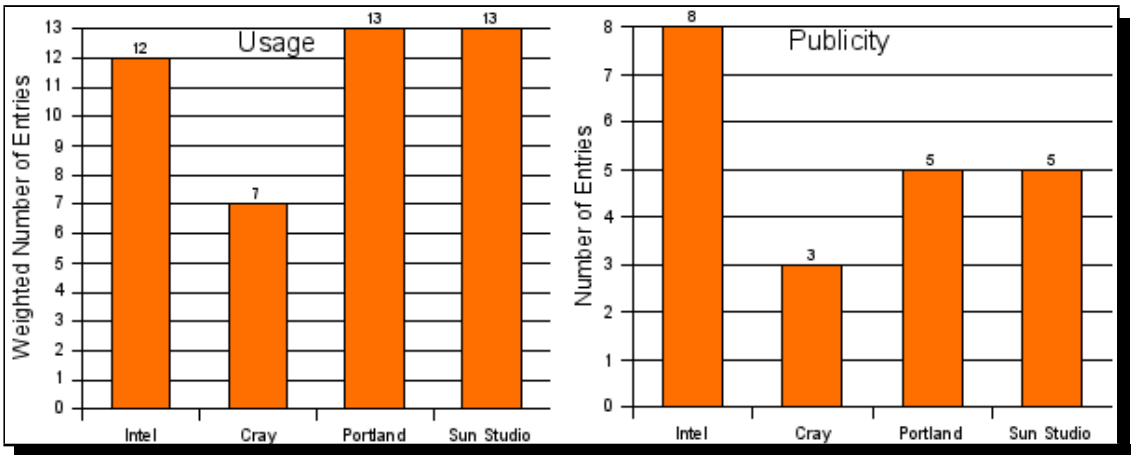


Figure 3.10: Parallelizing compilers

Question 8 – Problems

In another question, participants were asked to enter their problems regarding parallel programming in a text field. The question was formulated as:

- What major problems do you see with the currently available parallel programming systems?

For the analysis, we classified the entered problems into categories, of which six were entered more than ten times: parallelization overhead (41 submissions), need for better debugging tools (36 submissions), need for bug-free compilers or libraries (22 submissions), lack of support by a community/vendor or lack of documentation (15 submissions), need for higher-level development tools (11 submissions), and problems with special hardware (10 submissions). Most of these problems have been described already in Section 2.2 in the foundations of this work, some are also addressed later in this thesis.

It turns out, that the biggest problem with the currently available parallel programming systems was parallelization overhead. Some participants went on to mention that writing parallel programs was very hard in the comments section of the survey.

This closes our description of the parallel programming survey. In the next section, we are going to describe a different aspect of our work to research the present state of the art with regards to parallel programming. It shows frequently made mistakes with OpenMP. Although the section concentrates on OpenMP, many of the mistakes and solutions described there can be easily adapted to other parallel programming systems.

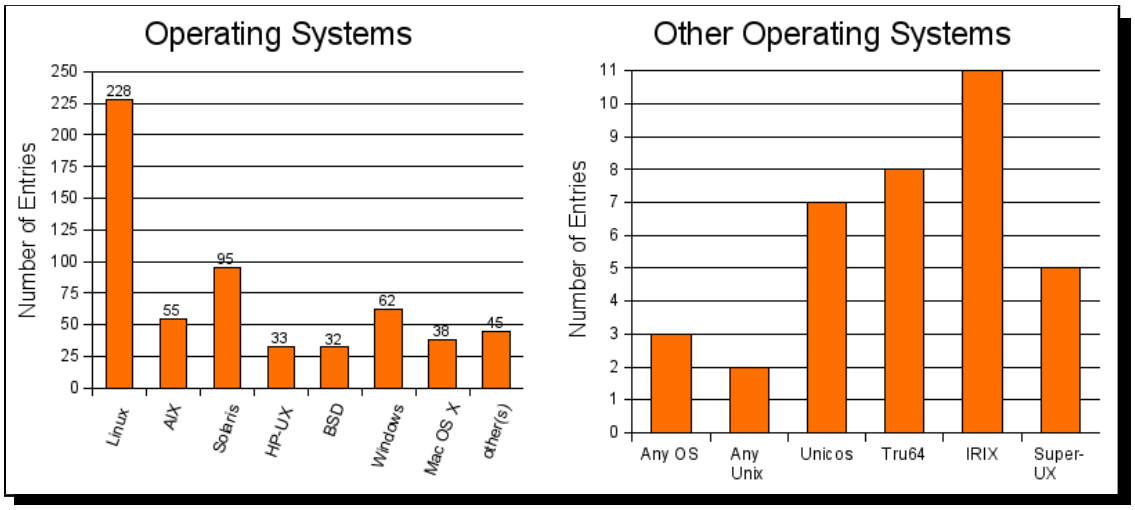


Figure 3.11: Operating systems

3.2 A Study on Frequently Made Mistakes in OpenMP

People who make no mistakes do not usually make anything. (Unknown)

This section is derived from a conference paper published in 2006 [SL06a]. One of the main design goals of OpenMP was to make parallel programming easier. Yet, there are still fallacies and pitfalls to be observed when novice programmers are using the system. We have therefore conducted a study on a total of 85 students visiting our lecture on parallel programming, and observed the mistakes they made when asked to prepare assignments in OpenMP. We hope to save the novice programmer some time and grief with this list, as it is possible to learn from it. Later in this thesis, we will also try to provide solutions that make the mistakes less likely to occur. The study is described in detail in Section 3.2.1.

We are concentrating on the most common mistakes from our study for the rest of this section. They are briefly introduced in Table 3.1, along with a count of how many teams (consisting of two students each) have made the mistake each year. We have chosen to divide the programming mistakes into two categories:

1. *Correctness Mistakes:* all errors impacting the correctness of the program.
2. *Performance Mistakes:* all errors impacting the speed of the program. These lead to slower programs, but do not produce incorrect results.

Section 3.2.2 explains the mistakes in more detail. Also in this section, we propose possible ways and best practices for novice programmers to avoid these errors in the future. Section 3.2.3 reports on tests that we conducted on a variety of OpenMP compilers to figure out, if any of the programming mistakes are spotted and/or possibly corrected by any of the available compilers. Section 3.2.4 reviews related work.

No.	*Problem*	*04*	*05*	Σ
	Correctness Mistakes			
1.	Access to shared variables not protected	8	10	18
2.	Use of locks without `flush`	7	11	18
3.	Read of shared variable without `flush` and synchronization	5	10	15
4.	Forget to mark `private` variables as such	6	5	11
5.	Use of `ordered` clause without `ordered` construct	2	2	4
6.	Declare loop variable in `for` construct as `shared`	1	2	3
7.	Forget to put `for` in `#pragma omp parallel for`	2	0	2
8.	Try to change num. of thr. in parallel reg. after start of reg.	0	2	2
9.	`omp_unset_lock` called from non-owner thread	2	0	2
10.	Attempt to change loop var. while in `#pragma omp for`	0	2	2
	Performance Mistakes			
11.	Use of `critical` when `atomic` would be sufficient	8	1	9
12.	Put too much work inside `critical` region	2	4	6
13.	Use of orphaned construct outside parallel region	2	2	4
14.	Use of unnecessary `flush`	3	1	4
15.	Use of unnecessary `critical`	2	0	2
	Total Number of Groups	26	17	43

Table 3.1: The list of frequently made mistakes when programming in OpenMP

3.2.1 Survey Methodology

We have evaluated two courses for this study. Both consisted of students on an undergraduate level. The first course took place in the winter term of 2004/2005, while the second one took place in the winter term of 2005/2006. The first course had 51 participants (26 groups of mostly two students), the second one had 33 participants (17 groups). The lecture consisted of an introduction to parallel computing and parallel algorithms in general, followed by a short introduction of about five hours on OpenMP. Afterwards, the students were asked to prepare programming assignments in teams of two people, which had to be defended before us. During these sessions (and afterwards in preparation for this paper), we analyzed the assignments for mistakes, and the ones having to do with OpenMP are presented here.

The assignments consisted of small to medium-sized programs, among them:

- find the first N prime numbers
- simulate the dining philosophers problem using multiple threads

- count the number of connected components in a graph
- write test cases for OpenMP directives/clauses/functions

A total of 231 student programs in C or C++ using OpenMP were taken into account and tested on a variety of compilers (e. g. from SUN, Intel, Portland Group, and IBM). Before we begin to evaluate the results, we want to add a word of warning: Of course, the programming errors presented here have a direct connection to the way we taught the lecture. Topics we talked about in detail will have led to fewer mistakes, while for other topics, the students had to rely on the specification or other literature. Moreover, mistakes that have been corrected by the students before submitting their solution are not taken into account here. For these reasons, the numbers presented in Table 3.1 are mere indications of programming errors that novice programmer may make.

3.2.2 Common Mistakes in OpenMP and Best Practices to Avoid Them

In this section, we will discuss the most frequently made mistakes observed during our study, as well as suggest possible solutions to make them occur less likely. There is one universal remark for instructors that we want to discuss beforehand: We based our lecture on assignments and personal feedback, and found this approach to be quite effective: As soon as we pointed out a mistake in the students programs during the exam, a group would rarely repeat it again. Only showing example programs in the lecture and pointing out possible problems did not have the same effect.

There are some mistakes, where we cannot think of any best practices to avoid the error. Therefore, we will just shortly sketch these at this point, while all other mistakes are discussed in their own section below (the number before the mistake is the same as in Table 3.1):

2. *Use of locks without flush*: Before version 2.5 of the OpenMP specification, lock operations did not include a `flush`. The compilers used by our students were not OpenMP 2.5 compliant, and therefore we had to mark a missing `flush` directive as a correctness error.

5. *Use of ordered clause without ordered construct*: The mistake here is to put an `ordered` clause into a `for` work-sharing construct, without specifying with a separate `ordered` clause inside the enclosed `for` loop, what is supposed to be carried out in order.

8. *Try to change number of threads in parallel region after start of region*: The number of threads carrying out a parallel region can only be changed before the start of the region. It is therefore a mistake to attempt to change this number from inside the region.

10. *Attempt to change loop variable while in* `#pragma omp for`: It is explicitly forbidden in the specification to change the loop variable from inside the loop.

11. *Use of critical when atomic would be sufficient*: There are special cases when synchronization can be achieved with a simple `atomic` construct. Not using it in this case leads to potentially slower programs and is therefore a performance mistake.

14. *Use of unnecessary flush*: `flush` directives are implicitly included in certain positions of the code by the compiler. Explicitly specifying a `flush` immediately before or after these positions is considered a performance mistake.

15. *Use of unnecessary critical*: The mistake here is to protect memory accesses with a `critical` construct, although they need no protection (e.g. on private variables or on other occasions, where only one thread is guaranteed to access the location).

Access to Shared Variables Not Protected

The most frequently made and most severe mistake during our study was to not avoid concurrent access to the same memory location. As described in Section 2.4.3, OpenMP provides several constructs for protecting critical regions, such as the `critical` directive, the `atomic` directive and locks. Although all three of these constructs were introduced during the lecture, many groups did not use them at all, or forgot to use them on occasions. When asked about it, most of them could explain what a critical region was for and how to use the constructs, yet to spot these regions in the code appears to be difficult for novice parallel programmers.

A way to make novice programmers aware of the issue is to use the available tools to diagnose OpenMP programs. For example, both the Intel Thread Checker and the Assure tool find concurrent accesses to a memory location.

Read of Shared Variable without Flush and Synchronization

The OpenMP memory model is complicated. Whole sections in the OpenMP specification have been dedicated to it, whole papers written about it [HdS05] and this thesis also includes a short introduction to it (see Section 2.4.8). One of its complications is the error described here. Simply put, when reading a shared variable without flushing it first (in this order), it has an unspecified value. Actually, the problem is even more complicated, as not only the reading thread has to flush the variable, but also any thread writing to it beforehand. The order of these operations is important here as well and this order can only be guaranteed by using other forms of synchronization (e. g. `critical` or `barrier`). Many students did not realize this and just read shared variables without any further consideration. On many common architectures this will not be a problem, because the memory model of these architectures offers more guarantees than the more relaxed one

of OpenMP. Quite often, the problem does not surface in real-world programs, because there are implicit flushes contained in many OpenMP constructs, as well.

In other cases, students simply avoided the problem by declaring shared variables as `volatile`, which puts an implicit `flush` before every read and after every write of any such variable. Of course, it also disables many compiler optimizations for this variable and therefore is often the inferior solution.

The proper solution, of course, is to make every OpenMP programmer aware of this problem, by clearly stating that every read to a shared variable must be protected by a critical region, except in very rare edge-cases not discussed here.

Version 2.5 of the OpenMP specification includes a new paragraph on the memory model. Whether or not this is enough to make novice programmers aware of this pitfall remains to be seen.

Forget to Mark Private Variables as such

This programming error has come up surprisingly often in our study. It was simply forgotten to declare certain variables as private, although they were used in this way. The default sharing attribute rules will make the variable shared in this case.

Our first advice to C and C++ programmers to avoid this error in the future is to use the scoping rules of the language itself. C and C++ both allow variables to be declared inside a parallel region. These variables will be private (except in rare edge cases described in the specification, e. g. static variables), and it is therefore not necessary to explicitly mark them as such, avoiding the mistake altogether.

Our second advice to novice programmers is to use the `default(none)` clause. It will force each variable to be explicitly declared in a data-sharing attribute clause, or else the compiler will complain. We will not go as far as to suggest to make this the default behavior, because it certainly saves the experienced programmer some time to not have to put down each and every shared variable in a shared clause. But on the other hand, it would certainly help novice programmers who probably do not even know about the `default` clause.

It might also help if the OpenMP compilers provided a switch for showing the data-sharing attributes for each variable at the beginning of the parallel region. This would enable programmers to check if all their variables are marked as intended. An external tool for checking OpenMP programs would be sufficient for this purpose as well. Another solution to the problem is the use of autoscoping as proposed by Lin et al. [LCTaM04]. According to this proposal, all data-sharing attributes are determined automatically, and therefore the compiler would correctly privatize the variables in question. The proposed functionality is available in the Sun Compiler 9 and newer.

Last but not least, the already mentioned tools can detect concurrent accesses to a shared variable. Since the wrongly declared variables fall into this category, these tools should throw a warning and alert the programmer that something is wrong.

Declare Loop Variable in `for`-Construct as `shared`

This mistake shows a clear misunderstanding of the way the `for` work-sharing construct works. The OpenMP specification states clearly that these variables are implicitly converted to private, and all the compilers we tested this on performed the conversion. The surprising fact here is that many compilers did the conversion silently, ignoring the shared declaration and not even throwing a warning. More warning messages from the compilers would certainly help here.

Forget to Put for in #pragma omp parallel for

The mistake here is, to attempt to use the combined work-sharing construct `#pragma omp parallel for`, but forget to put down the `for` in there. This will lead to every thread executing the whole loop, and not only parts of it as intended by the programmer.

In most cases, this mistake will lead to the first mistake described in this section, and therefore can be detected and avoided by using the tools specified there.

One way to avoid the mistake altogether is to specify the desired `schedule` clause, when using the `for` work-sharing construct. This is a good idea for portability anyways, as the default `schedule` clause is implementation defined. It will also lead to the compiler detecting the mistake we have outlined here, as `#pragma omp parallel schedule(static)` is not allowed by the specification and yields compiler errors.

omp_unset_lock Called from Non-Owner Thread

The OpenMP-specification clearly states:

> *The thread which sets the lock is then said to own the lock. A thread which owns a lock may unset that lock, returning it to the unlocked state. A thread may not set or unset a lock which is owned by another thread.*
>
> ([Ope05, p. 102])

Some of our students still made the mistake to try to unset a lock from a non-owner thread. This will even work on most of the compilers we tested, but might lead to unspecified behavior in the future.

To avoid this mistake, we have proposed to our students to use locks only when absolutely necessary. There are cases when they are needed (for example to lock parts of a variable-sized array), but most of the times, the `critical` construct provided by OpenMP will be sufficient and easier to use. The use of guard objects as described in Section 5.1.2 would also make this mistake impossible to happen, as they are unset automatically.

Put too much Work inside Critical Region

This programming error is probably due to the lack of sensitivity for the cost of a critical region found in many novice programmers. The issue can be split into two subissues:

1. Put more code inside a critical region than necessary, thereby potentially blocking other threads longer than needed.

2. Go through the critical region more often than necessary, thereby paying the maintenance costs associated with such a region more often than needed.

The solution to the first case is obvious: The programmer needs to check if each and every line of code that is inside a critical region really needs to be there. Complicated function calls, for example, have no business being in there most of the time, and should be calculated beforehand if possible.

As an example for the second case, consider the following piece of code, which some of our students used to find the maximum value in an array:

```
1  #pragma omp parallel for
2  for (i = 0; i < N; ++i) {
3    #pragma omp critical
4    {
5      if (arr[i] > max) max = arr[i];
6    }
7  }
```

The critical region is clearly in the critical path in this version, and the cost for it therefore has to be paid `N` times. Now consider this slightly improved version:

```
1  #pragma omp parallel for
2  for (i = 0; i < N; ++i) {
3    #pragma omp flush (max)
4    if (arr[i] > max) {
5      #pragma omp critical
6      {
7        if (arr[i] > max) max = arr[i];
8      }
9    }
10 }
```

This is called *double-checked locking* and will be faster (at least on architectures, where the `flush`-operation is significantly faster than a critical region), because the critical region is entered less often. Unfortunately, the solution is also incorrect in OpenMP, because of constraints in the memory model (see Section 5.2.2 for an explanation). Finally, consider this version:

```
1  #pragma omp parallel
2  {
3    int priv_max;
4    #pragma omp for
5    for (i = 0; i < N; ++i) {
6      if (arr[i] > priv_max)  priv_max = arr[i];
7    }
8    #pragma omp critical
9    {
10     if (priv_max > max) max = priv_max;
11   }
12 }
```

This is essentially a reimplementation of a reduction using the `max` operator. We have to resort to reimplementing this reduction from scratch here, because reductions using the `max` operator are only defined in the Fortran version of OpenMP (which in itself is a fact that many of our students reported to have caused confusion). Nevertheless, it is possible to write programs this way, and by showing novice programmers techniques like the ones sketched above, they get more aware of performance issues.

Another way to avoid the reimplementation of a reduction described above is to use the `reduce` function provided by the AthenaMP library, see Section 5.3.1.

Use of Orphaned Construct Outside Parallel Region:

There are essentially two subcases to this mistake:

1. Attempting to use a combined parallel work-sharing construct and forgetting to actually put down the `parallel` (writing e. g. `#pragma omp for` outside of a parallel region)

2. Using a synchronization or other construct outside of a region

While there is not much we can suggest for case two (except for the programmer to be aware that this might happen), there is something that can be done in the first case: It has already been suggested to use `default(none)` on every `parallel` construct for novice programmers. When taking this advice to heart, the mistake will be spotted by the compiler, as e. g.

`#pragma omp for default (none)`

is not allowed in the specification and will be detected by the compiler as a syntax error.

3.2.3 Compilers and Tools

There are a multitude of different compilers for OpenMP available, and we wanted to know, if any of them were able to detect the programming errors sketched in Section 3.2.2.

No.	*File*	*icc*	*pgcc*	*sun*	*guide*	*xlc*	*assure*	*itc*
	Correctness Mistakes							
1.	access_shared	-	-	-	-	-	eE	eE
2.	locks_flush	-	-	-	-	-	-	-
3.	read_shared_var	-	-	-	-	-	-	(eE)
4.	forget_private (=No. 1)	-	-	-	-	-	eE	eE
5.	ordered_wo_ordered	-	-	-	-	-	-	eW
6.	shared_loop_var	cE	cC	cW+C	cE	cC	cE	cE
7.	forget_for	-	-	-	-	-	(eE)	(eE)
8.	change_num_threads	-	-	-	-	-	-	-
9.	unset_lock_diff_thread	-	-	-	-	-	-	-
10.	change_loop_var	-	-	-	-	cW	-	-
	Performance Mistakes							
11.	crit_when_atomic	-	-	-	-	-	-	-
12.	too_much_crit							
13.	orphaned_const	-	-	rW	-	-	-	-
14.	unnec_flush	-	-	-	-	-	-	-
15.	unnec_crit	-	-	-	-	-	-	-

Table 3.2: How compilers deal with the problems

Therefore we have written a short testcase for each of the programming mistakes. Table 3.2 describes the results of our tests on different compilers.

The numbers in the first column are the same as in Table 3.1. The second column contains the names of our test programs. We could not think of a sound test for problem 12 (put too much work inside `critical` region), and therefore the results for this problem are omitted. Test program four is the same as test program one, and therefore the results are the same as well. The rest of the table depicts results for the following compilers (this list is not sorted by importance, nor in any way representative, but merely includes all the OpenMP-compilers we had access to at the time):

- Intel Compiler 9.0 (icc)
- Portland Group Compiler 6.0 (pgcc)
- Sun Compiler 5.7 (sun)
- Guide component of the KAP/Pro Toolset C/C++ 4.0 (guide)
- IBM XL C/C++ Enterprise Edition 7.0 (xlc)

- Assure component of the KAP/Pro Toolset C/C++ 4.0 (assure)
- Intel Thread Checker 2.2 (itc)

The last two entries (assure and itc) are not compilers, but tools to help the programmer find mistakes in their OpenMP programs. As far as we know, Assure was superseded by the Intel Thread Checker and is no longer available, nevertheless it is still installed in many computing centers. We were not able to find any lint-like tools to check OpenMP programs in C, there are however solutions for Fortran available commercially.

The alphabetic codes used in the table are to be read as follows: the first (uncapitalized) letter is one of (c)ompiletime, (r)untime or (e)valuation time, and describes, when the mistake was spotted by the compiler. Only Assure and the Intel Thread Checker have an evaluation step after the actual program run. The second (capitalized) letter describes, what kind of reaction was generated by the compiler, and is one of the following: (W)arning, (E)rror or (C)onversion. Conversion in this context means that the mistake was fixed by the compiler without generating a warning. Conversion was done for problem six, where the compilers privatized the shared loop variable. W+C means, that the compiler generated a warning, but also fixed the problem at the same time. There is one last convention to describe in the alphabetic codes: When there are braces around the code, it means that a related problem was found by the program, which could be traced back to the actual mistake. An example: When the programmer forgets to put down `for` in a parallel work-sharing construct (problem seven), it will lead to a data race. This race is detected by the Intel Thread Checker, and therefore the problem becomes obvious. All tests were performed with all warnings turned to the highest level for all compilers.

It is obvious from these numbers that most of the compilers observed are no big help in avoiding the problems described in this paper. Tools such as the Intel Thread Checker are more successful, but still it is most important that programmers avoid the mistakes in the first place. This section and all the work presented in Chapter 4 are a step in this direction.

3.2.4 Related Work

We are not aware of any other studies regarding frequently made mistakes in OpenMP. Of course, in textbooks [CDK00] and presentations teaching OpenMP, some warnings for mistakes are included along with techniques to increase performance, but most of the time, these are more about general pitfalls regarding parallel programming (like e. g. warnings to avoid deadlocks). There is one interesting resource to mention though: the blog of Yuan Lin [Lin05], where he has started to describe frequently made mistakes with OpenMP. Interestingly, at the time of our study, he had not touched any errors that we had described as well, which leads us to think that there are many more potential sources of errors hidden inside the OpenMP specification and the shared memory idiom that it is based on.

This closes our work on frequent mistakes in OpenMP. As already mentioned above, although they are specific to this particular system, at least some of them can happen

in other systems as well. In the next section, we are going to show an attempt to bring knowledge and experiences with regards to parallel programming from different sources into one location to make it more easily accessible: the *Parawiki*.

3.3 The Parawiki

This section is derived from a Technical Report published in 2005 [SL]. It describes an effort to gather knowledge in the field of parallel programming in a single location on the web – the *Parawiki*. The wiki tried to address the following questions: How does one choose one of the available parallel programming systems for a particular project? Where can one go hunt for information about features, reliability and real user experiences (not supplied by the programmers of the system in question)?

Of course, the most valuable resources in a field like this are personal experiences over several projects and years. But many people do not have these, and still need to make educated decisions about the tools for their next project.

Of course, one can go ask on one of the available mailing lists or forums (should one be able to find one that fits), or try one of the available link collections (like DMOZ [dmo]). Search engines might be of help, too, but they require much time to hunt through a lot of hits. Books such as the one by Leopold [Leo01b] may be a nice starting point, but are unfortunately never really up to date, especially when it comes to research projects. For these, one can look into conference proceedings or at the home page of the project, but the information supplied there is rarely unbiased.

All of these methods have one thing in common: They are very time consuming and not likely to lead to reliable and objective information. The solution described here is not new, but has not yet been attempted for this particular field – a web-portal called *Parawiki* [Theb], where information was to be collected about parallel programming systems. Topics included the history of a system, whether or not the system is still actively maintained, features and caveats, as well as links to resources on the web and maybe books describing the system. General information about parallel programming and parallel architectures, for instance definitions of basic terms used in the field were included as well.

So far this sounds much like one of the existing portals about programming languages such as the *Language Guide* [Thea] or one of the available online dictionaries [Fre, Com]. This is true to a certain extent; in fact these websites even served as an inspiration to try something similar for the topic of parallel programming, for which there appeared to be no such site.

A crucial difference between our resource and these sites was that in the Parawiki, the entries were supposed to be written and kept up to date by volunteers from the parallel programming community (might that be experienced students, application programmers or scientists). Unfortunately, this is the first reason why we consider the attempt failed, because the wiki has not attracted enough attention in the community. Merely a few users

registered and a few edits were entered from outside our research group, probably because we did not manage to create enough attention and enthusiasm about the project.

The flexibility of this approach was best served by a wiki, which was the central part of our portal. It was based on the Mediawiki-software, which is also used by the Wikipedia [Wik] and several other large and successfull wikis. In addition to the functionalities that wiki systems generally support, the Parawiki incorporated classifications through a feature called *trees*. Usually, wiki systems have a flat structure, that is, all pages are located at the same level, and connections are provided through hyperlinks. Putting pages in categories is as far as normal wikis usually go. On the other hand, it is often observed that classifications help to organize a complex field. With classifications, it is easier to grasp the overall picture and, especially for a diverse field such as parallel programming, classifications can point to opportunities in the design space that have not been explored so far.

The Parawiki invited the community to invent and improve classification schemes. We had defined a syntax to input tree-like categorizations that provided a quick overview over certain subtopics and additionally served as tables of contents. New systems could be inserted into existing trees, and these could be reorganized according to changing times and tastes by any user. Many categorizations may have lived side by side, as of course ordering by different properties was possible and encouraged. See Figure 3.12 for an example. The trees are a second example of why we consider the effort failed: no trees were created by users outside of our research group and we have never received any feedback on the existing ones.

Up until this point in time, there have been more than 100 substantial pages created in the Parawiki, mostly by our students and us. We have attempted to seed the wiki with useful information about a variety of common systems, architectures and terms with regards to parallel programming. This has lead to some traffic from search engines, but has not enabled us to reach the critical mass where the wiki would receive contributions from people outside of our group. Unfortunately, it has been enough to attract a large amount of spam, which is the third reason why we consider it a failed effort. This has finally forced us to disable registration for new users in the wiki – effectively abandoning the idea altogether.

3.4 Chapter Summary

We have explored three different directions to evaluate the state of the art of parallel programming today: a survey among parallel programmers (*Parasurvey*), a study on frequent mistakes and the creation of a resource to collect knowledge and experiences in the field of parallel programming (the *Parawiki*).

The most important observations of the Parasurvey were:

- C, C++ and Fortran were well-known and widely used in the parallel programming community, the most widely used language in our survey group was C.

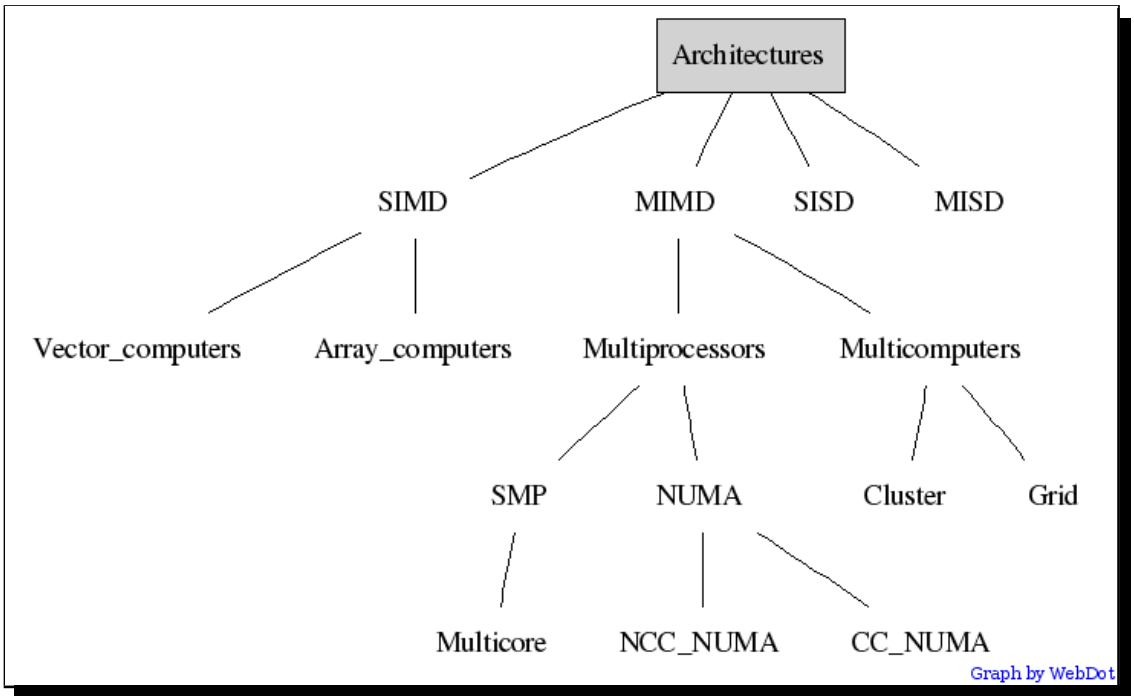


Figure 3.12: A tree visualizing parallel architectures

- Some other languages (e. g. Erlang, Perl and Python) were known and used by at least a few programmers.
- Java was well-known, but not frequently used.
- MPI appeared to be the most popular parallel programming system, followed by POSIX threads, OpenMP and Java threads.
- Algorithmic skeletons, parallelizing compilers and distributed shared memory systems did not seem to be widely accepted.
- The target platform for most parallel applications was a variant of UNIX, predominantly Linux.
- Both distributed memory architectures and shared memory architectures were used in the parallel programming community, grids were not yet common at the time of our survey.
- The major problems with parallel programming for our survey group were parallelization overhead, need for better debugging tools and the need for bug-free compilers or libraries.
- Parallel programming was still felt to be hard by many respondents.

It must be kept in mind (once again) that the data presented was not statistically backed up and therefore only valid for our specific survey group!

In Section 3.2, we have presented a study on frequently made mistakes with OpenMP. Students from our parallel programming courses have been observed for two terms to find out, which were their most frequent sources of errors. We presented 15 mistakes and recommendations for best practices to avoid them in the future. The best practices will be put into a checklist for novice programmers in Section 4.1, along with suggestions from the authors' own experiences. It has also been shown, that the OpenMP-compilers available today were not able to protect the programmer from making these mistakes.

Finally, our attempt to create a shared resource to gather knowledge and experiences in the field of parallel programming has been sketched in Section 3.3, along with reasons why it has failed its purpose.

Chapter 4

Educating Programmers

In this chapter we describe our efforts to educate programmers about parallel programming. The theory behind this effort is that it becomes easier for educated programmers to deal with the difficulties associated with parallel programming – therefore making parallel programming easier.

Two of our efforts have already been put down in previous sections. In Section 3.3 we have described the *Parawiki*, an online resource where information about different PPS can be found. The second attempt is the study about frequently made mistakes in OpenMP in Section 3.2. Ways to avoid the mistakes have been described there as well.

In Section 4.2 we describe an effort to use a weblog called *Thinking Parallel* to educate programmers about the topic of parallel programming. The chapter closes with a short summary in Section 4.3. For a quick overview on how the contents of this chapter fit into this thesis, see Figure 4.1.

4.1 A Checklist for OpenMP Programmers

This section is derived from a conference paper published in 2006 [SL06a], where we have summarized the advice to novice programmers from Section 3.2 and rephrased it into a checklist. For this purpose, we changed the form of address and addressed the novice programmer directly. The checklist also contains other items, which we have accumulated during our own use of and experiences with OpenMP. The list is specific to OpenMP, but some of the items can be adapted to different threading systems.

4.1.1 General

- It is tempting to use fine-grained task decompositions with OpenMP, throwing in an occasional `#pragma omp parallel for` before loops (see Section 2.1.1 for a short description of fine-grained vs. coarse-grained task decomposition). Unfortunately, this rarely leads to big performance gains, because of overhead such as thread creation and scheduling. You therefore have to search for potential for coarse-grained parallelism whenever possible for better results.

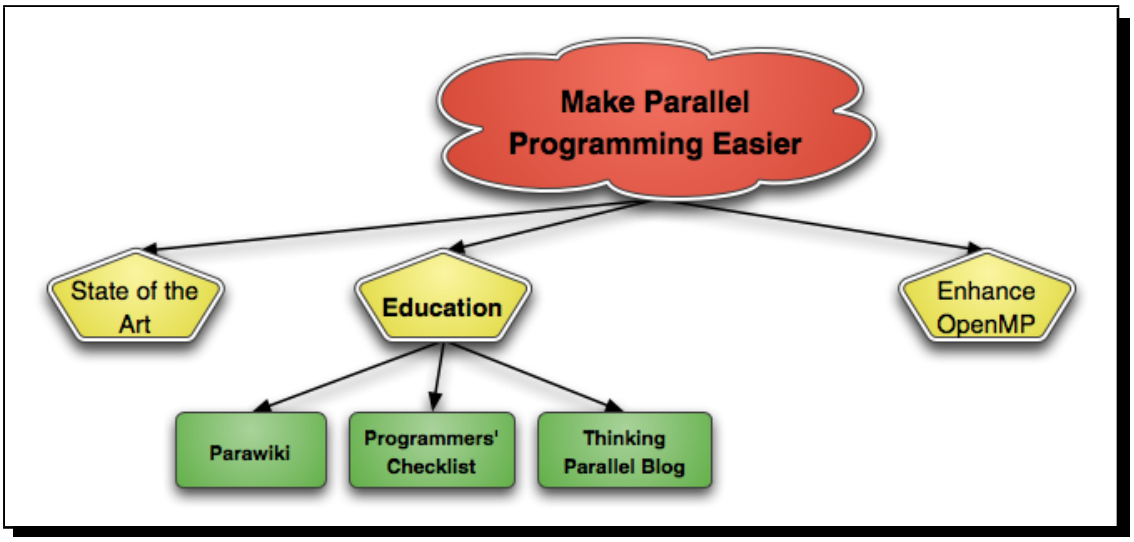


Figure 4.1: Objectives, aims and contributions of this thesis (education)

- Related to the point above, when you have nested loops, try to parallelize only the outer loop. Loop reordering techniques can sometimes help here (see the book by Wolfe [Wol96] for an extensive treatment of these).

- Use reduction where applicable. Even if the operation you need is not predefined, implement it yourself if possible, as shown in Section 3.2.2.

- Use nested parallelism with care, as many compilers still do not support it at the point of this writing, and even if it is supported, nested parallelism may not give you any speed increases in itself.

- Nested parallelism and threadprivate data do not match, as soon as nested parallelism is employed you usually cannot rely on threadprivate data being persistent across parallel regions.

- When doing I/O (either to the screen or to a file), large time savings are possible by writing the information to a buffer first (this can sometimes even be done in parallel), and then pushing it to the device in one run.

- Test your programs with multiple compilers and all warnings turned on, because different compilers will find different mistakes.

- Use tools such as the Intel Thread Checker or Assure, which help you to detect programming errors and write better performing programs.

4.1.2 Parallel Regions

- If you want to specify the number of threads to carry out a parallel region, you must invoke `omp_set_num_threads` before the start of that region (or use other means to specify the number of threads before entering the region).

- If you rely on the number of threads in a parallel region (e. g. for manual work distribution), make sure you actually get this number. This can be done by checking `omp_get_num_threads` after entering the region. Sometimes, the runtime system will give you less threads, even when the dynamic adjustment of threads is off!

- Try to get rid of the `private` clause, and declare private variables at the beginning of the parallel region instead. Among other reasons, this makes your data-sharing attribute clauses more manageable and makes sure, you do not accidentally forget to privatize the variable.

- Use `default(none)`, because it makes you think about your data-sharing attribute clauses for all variables and avoids some errors.

4.1.3 Work-sharing Constructs

- For each loop you parallelize, check whether or not every iteration of the loop has to do the same amount of work. If this is not the case, the static work schedule (which is often the default in compilers) might hurt your performance and you should consider dynamic or guided scheduling.

- Whatever kind of schedule you choose, explicitly specify it in the work-sharing construct, as the default is implementation-defined!

- If you use `ordered`, remember that you always have to use both the `ordered` clause and the `ordered` construct.

- Prefer work-sharing constructs to manual work-sharing whenever possible, as this enables the runtime system to optimize more and also helps the correctness checking tools mentioned above do their work.

4.1.4 Synchronization

- If more than one thread accesses a variable and one of the accesses is a write, you must use synchronization, even if it is just a simple write-operation like $i = 1$. The `flush` directive does not count as synchronization in this context! There are no guarantees by OpenMP on the results otherwise!

- Use `atomic` instead of `critical` if possible, because the compiler might be able to optimize out the `atomic` or substitute it with a powerful machine instruction, while it can rarely do that for `critical`.
- Try to put as little code inside critical regions as possible. Complicated function calls, for example, can often be carried out beforehand.
- Try to avoid the costs associated with repeatedly calling critical regions, for instance by checking for a condition before entering the critical region. Beware of the memory model when doing so, though, as the results you get might not be accurate, as shown in Section 5.2.2.
- Only use locks when necessary and resort to the `critical` directive in all other cases. If you absolutely have to use locks, make sure to invoke `omp_set_lock` and `omp_unset_lock` from the same thread. When you are writing C++ code, consider scoped locking (as described in Section 5.1.2) as an alternative to locks.
- Avoid nesting of critical regions and, if needed, beware of deadlocks. Ways to avoid these are described in detail in Section 5.1.3.
- A critical region is usually the most expensive synchronization construct and takes about twice as much time to carry out as e. g. a barrier on many architectures, therefore start optimizing your programs accordingly. Keep in mind that these numbers only account for the time needed to actually perform the synchronization, and not the time a thread has to wait on a barrier or before a critical region. The latter time depends on various factors, among them the structure of your program and the scheduler.

4.1.5 Memory Model

- Beware of the OpenMP memory model. Even if you only read a shared variable, you have to flush it beforehand, except in very rare edge cases described in the specification. In the most common cases, even a flush is not sufficient, as the memory model states that when multiple threads read and write to the same memory location at the same time without synchronization (`flush` does not count as synchronization in this context), the result for the reading thread is undefined. Except for rare edge-cases, this means you have to use a critical region or locks even to read a shared variable.
- Be sure to remember that locking operations do not imply an implicit flush before OpenMP 2.5.

This closes our checklist. In the next section, we are going to look at an entirely different way to educate programmers: a weblog on parallel programming.

Figure 4.2: Thinking Parallel screenshot

4.2 Thinking Parallel Weblog

Weblogs have become a very common way to interact, communicate and educate on the internet in our days. The top blogs have a daily readership of many hundred thousand, if not one million readers. While most of the personal weblogs serve a small audience and have only a limited educational value, there are others that restrict themselves to a very special topic and have great success in reaching their target audience and educating them about their topic. Examples of such weblogs are *Joel on Software* by Joel Spolsky [Spo00] and *Eric.Weblog()* by Eric Sink [Sin] for the topic of software development, or the blog by Guy Kawasaki [Kaw] on topics related to startups. With that many readers, these blogs really do make a difference and contribute a great deal to the common knowledge about a special topic.

While there are some weblogs on parallel programming out there, most of them are rarely updated [Lin05] or concentrate on one parallel programming system (e. g. Erlang [Sad]). I have therefore setup my own weblog on parallel programming called *Thinking Parallel* [Sue], where I discuss topics related to parallel programming, parallel programming systems (with an emphasis on OpenMP), parallel architectures and high performance computing, as well as news from the blogsphere. A screenshot can be seen in Figure 4.2.

There are two main kinds of articles posted at Thinking Parallel: educational ones about a special topic of interest related to parallel programming, and commentaries on current

events or articles by other bloggers. An example of the first kind is the article titled *A Short Guide to Mastering Thread Safety* [Sue06a]. In this article, the basics of thread safety and how to achieve it are shown. At this point in time, more than 15.000 readers have read this article alone.

A post from the second category is called *Please Don't Rely on Memory Barriers for Synchronization!* [Sue07c]. The article is a follow-up post to a fellow blogger that recommended memory barriers for synchronizing threads. As this is a dangerous practice and hard to get right (especially for beginners), I felt the need to warn people about using the techniques described there.

Lately, I have also added interviews with interesting people in the field, e. g. Joe Armstong, David Butenhof, Sanjiv Shah or Joe Duffy to name just a few. There have been more than 150 articles posted and 350 comments made. By providing all this content, I have managed to attract a sizeable readership to the weblog. Each month, more than 9000 visitors visit the blog on average. On top of that, I have more than 1200 regular subscribers via RSS or email. Although the weblog can not be considered research in the traditional sense, I am convinced that it has helped my goal to educate programmers about parallel programming and concurrency considerably, most likely even more than all other efforts from this thesis combined.

4.3 Chapter Summary

In this chapter, we have described our efforts to educate programmers about the field of parallel programming.There was some overlap of this chapter with the contents of Chapter 3, where we have described the Parawiki (Section 3.3) and some common mistakes made by novice programmers (Section 3.2). Both of these topics are not only useful to evaluate the field, but also to educate programmers.

In this chapter, we built on that knowledge by giving a checklist for OpenMP programmers in Section 4.1. Our second attempt to educate programmers is the Thinking Parallel Weblog described in Section 4.2, where our articles about parallel programming are frequently read by thousands of programmers.

Chapter 5

A Library Approach to Enhancing the Power of OpenMP

In this chapter, we describe our work on making OpenMP more powerful using a library approach. For a quick overview on how the contents of this chapter fit into this thesis, see Figure 5.1.

Powerful libraries are one of the most important building blocks of a successful parallel programming system. This could be observed e. g. in the field of numerics for a long time, where premium quality libraries that encapsulate parallelism are available. For the more general field of programming, design patterns are considered a good way to enable programmers to cope with the difficulties of parallel programming [MSM04].

Although there are a variety of libraries available both commercially and in a research stage, what is missing from the picture is a pattern library in OpenMP. This claim is backed up by our parallel programming survey in Section 3.1, where one of the most prominent problems mentioned is the lack of good parallel libraries. Design patterns implemented in OpenMP should be a viable learning aid to beginners in the field of concurrent programming. At the same time, they are able to encapsulate parallelism and hide it from the programmer if required. For this reason, the AthenaMP project [Sue07a] was created that implements various parallel programming patterns in C++ and OpenMP. This chapter is about AthenaMP, its implementation, and the functionality it provides. A quick overview is provided in Figure 5.2. In that figure, the green components are parts of AthenaMP that we consider finished and ready for release, while the blue ones need more work and are therefore described only at the end of this chapter.

The main goal of AthenaMP is to provide implementations for a set of concurrent patterns of different varieties. These patterns demonstrate solutions to parallel programming problems, as a reference for programmers, and additionally can be used directly as generic components. Because of the open-source nature of the project, it is even possible to tailor the functions in AthenaMP to the need of the programmer. The code is also useful for compiler vendors testing their OpenMP implementations against more involved C++-code, an area where many compilers today still have difficulties. A more extensive project description is provided in my weblog [Sue06b].

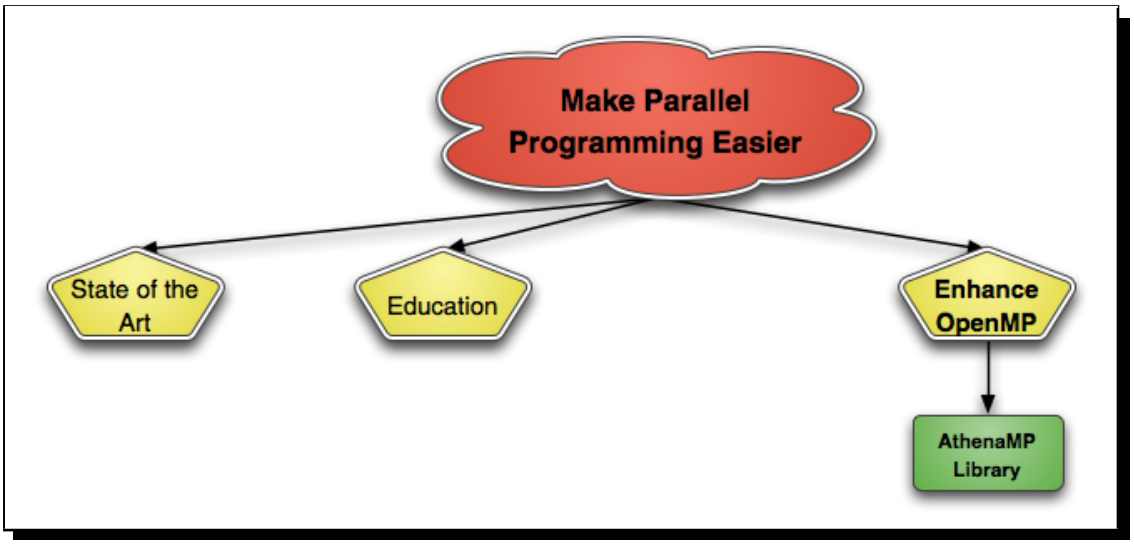


Figure 5.1: Objectives, aims and contributions of this thesis (AthenaMP library)

In Section 5.1, we start by introducing a useful abstraction called *lock adapters*. It wraps the common lock types provided by many parallel programming systems including OpenMP (which are very C-ish) into classes, enables exception safety and prevents programmers from forgetting to destroy locks. It is also the basis for many of the synchronization patterns described in the rest of this section. They attempt to enhance the simple synchronization primitives provided by many parallel programming systems with additional functionality, such as to avoid deadlocks or to provide exception safety transparently to the programmer.

Next, Section 5.2 describes multiple ways to make a very common object-oriented pattern called *Singleton* thread-safe. This pattern is well-known throughout the literature and has been implemented countless times. It guarantees a single point of access to a class and makes sure that there is exactly one instance of the class available throughout the whole program. Yet, to make the pattern safe to use in the presence of multiple threads is another matter altogether, therefore we show ways to do so using OpenMP.

In Section 5.3, we describe implementations of data-parallel patterns: `modify_each`, `transmute`, `combine`, `reduce`, `filter` and `prefix`. The results of two benchmarks showing minor or no performance losses when compared to a pure OpenMP implementation are also described.

Section 5.4 is about parallelizing irregular algorithms with OpenMP. These algorithms have properties such as the need for dynamic task mapping (as explained in Section 2.1.2) that make the introduction of a new datastructure necessary – the *task pool*. This datastructure is also known as *thread pool*, and its correct use is a first example of a generic component/pattern in this chapter. Several versions of task pools were implemented and

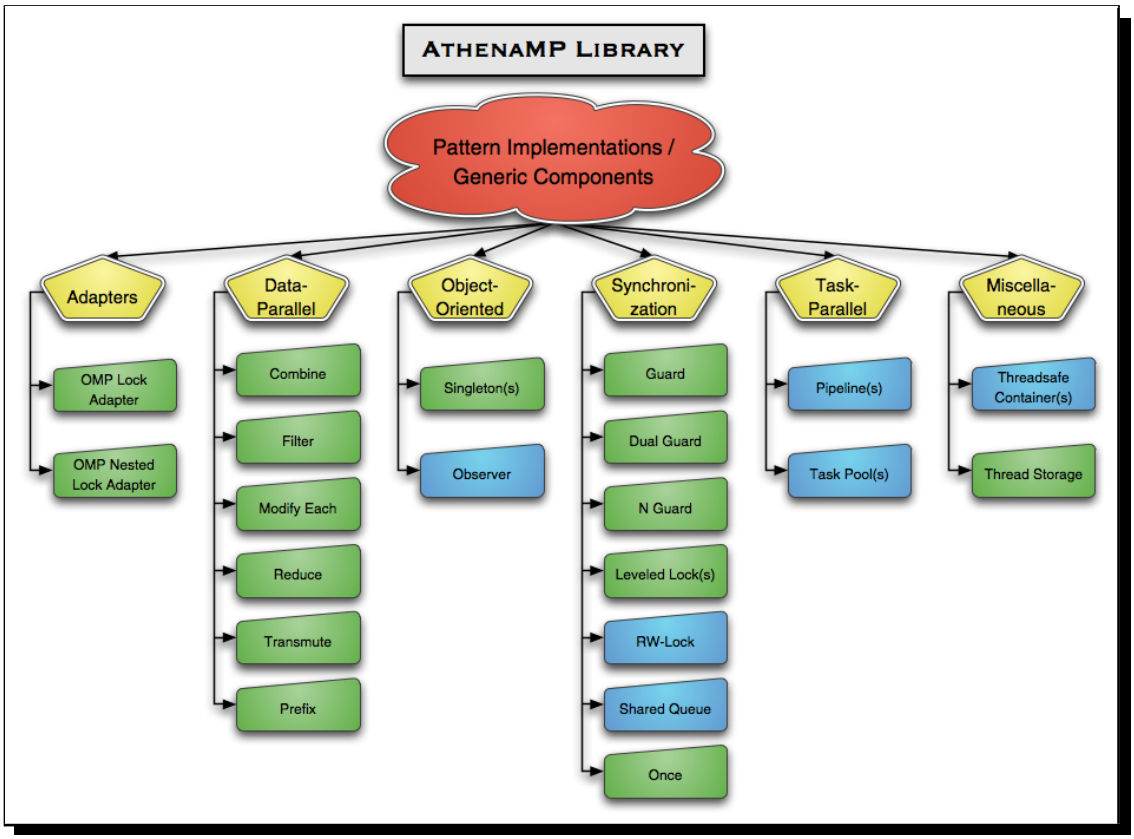


Figure 5.2: The AthenaMP library in a nutshell

are compared. The work described in this section and for the rest of this chapter is not yet integrated into AthenaMP, but work is presently under way to change this.

Section 5.5 sketches more parallel patterns/components from various fields. A summary of our work on AthenaMP closes the chapter in Section 5.6.

5.1 Synchronization Patterns

Locks are one of the most important building blocks of concurrent programming today. As with the advent of multi-core processors, parallel programming starts to move into the mainstream, the problems associated with locks become more visible, especially with higher-level languages like C++. This section address the following ones:

- lock initialization and destruction is not exception-safe and very C-ish. Lock destruction is forgotten frequently.

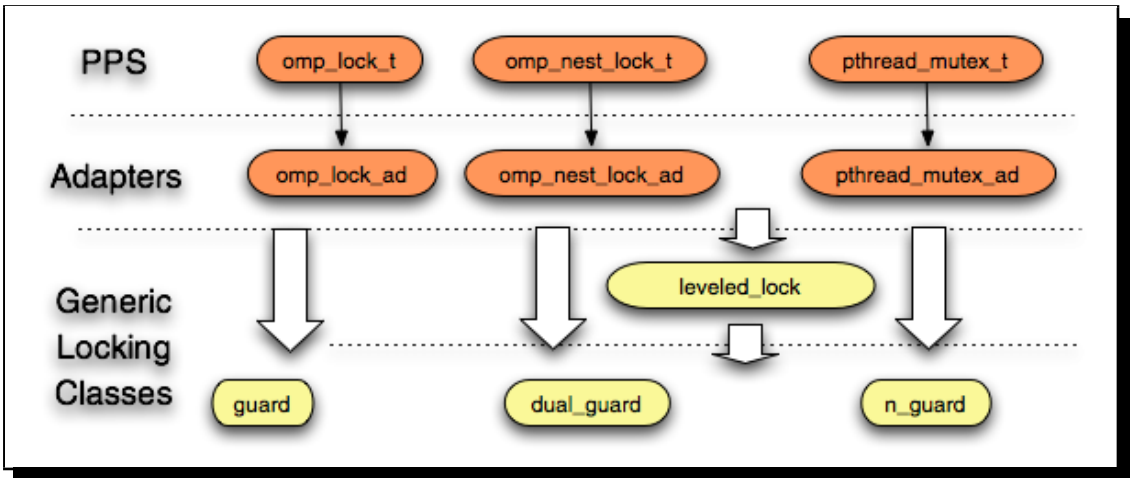


Figure 5.3: Outline of the functionality described in Section 5.1

- setting and releasing locks is not exception-safe and C-ish, as well. Unsetting Locks may be forgotten in complicated code-paths with a lot of branches or may be requested incorrectly by a non-owner thread (see Section 3.2)

- deadlocks are possible when using multiple locks at the same time

To solve the first two problems, a common C++ idiom called *RAII* is used. RAII stands for *Resource Acquisition is Initialization* [Str97] and combines acquisition/initialization and release/destruction of resources with construction and destruction of variables. Our solution to the first problem is called *Lock Adapter* and has the effect that locks are initialized and destroyed in constructors and destructors of variables, respectively. Our solution for the second problem is already well known as *Guard Objects* or *Scoped Locking* and means that locks are set and unset in constructors and destructors of local objects, respectively.

The third problem is solved in two ways: first by extending the guard objects to multiple locks and internally choosing the locking order in a deterministic way, and second by introducing so-called leveled locks that enable the creation and automatic control of a lock hierarchy that detects possible deadlocks at runtime.

The term *generic* warrants some more explanations at this point. From all the functionality introduced in this section, only the adapters are tied to a particular lock type as provided by the parallel programming system. New adapters are trivial to implement for different threading systems, and we have done so as a proof-of-concept for POSIX threads. As soon as these adapters exist, higher-level functionality built on top of the adapters can be used. A slight deviation from this rule are the leveled locks – they have a configurable backend to actually store the level-information, and one of these backends uses thread-local storage as is available in OpenMP. To make up for this, an otherwise

```
class omp_lock_ad {
public:
  omp_lock_ad();
  void set();
  void unset();
  int test();
  omp_lock_t& get_lock();
  ~omp_lock_ad();
};
```

Figure 5.4: Lock Adapter interface

```
class account {
  omp_lock_ad lock;
  double balance;
};

void bank_transfer (account& send,
  account& recv, double amount)
{
  send.lock.set();
  recv.lock.set();

  send.balance -= amount;
  recv.balance += amount;

  recv.lock.unset();
  send.lock.unset();
}
```

Figure 5.4: A bank transfer with Lock Adapters

identical backend has been implemented that is as generic as the rest of the components. All locking abstractions provided are shown in Figure 5.3.

This section is structured as follows: we start by introducing the lock adapters in more depth in Section 5.1.1. Afterwards, we first describe so-called *Guard Objects* to solve the problem of exception safety when setting/unsetting locks (Section 5.1.2). Next, we focus on deadlocks and how to avoid them (Section 5.1.3). This part of our work is closed with performance numbers in Section 5.1.4 and related work in Section 5.1.5.

5.1.1 Lock Adapters in Depth

A lock adapter is a simple wrapper-object for a lock as found in most threading systems. In AthenaMP, lock adapters are provided for the OpenMP types `omp_lock_t` (which is called `omp_lock_ad`) and `omp_nest_lock_t` (`omp_nest_lock_ad`). The interface for an `omp_lock_ad` is shown in Figure 5.4, along with a simple example of how to use it in Figure 5.4. The interface should be self-explanatory, possibly except for the `get_lock` method. It is useful, if library routines that expect the native lock type need to be called, as explained by Meyers [Mey05]. In the example, a simple bank transfer function is sketched (stripped to the bare essentials). The locks are encapsulated in an `account`-class in this case, each account has its own lock to provide for maximum concurrency.

A nice side-effect of the lock adapters is that they turn the traditional C-style locks into more C++-style types. An example: it is a common mistake in OpenMP to forget to initialize a lock before using it (using `omp_init_lock`), or to forget to destroy it

```
template <class LockType>
class guard {
public:
  guard (LockType& lock);
  void acquire();
  void release();
  LockType& get_lock();
  ~guard();
};
```

Figure 5.5: Guard interface

```
void bank_transfer (account& send,
  account& recv, double amount)
{
  guard<omp_lock_ad>
    send_guard(send.lock),
    recv_guard(recv.lock);

  send.balance -= amount;
  recv.balance += amount;
}
```

Figure 5.5: A bank transfer with guard objects

(`omp_destroy_lock`) after it is needed. The *RAII* idiom is employed to avoid the mistake: the locks are initialized in the constructor of the lock adapter and destroyed in the destructor. This ensures that locks are always properly initialized and destroyed without intervention from the programmer, even in the presence of exceptions. Should an exception be thrown and the thread leaves the area where the lock is defined, the lock adapter will go out of scope. As soon as this happens, its destructor is called automatically by the C++ runtime library, properly destroying the lock in the process. Thus, our lock adapters are exception-safe.

5.1.2 Scoped Locking with Guard Objects

The first generic lock type that builds on the lock adapters introduced in the last section is called `guard`. It employs the *RAII*-idiom once again. This special case is so common that it has its own name: *Scoped Locking* [SSRB00]. A local, private guard object is passed a lock as a parameter in its constructor. The guard object sets the lock there and releases it when it goes out of scope (in its destructor). This way, it is impossible to forget to unset the lock (another common mistake when dealing with locks), even in the presence of multiple exit points or exceptions (as described in Section 5.1.1). During normal usage, it also becomes less likely that programmers will unset the lock from a different, non-owner thread, which is another frequently made mistake already described in Section 3.2.

It is also possible to unset the lock directly (using `release`) and to acquire again later (using `acquire`), or to extract the lock out of the guard object (using `get_lock`). The interface of the guard objects is presented in Fig 5.5, along with a short example (a bank transfer again) in Figure 5.5.

A guard object as implemented in AthenaMP can be instantiated with any locking class that provides the basic locking methods (i.e. `set`, `unset`, `test`). It does not depend on OpenMP in any way. In the next section, we build on the abstractions introduced here and in the previous section to turn to a serious problem while using locks: deadlocks and how to avoid them.

```
1  template <class LockType, class LockLevelStoragePolicy>
2  class leveled_lock {
3  public:
4    explicit leveled_lock (const int _lock_level = 0,
5      const int _max_threads = omp_get_max_threads());
6    void set_lock_level (const int _lock_level);
7    int lock_level() const;
8    void set (const int _thread_num = omp_get_thread_num());
9    void unset (const int _thread_num = omp_get_thread_num());
10   int test (const int _thread_num = omp_get_thread_num());
11   LockType& get_lock() const;
12   ~leveled_lock();
13 };
```

Figure 5.6: Leveled Lock interface

5.1.3 Deadlock Detection and Prevention

Deadlocks are an important and all-too common problem in multi-threaded code today. Consider the example code sketched in Figure 5.4 again that shows how a bank transfer can be implemented. A deadlock might occur in this code, as soon as one thread performs a transfer from one account (let's call it account no. 1) to a different account (account no. 2), while another thread does a transfer the other way round (from account no. 2 to account no. 1) at the same time. Both threads will lock the sender's lock first and stall waiting for the receiver's lock, which will never become available because the other thread already owns it. More subtle deadlocks might occur, as soon as more accounts are involved, creating circular dependencies.

We now describe a way to detect deadlocks semi-automatically, along with a corresponding implementation. Next, we show how to avoid deadlocks for an important subproblem.

Deadlock Detection using Leveled Locks

A common idiom to prevent deadlocks are lock hierarchies. If you always lock your resources in a predefined, absolute order, no deadlocks are possible. For our example, this means e.g. always locking account no. 2 before account no. 1. It can sometimes be hard to define an absolute order, though, a possible solution for part of this problem is presented in the next subsection.

Once a lock hierarchy for a project is defined, it may be documented in the project guidelines and developers are expected to obey it. Of course, there are no guarantees they will actually do so or even read the guidelines, therefore a more automated solution may be in order.

```
1  void bank_transfer (account& send, account& recv, double amount)
2  {
3    try {
4      guard<leveled_lock<omp_lock_ad, simple_level_storage> >
5        send_guard(send.lock), recv_guard(recv.lock);
6
7      send.balance -= amount;
8      recv.balance += amount;
9
10   } catch(const lock_level_error& ex) {
11     std::cerr<<ex.what()<<std::endl;
12     return EXIT_FAILURE;
13   }
14 }
```

Figure 5.7: A bank transfer with Guard objects and Leveled Locks

This solution is provided with our second generic lock type: the `leveled_lock`. It encapsulates a lock adapter and adds a `_lock_level` to it, which is passed into the constructor of the class, associating a level with each lock. If a thread already holds a lock, it can only set locks with a lower (or the same - to allow for nested locks) level than the ones it already acquired. If it tries to set a lock with a higher level, a runtime exception of type `lock_level_error` is thrown, alerting the programmer (or quality assurance) that the lock hierarchy has been violated and deadlocks are possible. The interface of the leveled lock is shown in Figure 5.6. Our bank transfer example with leveled locks can be seen in Figure 5.7.

Like guard objects, a leveled lock as implemented in AthenaMP can be instantiated with any locking class that provides the basic locking methods (i.e. `set`, `unset`, `test`). Since the leveled locks provide these methods as well, guard objects can also be instantiated with them, thereby combining their advantages.

Our leveled locks have a configurable backend (via a template parameter) that actually stores the locks presently held for each thread. The first version we implemented depended on OpenMP, since their implementation uses threadprivate memory internally. This has also been a major implementation problem, because we found no OpenMP-compiler that could handle static, threadprivate containers – although judging from the OpenMP-specification this should be possible.

To fix this, a more generic backend was implemented. This version does not use threadprivate memory and does not depend on OpenMP. Instead it uses another generic component implemented in AthenaMP called `thread_storage` (described in more detail in Section 5.5). The component stores all data in an `std::vector` that is indexed by a user-supplied thread-id. For OpenMP, this can be the number returned by

`omp_get_thread_num`, for all other threading systems its the users responsibility to supply this id.

Both versions of locks with level checking induce a performance penalty. Furthermore, the use of throwing runtime exceptions in production code is limited. For this reason, a third backend for the leveled lock was implemented. It has the same interface as the other backends, but does no level checking. Of course, the basic locking functionality is provided. This class enables the programmer to switch between the expensive, checked version and the cheap, unchecked version with a simple `typedef`-command. This version of the leveled lock is called `dummy_leveled_lock` for the rest of this work. The backend is selected at compile time using the second template parameter of the leveled lock called `LockLevelStoragePolicy`.

Deadlock Prevention using Dual-Guards/n-Guards

It has been shown in the last paragraphs that lock hierarchies are a very powerful measure to counter deadlocks. An argument that has been used against them in the past is that you may not be able to assign a unique number to all resources throughout the program. This is easy in our example of bank transfers, because every account most likely has a number and therefore this number can be used. It becomes more difficult when data from multiple sources (e.g. vectors, tables or even databases) need to share a single hierarchy.

While the problem in general is difficult to solve, there is a solution for an important subclass: in our bank transfer example, two locks are needed at the same time for a short period. Even if there was no account number to order our locks into a hierarchy, there is another choice: although not every resource may have a unique number associated with it, every lock in our application does, since the lock's address remains constant during its lifetime.

One possible way to use this knowledge is to tell the user of our library to set their locks according to this implied hierarchy. But there is a better way: As has been described in Section 5.1.2, guard objects provide a convenient way to utilize exception-safe locking. Merging the idea presented above with guard objects results in a new generic locking type: the `dual_guard`. Its interface is sketched in Figure 5.8 and our bank transfer example is adapted to it in Figure 5.8.

No deadlocks are possible with this implementation, because the dual-guard will check the locks' addresses internally and make sure they are always locked in a predefined order (the lock with the higher address before the lock with the lower address to make them similar to the leveled locks).

It is also obvious from this example, how the generic high-level lock types provided by our library raise the level of abstraction when compared to the more traditional locks offered by the common threading systems: to get the functionality that is provided with the one line declaration of the dual-guard, combined with an `omp_lock_ad`, two calls to initialize locks, two calls to destroy locks, two calls to set the locks and two calls to unset the locks are necessary, resulting in a total amount of eight lines of code. These

```
template <class LockType>
class dual_guard {
public:
  dual_guard(LockType&
    lock1, LockType& lock2);
  void acquire();
  void release();
  LockType& get_lock1();
  LockType& get_lock2();
  ~dual_guard();
};
```

Figure 5.8: Dual-Guard interface

```
void bank_transfer (account& send,
  account& recv, double amount)
{
  dual_guard<omp_lock_ad>
    sr_guard(send.lock, recv.lock);

  send.balance -= amount;
  recv.balance += amount;
}
```

Figure 5.8: A bank transfer with Dual-Guards

eight lines of code are not exception-safe and possibly suffer from deadlocks, where the dual-guards are guaranteed to not have these problems.

As always, a dual-guard object as implemented in AthenaMP can be instantiated with any locking class that provides the basic locking methods (i.e. `set`, `unset`, `test`). It does not depend on OpenMP in any way. This also includes all variants of the leveled locks. When instantiated with a `leveled_lock`, using the lock's address to order them into a hierarchy is unnecessary, as there is a user-provided lock level inherent in these locks. In this case, the lock level is used for comparing two locks by the dual-guards automatically.

Generalizing the ideas presented in this section, we go one step further in AthenaMP and also provide an `n_guard` that takes an arbitrary number of locks. Since variadic functions are not well-supported in C++ and it is generally not recommended to use `varargs`, we have decided to let it take a `std::vector` of locks as argument. Apart from this, the functionality provided by these guard objects is equivalent to the dual-guards described above.

It should be noted again, that the dual-guards and n-guards presented in this section are only able to solve a subset of the general deadlock problem: only when two (or more) locks have to be set at the same point in the program, they can be employed. The solution is not applicable as soon as multiple locks have to be set at different times, possibly even in different methods, because the guards must be able to choose which lock to set first.

5.1.4 Performance

We carried out some really simple benchmarks to evaluate, how much the added features and safety impact the performance of locking. Table 5.1 shows how long it took to carry out a pair of set/unset operations for the respective locks. Table 5.2 shows the same data for the guard objects. All numbers are normalized over the course of 500.000 operations. To make the numbers for the dual-guards and n-guards comparable, we have also normal-

Platform (Threads)	omp_lock_t	critical	omp_lock_ad	dummy_ll	ll
AMD (4)	0.40	0.38	0.56	0.67	1.48
SPARC (8)	1.02	1.47	1.09	1.56	4.96
IBM (8)	0.41	0.63	0.41	0.44	1.23

Table 5.1: Wall-clock times (in ms) for one pair of lock/unlock operations

Platform (Threads)	guard	dual_guard	n_guard (20 locks)
AMD (4)	0.65	0.43	0.26
SPARC (8)	1.51	0.87	0.37
IBM (8)	0.42	0.42	0.41

Table 5.2: Wall-clock times (in ms) for one pair of lock/unlock operations for `guards`

ized those to one pair of operations by dividing by two or n respectively ($n = 20$ for the benchmark shown here).

These tables show that there is a performance penalty associated with the added functionality. The leveled locks (shown as ll in the table) are the worst offenders and therefore the implementation of the dummy leveled lock (dll in the table) does make a lot of sense.

5.1.5 Related Work and Contributions

A different implementation of guard objects in C++ can be found in the Boost.Threads library [Kem01], where they are called `boost::mutex::scoped_lock`. Their approach to the problem is different, though: Boost tries to be portable by providing different mutex implementations for different platforms. Our guard objects and high-level locks work on any platform, where a lock adapter can be implemented. Beyond that, this approach allows us to provide guard objects and advanced locking constructs on top of different lock variants, e.g. mutex variables, spinlocks, nested locks or others.

ZThreads [Cra00] provides portable implementations for different lock types with C++, as well. The library includes guard objects that are instantiable with different lock types, but does not include our more advanced abstractions (e.g. leveled locks or dual-guards). It is a portable threading library, with focus on low level abstractions like condition variables and locks. AthenaMP, on the other hand, builds on OpenMP (which is already portable) and can therefore focus on higher-level components and patterns.

The idea of using lock hierarchies to prevent deadlocks is well-known [Tan01]. The idea to automatically check the hierarchies has been described by Duffy for C# [Duf06]. There is also a dynamic lock order checker called Witness available for the locks in the FreeBSD kernel [Bal02].

As far as we know, none of the functionality described in this section has been implemented with C++ and OpenMP. The idea of using memory addresses of locks to create a

consistent lock hierarchy is also a novel contribution, as is the idea of using dual-guards (and n-guards) to hide the complexity of enforcing the lock hierarchy from the user.

In the next section, we are going to concentrate on an entirely different problem: how to implement the well-known singleton pattern in a thread-safe manner using C++ and OpenMP.

5.2 A Thread-safe Singleton Pattern

The work presented in this section is derived from a workshop publication [SL07d]. One of the most widely known patterns among programmers is the *Singleton*. It has been thoroughly analyzed and implemented. Several thread-safe implementations are described in depth here.

The section is organized as follows: In Section 5.2.1, the singleton pattern is introduced, along with a simple, non-threaded implementation in C++. Section 5.2.2 describes various thread-safe implementations, highlights problems with OpenMP and explains possible workarounds. The performance of our solutions is benchmarked in Section 5.2.3.

5.2.1 The Singleton Pattern

The most famous description of the singleton pattern is from Gamma et al. [GHJV95]:

> *Ensure a class only has one instance, and provide a global point of access to it.* (Gamma et al. [GHJV95])

Singletons are useful to ensure that only one object of a certain type exists throughout the program. Possible applications are a printer spooler or the state of the world in a computer game. While singletons provide a convenient way to access a resource throughout the program, one needs to keep in mind that they are not much more than glorified global variables, and just like them need to be used with care (or not at all, if possible). Yegge explains this in great detail in one of his weblog posts [Yeg04].

While singletons can be implemented in C++ using inheritance, we have decided to implement them using wrappers and templates, as described by Schmidt et al. [SSRB00]. These authors call their classes adapters, but we stick to the more general name *singleton wrapper* instead, to avoid confusion with the well-known adapter pattern (which is different from what we are doing). We provide wrapper classes that can be instantiated with any class to get the singleton functionality. For example, to treat the class `my_class` as a singleton and call the method `my_method` on it, one needs to do the following:

```
singleton_wrapper<my_class>::instance().my_method();
```

Provided that all accesses to the class `my_class` are carried out in this way, then and only then the class is a singleton. The code of class `my_class` does not need to be

changed in any way, but all of the singleton functionality is provided by the wrappers. This is the biggest advantage of this implementation over an inheritance-based approach. Typical requirements for an implementation state that the singleton:

- must not require prior initialization, because forgetting to do so is a frequent mistake by programmers
- must be initialized only when it is needed (lazy initialization)
- must return the same instance of the protected object, regardless of whether it is called from within or outside a parallel region

A non-thread-safe version of the `instance` method of a singleton wrapper is shown in Figure 5.9. Here, `instance_` is a private static member variable, omitted for brevity in all of our examples. This implementation, of course, has problems in a multi-threaded environment, as the `instance_` variable is a shared resource and needs to be protected from concurrent access. Ways to deal with this problem are shown in the next section.

5.2.2 Thread-Safe Singleton Implementation Variants

A safe and simple version of a multi-threaded singleton wrapper is shown in Figure 5.9. It uses a named critical region to protect access to the singleton instance. While this solution solves the general problem of protecting access to the singleton class, it has two major drawbacks, both of which we are going to solve later: First, it uses the same named critical construct to protect all classes. If e. g. a singleton for a printer spooler and a singleton for the world state is needed in the same program, access to these classes goes through the same critical region and therefore these accesses can not happen concurrently. This restriction is addressed next. The second problem is that each access to the singleton has to pay the cost of the critical region, although technically it is safe to work with the singleton after it has been properly initialized and published to all threads. An obvious (but incorrect) attempt to solve this problem is shown here first, others follow later in this section.

The Safe Version using One Lock per Protected Object In the last paragraph, we have shown a thread-safe singleton that used one critical region to protect accesses to all singletons in the program. This introduces unnecessary serialization, as a different critical region per singleton is sufficient. We will show four different ways to solve the problem. The first two require changes in the OpenMP specification to work but are quite simple from a programmer's point of view, the third can be done today but requires a lot of code. The fourth solution requires a helper-class and has problems with some compilers.

```
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  if (instance_ == 0) {
    instance_ = new Type;
  }
  return *instance_;
}
};
```

Figure 5.9: The `instance` method of a sequential singleton wrapper

```
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  #pragma omp critical (ATHENAMP_1)
  {
    if (instance_ == 0) {
      instance_ = new Type;
    }
  }
  return *instance_;
}
};
```

Figure 5.9: The `instance` method of a simple, thread-safe singleton wrapper

Attempt 1: Extending Critical: Our first attempt at solving the problem is shown in Figure 5.10. The idea is to give each critical region a unique name by using the template parameter of the singleton wrapper. Unfortunately, this idea does not work, because the compilers treat the name of the critical region as a string and perform no name substitution on it. While it would be theoretically feasible to change this in compilers, because template instantiation happens at compile-time, it would still require a change in the OpenMP specification, and we suspect the demand for that feature to be small. For this reason, we are not covering this attempt any further in Section 6.3, where we describe proposed changes to the OpenMP specification.

Attempt 2: Static Lock initialized with OMP_LOCK_INIT: Our second attempt to solve the problem uses OpenMP locks. The code employs a static lock to protect access to the shared instance variable. Since each instance of the template function is technically a different function, each instance gets its own lock as well, and therefore each singleton is protected by a different critical region.

The big problem with this approach is to find a way to initialize the lock properly. There is only one way to initialize a lock in OpenMP – by calling `omp_init_lock`. This must only be done once, but OpenMP does not provide a way to carry out a piece of code only once (a solution to this shortcoming is presented in Section 5.5.4). One of our requirements for the singleton is that it must not need initialization beforehand, therefore we are stuck.

A solution to the problem is shown in Figure 5.10 and uses static variable initialization to work around the problem, by initializing the lock with the constant `OMP_LOCK_INIT`. This is adapted from POSIX threads. In that parallel programming system, a mutex can be initialized with `PTHREAD_MUTEX_INITIALIZER`. In a thread-safe environment

```
// this code works, but does
// not solve the problem!
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  #pragma omp critical(Type)
  {
    if (instance_ == 0) {
      instance_ = new Type;
    }
  }
  return *instance_;
}
};
```

Figure 5.10: Attempt 1: `instance` method with multiple critical regions

```
// this code does not work!
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  static omp_lock_t my_lock
    = OMP_LOCK_INIT;
  omp_set_lock (&my_lock);
  if (instance_ == 0) {
    instance_ = new Type;
  }
  omp_unset_lock (&my_lock);
  return *instance_;
}
};
```

Figure 5.10: Attempt 2: `instance` method with multiple critical regions

(which OpenMP guarantees for the base language, see Section 2.4.9), the runtime system should make sure this initialization is carried out only once, and all the compilers we have tested this with actually do so. Of course, `OMP_LOCK_INIT` is not in the OpenMP specification, but we believe it would be a worthy addition to solve this and similar problems, not only related to singletons. We come back to this proposal in Section 6.3. For the reasons stated, although quite an elegant solution, this attempt does not work with OpenMP today as well.

Attempt 3: Doing it Once: It has been shown in the previous paragraph that the function `omp_init_lock` needs to be called only once, but OpenMP provides no facilities to achieve that. It is possible to code this functionality in the program itself, though, as we will describe in Section 5.5.4. Here, we restrict ourselves to showing how the functionality can be used.

Figure 5.11 depicts the `instance` method with once functionality. Line 19 has the actual call to the `once` template function defined in AthenaMP. The function takes two parameters, the first one being a functor that includes what needs to happen only once (also shown at the top of Figure 5.11). The second parameter is a flag of the type `once_flag` that is an implementation detail to be handled by the user.

This attempt to solve our problem works. Nevertheless, it needs a lot of code, especially if you also count the code in the `once` method. Moreover, it relies on static variable initialization being thread-safe, as our last attempt. For these reasons, we are going to solve the problem one last time.

```
1  struct init_func {
2    omp_lock_t *lock;
3    void operator() ()
4    {
5      omp_init_lock (lock);
6    }
7  };
8
9  template <class Type>
10 class singleton_wrapper {
11 static Type& instance ()
12 {
13   static omp_lock_t my_lock;
14   static once_flag flag
15     = ATHENAMP_ONCE_INIT;
16   init_func my_func;
17   my_func.lock = &my_lock;
18
19   once (my_func, flag);
20
21   omp_set_lock (&my_lock);
22   if (instance_ == 0) {
23     instance_ = new Type;
24   }
25   omp_unset_lock (&my_lock);
26   return *instance_;
27 }
28 };
```

Figure 5.11: Attempt 3: `instance` method with multiple critical regions

```
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  static omp_lock_ad my_lock;
  my_lock.set();
  if (instance_ == 0) {
    instance_ = new Type;
  }
  my_lock.unset();
  return *instance_;
}
};
```

Figure 5.11: Attempt 4: `instance` method with multiple critical regions

Attempt 4: Lock Adapters to the Rescue: Figure 5.11 shows our last attempt at protecting each singleton with its own critical region. It is substantially smaller than all previous versions and should also work with OpenMP today. It uses lock adapters as described in Section 5.1.1.

Because lock initialization takes place automatically when an instance of the class `omp_lock_ad` is created, our problem with the initialization having to be carried out only once disappears. Or at least: it should disappear. Unfortunately, some compilers we have tested this with call the constructor of the lock adapter more than once in this setting, although it is a shared object. The next paragraph explains this more thoroughly.

The difference between static variable construction and static variable initialization becomes important here, because static variable initialization works correctly in a multithreaded setting with OpenMP on all compilers we have tested, but static variable con-

struction has problems on some compilers. Static variable initialization basically means declaring a variable of a primitive data type and initializing it at the same time:

```
static int my_int = 5;
```

This makes `my_int` a shared variable and as in the sequential case, it's initialization is carried out once. This works on all compilers we have tested. Static variable construction on the other hand looks like this:

```
static my_class_t my_class;
```

Since `my_class_t` is a user-defined class, it's constructor is called at this point in the program. Since the variable is static, it is a shared object and therefore the constructor should only be called once. This does not happen on all compilers we have tested (although we believe this to be a bug in them, since OpenMP guarantees thread safety of the base language) and is therefore to be used with care. A trivial test program to find out if your compiler correctly supports static variable construction with OpenMP is available from the authors on request.

This solution is the most elegant and also the shortest one to provide each singleton with its own critical region. Nevertheless, our second problem is still there: each access to the singleton has to go through a critical region, even though only the creation of the singleton needs to be protected. We are going to concentrate on this problem in the next few paragraphs.

Double-Checked Locking and Why it Fails

Even in some textbooks, the *double-checked locking* pattern is recommended to solve the problem of having to go through a critical region to access a singleton [SSRB00]. Our `instance` method with this pattern is shown in Figure 5.12.

Unfortunately, the pattern has multiple problems, among them possible instruction reorderings by the compiler and missing memory barriers, as explained by Meyers and Alexandrescu [MA04], as well as problems with the OpenMP memory model, as explained by de Supinski on the OpenMP mailing list [dS06]. Since the pattern may and will fail in most subtle ways, it should not be employed.

Using a Singleton Cache

Meyers and Alexandrescu [MA04] also suggest caching a pointer to the singleton in each thread, to avoid hitting the critical region every time the `instance` method is called. This can of course be done by the user, but we wanted to know if it was possible to extend our singleton wrapper to do this automatically. We therefore came up with the implementation shown in Figure 5.12.

```
// this code is wrong!!
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  #pragma omp flush(instance_)
  if (instance_ == 0) {
    #pragma omp critical(A_2)
    {
      if (instance_ == 0)
        instance_ = new Type;
    }
  }
  return instance_;
}
};
```

Figure 5.12: `instance` method with double-checked locking

```
template <class Type>
class singleton_wrapper {
static Type* cache_;
static Type& instance ()
{
  #pragma omp threadprivate(cache_)
  if (cache_ == 0) {
    cache_ =
      &singleton<Type>::instance();
  }
  return *cache_;
}
};
```

Figure 5.12: `instance` method using caching

The implementation solves the problem, as the critical region for each singleton is only entered once. It relies on declaring a static member variable threadprivate. Unfortunately, the OpenMP specification does not allow to privatize static member variables in this way. In our opinion, this is an important omission not restricted to singletons, and therefore this point is raised again later in this work, when we put up a list of enhancement proposals for the OpenMP specification in Section 6.3.

There is a workaround, however, which was suggested by Meyers in his landmark publication Effective C++ [Mey05] as a solution to a different problem. Instead of making the cache a static member variable (that cannot be declared threadprivate), it can be declared as a static local variable in the `instance` method. Declaring local variables threadprivate is allowed by the specification, and this solves the problem without any further disadvantages.

The whole solution unfortunately has some problems. It relies on threadprivate data declared with the `threadprivate` directive, but in OpenMP these have some restrictions. In a nutshell, these data become undefined as soon as nested parallelism is employed or as soon as the number of threads changes over the course of multiple parallel regions (see the OpenMP specification for details [Ope05]). There is no way to work around these limitations for this solution, therefore the user has to be made aware of them by carefully documenting the restrictions. Fortunately, there is a different way to achieve the desired effect and it is explained next.

```
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  static Type instance;
  return instance;
}
};
```

Figure 5.13: `instance` method using a Meyers singleton

The Meyers Singleton

One of the most well-known singleton implementations today is the so-called *Meyers Singleton* that is described in Effective C++ [Mey05]. It is quite elegant, small, and shown in Figure 5.13.

Meyers himself warns about using his implementation in a multi-threaded setting, because it relies on static variable construction being thread-safe. Of course, Meyers did not write about OpenMP. OpenMP guarantees thread safety of the base language in the specification, and this should cover proper construction of static variables in a multi-threaded setting as well. Unfortunately, as described in Section 5.2.2 in detail, our tests show that some OpenMP-aware compilers still do have problems with this. Some of them were even calling the constructor of the same singleton twice, which should never happen in C++. Therefore, although it is the smallest and most elegant solution to the problem, it cannot be recommended for everyone at this point in time.

5.2.3 Performance

This section discusses the performance of the proposed singleton wrapper implementations. We have setup a very simple test to access the performance of our solution. It is shown in Figure 5.14.

Two different singletons are used in the example, one is an integer and one is a double. The protected singletons will usually be classes, of course, but for our simple performance measurements primitive data-types will do. The singletons are initialized prior to the parallel region. Inside the parallel region, they are read only and their result is added up and tested outside the parallel region for correctness (not shown in the figure). The results of our tests are shown in Table 5.3. The measured numbers are in seconds and show the best of three runs with 10.000.000 singleton accesses per thread. Only the entries printed in bold are correct and safe!

Here is a short summary of the table headings:

- **one_crit**: a singleton wrapper using the same critical region for all protected singletons (see Figure 5.9)

```
int counter = 0;
double fcounter = 0.0;

double start = omp_get_wtime ();

singleton_wrapper<int>::instance() = 1;
singleton_wrapper<double>::instance() = 1.0;
#pragma omp parallel reduction(+:counter) reduction(+:fcounter)
{
  for (int i=0; i<numtries; ++i) {
    counter += singleton_wrapper<int>::instance();
    fcounter += singleton_wrapper<double>::instance();
  }
}

double end = omp_get_wtime ();
```

Figure 5.14: The code used to benchmark our singletons

Test Environment	one_crit	multi_crit	local_cache	meyers	dcl
AMD, Intel Comp., 4 Thr.	**32.8**	**18.6**	**1.38**	**0.04**	1.46
Sun, Sun Comp., 8 Thr.	**176**	**182**	n.a.	0.14	0.62
IBM, IBM Comp., 8 Thr.	**72.7**	**71.7**	**0.34**	0.01	0.85

Table 5.3: Measured singleton benchmark timings in seconds

- **multi_crit**: a singleton wrapper using a different critical region for all protected singletons with a lock adapter (see Figure 5.11)
- **local_cache**: a singleton wrapper that caches a pointer to the singleton in thread-private memory (see Figure 5.12)
- **meyers**: a singleton wrapper built after the Meyers Singleton (see Figure 5.13) - the numbers are only representative on the Intel Compiler, because the other two do not construct static classes correctly
- **dcl**: for comparison, we have also included the double-checked locking version (see Figure 5.12), although it is not safe to use!

As can be clearly seen by these numbers, the Meyers Singleton is to be preferred on all architectures, as it is the fastest by several orders of magnitude. We do not have any numbers for the version using a threadprivate cache on the Sun Compiler, as it was not able to translate our code. We believe this to be a bug in the compiler.

These results leave us with a disappointing situation: we have isolated a best solution (Meyers Singleton), the solution should be legal judging from the OpenMP specification,

yet some compilers do not implement it correctly. While we cannot fix the compilers, what we can do at this point is provide a short test program that shows which compilers behave correctly and which do not. It is available from the authors on request.

5.2.4 Related Work

Lots of work has been put into correctly implementing the singleton pattern. The most famous resource on the topic is the book by Gamma et al. [GHJV95]. The idea for our singleton wrapper is described by Schmidt et al. [SSRB00], along with double-checked locking that was later proved to be an anti-pattern by Meyers and Alexandrescu [MA04]. The idea for the singleton cache is also from the latter source. More involved descriptions of singletons with different properties in C++ are given by Alexandrescu [Ale01]. Yegge describes most clearly why singletons should be used with care [Yeg04]. In the next section we are leaving the field of object-oriented patterns and show our implementations of several data-parallel patterns.

5.3 Data-Parallel Patterns

The work presented in this section is derived from a conference publication [SL07c]. It reports on the implementations of data-parallel patterns: `modify_each`, `transmute`, `combine`, `reduce`, `filter` and `prefix`. The results of two benchmarks showing no or minor performance losses when compared to a pure OpenMP implementation are also included.

Section 5.3.1 starts with a general introduction to the features of our patterns/generic components and goes on to highlight each one of them in detail. The performance of our solutions is described in Section 5.3.2. Section 5.3.3 describes related work.

5.3.1 Implementation and Features

This subsection introduces the data-parallel patterns contained in AthenaMP to date. First, we explain some features of our implementation common to all of them.

The interfaces of our functions have been designed using the Standard Template Library (STL) as an example where applicable. Just like in the STL, all functions presented here are polymorphic through the use of templates. It is possible to mix the library calls freely with normal OpenMP code. This is useful e. g. for employing nested parallelism, where the user specifies the first level of parallelism in user's code and the second level is opened up by the library.

All functions take two or more iterators specifying one or more ranges of elements as parameters, as is customary in the STL. Behind the scenes, depending on the iterators supplied, there are two versions of the patterns: one that takes random-access iterators (as supplied by e. g. `std::vectors` or `std::deques`), and one that takes forward

or bidirectional iterators (as supplied by e. g. `std::lists`). Using template metaprogramming (i. e. iterator traits), the right version is selected transparently to the user by the compiler.

The first version makes use of the `schedule` clause supplied with OpenMP. Although it is set to static scheduling, this is easily changeable in the source of the library. When the functor supplied takes a different amount of time for different input values, using a dynamic schedule is a good idea. This random-access version is fast and has easy to understand internals, because it uses OpenMP's `parallel for` construct.

The second version is for forward iterators. Use of the scheduling functionality supplied by OpenMP would be very costly in that case, as to get to a specific position in the container, $O(n)$ operations are necessary in the worst case. This operation would need to be carried out n times, resulting in a complexity of $O(n^2)$. For this reason, we have implemented our own static scheduling. The internal `schedule_iter` template function is responsible for it. It takes references to two iterators. One of them marks the start and the other the end of the range to be processed. The function calculates the number of elements in the range, divides it among the threads, and changes the iterators supplied to point to the start and the end of the range to be processed by the calling thread (by invoking `std::advance`). This results in a total complexity of $O(n * t)$, where t is the number of threads involved. Iff each thread in a team calls the function and works on the iterators returned, the whole range is processed with only a few explicit calls to `std::advance`. However, the forward iterator version is still slower and less flexible (because changing the scheduling policy is not possible with this version) than the random access iterator version, as can be observed in Section 5.3.2.

The last parameter to each function is the number of threads to use in the computation. If the parameter is left out, it defaults to the number of threads as normally calculated by OpenMP. Unless otherwise specified in the description of the pattern, the original order of the elements in the range is preserved and no critical sections of any kind are required.

The parallelism inherent in the patterns described is totally hidden from the user, except of course when looking into the AthenaMP source code. This does not necessarily lead to smaller sources (as a lot of scaffolding code for the new functors is required), but to less error-prone ones. A lot of mistakes are common when working with OpenMP (some of which are described in Section 3.2) and these can be reduced by deploying the patterns.

It is also possible to nest patterns inside each other, e. g. by calling `reduce` inside a functor supplied to `modify_each` for two-dimensional containers. Nested parallelism is employed in this case, which becomes especially useful if you have a large number of processors to keep busy.

The following patterns and their implementations are discussed in the next few paragraphs:

- `modify_each`: also commonly known as *map*, modifies each element supplied
- `transmute`: also known as *transform* applies a function to each element provided and returns a new range containing the results of the operation

```
/** a user-supplied functor that adds diff to its argument */
class add_func : public std::unary_function<int, void> {
  public:
    add_func (const int diff) : diff_ (diff) {}
    void operator() (int& value) const { value += diff_; }
  private:
    const int diff_;
};

/* add 10 to all targets in place. */
athenamp::modify_each (targets_.begin(), targets_.end(),
    add_func (10));
```

Figure 5.15: `modify_each` in action

- `combine`: combines elements from two sources using a binary function and returns a new range containing the results of the operation
- `reduce`: also known as *fold*, combines all elements into a single result using an associative operation
- `filter`: filters the elements supplied according to a predicate function and returns the results using a new container
- `prefix`: combines elements into results using $x_1 \circ x_2 \circ \ldots \circ x_k$ for $1 \leq k \leq n$ and stores the results using a new container. The most well-known operation for $\circ$ is addition. In this case the computation is called *prefix sum*.

modify_each

The `modify_each` pattern provides a higher-order template function to apply a user-supplied functor to each element in a range in parallel. No provisions are made to protect any internal data in the functor or any side-effects of the functor from concurrent access. Figure 5.15 shows an example, where the pattern is used to add ten to all elements in an `std::vector`. The user is relatively free with regards to the functor supplied, as long as it contains an `operator()` method that is a unary function and takes a non-const reference as argument. If `operator()` has a return value, it is ignored.

transmute

The `transmute` pattern is similar to `modify_each` in the way that it applies a user-supplied unary functor to all elements in a range in parallel. While `modify_each` works on the original elements and modifies them, `transmute` stores its results in a different range and is therefore even able to change the type of each element. It is known in the

```
/* combine all elements from sources1_ with all elements from
 * sources2_ and store the results in sources1_ */
athenamp::combine (sources1_.begin(), sources1_.end(),
    sources2_.begin(), sources1_.begin(), std::plus<int>());
```

Figure 5.16: `combine` in action

STL as `transform` (but similar to many of our components, we could not use this name to avoid name-clashes with the STL-version).

The list of parameters it takes is similar to `modify_each` again, except for the fact that it takes an additional iterator that points to the location where the results are to be stored. The user is responsible for making sure that there is enough room to store all results. The user-supplied functor is similar to the one supplied for `modify_each`, except for the fact that it cannot work on its argument directly, but returns the result of its computation instead. The result is then put into the appropriate location by our library function `transmute`.

No provisions are made to protect any internal data in the functor or any side-effects of the functor from concurrent access. While it is possible and correct to use `transmute` to apply changes inplace by overlapping its supplied ranges, it is recommended to use `modify_each` in that case because it is faster, since it involves less copying of elements.

An example is omitted here for brevity and because of the similarity of this method to the already explained `modify_each` pattern.

combine

`combine` is a relatively simple pattern that is used to combine elements from two different ranges using a binary operation and put the result into a third range. It is similar to the `transmute` pattern explained above, except that it works on two ranges instead of one. Similar restrictions as for `transmute` also apply here: the user is responsible for making sure there is enough room to put the results in and the internals of the functor are also not protected from concurrent access. It is also possible to store the results inplace, as shown in Figure 5.16. What is also shown in this figure is that it is possible to use the functors provided by the STL for this pattern (`std::plus` in this example). If this is the case, many lines of code can be saved when compared to the parallel version without patterns.

reduce

The `reduce` pattern combines all elements in a given range into a single result using a binary, associative operation. It is also commonly known as *fold* or *for_each*. Many parallel programming systems feature a reduce-operation, among them OpenMP. The reduce-

```
/** A functor that can be used to find the maximum and sum of all
 * elements in a range by repeatedly applying operation() to all
 * elements in the range. */
class max_sum_functor : public std::unary_function<int, void> {
  public:
    max_sum_functor (int max, int sum) : max_(max), sum_(sum) {};
    int max () const { return max_; }
    int sum () const { return sum_; }

    void operator() (const int arg1)
    {
        max_ = std::max (arg1, max_);
        sum_ += arg1;
    }

    void combine (const max_sum_functor& func)
    {
        max_ = std::max (func.max (), max_);
        sum_ += func.sum ();
    }

  private:
    int max_;
    int sum_;
};

max_sum_functor func (-1, 0);
/* calculate maximum and sum of all elements in vector targets_ */
athenamp::reduce (targets_.begin(), targets_.end(), func);
/* check result */
std::cout << func.max() << std::endl << func.sum() << std::endl;
```

Figure 5.17: `reduce` in action

operation in OpenMP has a disadvantage, though: it is limited to a few simple, predefined operators, such as + or ∗. These operators can only be applied to a few data-types, such as `int`'s.

Our `reduce` method solves these problems, as a user-defined functor is specified as operator. This functor can work on any data-type. Multiple reductions in a single pass are possible as well, with a functor that does two or more operations at the same time. An example (see Figure 5.17) will make things clearer, before we go into more details. The functor in the example stores the variables `max_` and `sum_` internally. They are initialized appropriately in the constructor and can be read after the operation has completed using the `max` and `sum` methods. `operator()` is applied to each element in the range to find the maximum element and the sum of all elements.

```
/* filter all odd numbers and store them into results_ */
results_ = athenamp::filter<std::list<int> > (targets_.begin(),
    targets_.end(), std::bind2nd (std::modulus<int>(), 2));
```

Figure 5.18: `filter` in action

Internally, the `reduce` method creates a copy of the functor for each thread involved in the calculation. For this reason, the functors supplied must also have a copy constructor. In our example, the compiler takes care of this correctly, therefore we have omitted it. At the end of the reduction, the `combine` method (which has nothing in common with the `combine` pattern mentioned in the last section!) is used to correctly combine the different functors from each thread. As can be seen in the example, there is no need to protect anything from concurrent access, as the `reduce` method does this automatically. As a downside, this also means that our implementation has a critical region that must be carried out once by all threads involved, which of course decreases the performance slightly, especially for quick operations on few elements.

filter

The `filter` pattern filters a range of objects according to a predicate functor and stores all results for which the predicate returns true into a new container. The target container must be specified as template parameter and must offer both `push_back` and `insert` methods in its interface (all STL sequence containers do). A small example to make things clearer is shown in Figure 5.18.

In this example, all odd numbers are filtered out of the `targets_`-vector and stored into the `results_`-list. The predicate functor can either be a standard one from the STL (as shown in the example) or a user-defined one. In all cases, no protection of the internals of the functor from concurrent access is guaranteed. This should not be a problem, as most functors used in this case do not have any internals to protect.

The implementation has no critical sections, but a small sequential part at the end where the contents of each threads accumulated objects are copied together into a new container that is returned afterwards.

prefix

The prefix pattern combines elements into results using $x_1 \circ x_2 \circ \ldots \circ x_k$ for $1 \leq k \leq n$ and stores the results in a different range (or optionally inplace). The most well-known prefix version is the prefix sum (shown as an example in Figure 5.19). Prefix is different from the rest of our patterns, since it is the most complicated of them all and also the most difficult to parallelize. It does not contain a special version for random-access iterators, but should still run faster for them because it uses some STL functions that have fast

```
/* calculate prefix sum and store results in targets_ */
athenamp::prefix (sources_.begin(), sources_.end(),
    targets_.begin(), std::plus<int>());
```

Figure 5.19: `prefix` in action

Platform (Threads)	`modify_each`	OpenMP variant (lists)
AMD (4)	0.1085 (8.0511)	0.0889 (7.5558)
SPARC (8)	2.389 (35.7338)	2.3788 (34.7902)
IBM (8)	0.1457 (3.3544)	0.1415 (3.4071)

Table 5.4: Wall-clock times in seconds for the `modify_each` function.

versions for these iterators (e.g. `std::advance`). Multiple synchronization points are needed in the function and therefore it does not scale as well as the others.

5.3.2 Performance

Measuring performance of generic components such as the ones provided here is hard, as it depends heavily on the user code that is to be parallelized. Most patterns do not need locks or perform at most one locking operation per thread. Therefore, they are able to scale to a large number of processors. If the amount of work to be done in the user-supplied functor is too small, however, bus contention will become an issue and performance may not be satisfactory – but this is the case for pure OpenMP as well.

We have performed two different tests on our components. For the first test, we incorporated the `modify_each` pattern into the game-like application OpenSteerDemo [KL07], which is a testbed for the C++ open-source library OpenSteer [Rey04] written by Reynolds. It simulates and graphically displays the steering behavior of autonomous computer-controlled characters, called agents, in real-time. No difference at all was measurable between the pure OpenMP version and our version using generic components. We expect this to be the case in most applications.

The second test refers to the case that the user-supplied functor is too small. Our benchmark uses e. g. `modify_each` to add one to all elements in an `std::vector` or `std::list`, respectively. Similar, very simple operations are carried out for the other functions. The results are shown in Figures 5.4 to 5.9. The values in braces are measurements for the respective version using forward iterators. All tests carried out on containers with 100.000 elements, using 10.000 repetitions. Of course, this benchmark is not representative in any way, as it is clearly limited by the available memory bandwidth. Yet, since this is true for the pure OpenMP-version as well, it shows that the overhead introduced by using the pattern is marginal even for this case.

Platform (Threads)	`transmute` (lists)	OpenMP variant (lists)
AMD (4)	0.1042 (7.6689)	0.0761 (7.5658)
SPARC (8)	2.4251 (36.4147)	1.2908 (34.2533)
IBM (8)	0.1621 (3.3796)	0.0391 (3.4363)

Table 5.5: Wall-clock times in seconds for the `transmute` function.

Platform (Threads)	`combine` (lists)	OpenMP variant (lists)
AMD (4)	0.0987 (2.6799)	0.0862 (2.6684)
SPARC (8)	2.3426 (9.0076)	2.2568 (8.983)
IBM (8)	0.1803 (2.2424)	0.1576 (2.2531)

Table 5.6: Wall-clock times in seconds for the `combine` function.

It can also clearly be observed that the version of our patterns using forward iterators (as found in lists) is several orders of magnitude slower than the one for random access iterators. The reasons for this behavior have been explained in Sec. 5.3.1. For our other patterns, similar results can be observed.

5.3.3 Related Work

Many of the data-parallel patterns described here are similar to functionality provided by the Standard Template Library (STL). There are a variety of parallel STL implementations in a research stage, among them STAPL [AJR+01] or PSTL [JG97]. Similar functionality is also starting to appear in commercial projects, such as the Intel Threading Building Blocks [Int07] or in QT Concurrent [Tro07]. What differentiates this work from these projects is its strong focus on OpenMP on one hand (which no other project we know of provides), and its ability for programmers to study the source and learn from that. By using the expressiveness of OpenMP, its code is far easier to understand and adapt than any of the other libraries we are aware of.

This closes our work on data-parallel patterns. In the next section, we are going to describe our work on irregular algorithms and task pools.

Platform (Threads)	`reduce` (lists)	OpenMP variant (lists)
AMD (4)	0.078 (1.998)	0.0711 (1.9835)
SPARC (8)	2.2285 (7.1695)	2.2066 (6.806)
IBM (8)	0.1642 (1.8802)	0.1661 (1.6508)

Table 5.7: Wall-clock times in seconds for the `reduce` function.

Platform (Threads)	`filter` (lists)	OpenMP variant (lists)
AMD (4)	6.1821 (6.5745)	4.605 (5.5161)
SPARC (8)	10.5285 (13.4097)	9.2464 (14.0275)
IBM (8)	11.8475 (11.2674)	8.1289 (8.8956)

Table 5.8: Wall-clock times in seconds for the `filter` function.

Platform (Threads)	`prefix` (lists)	OpenMP variant (lists)
AMD (4)	2.9889 (3.8473)	2.7263 (4.1557)
SPARC (8)	4.063 (9.9796)	4.841 (9.8717)
IBM (8)	1.7942 (3.7026)	1.8125 (3.7398)

Table 5.9: Wall-clock times in seconds for the `prefix` function.

5.4 Task Pools

The work presented in this section is includes material from a workshop publication in 2004 [SL04] and a workshop publication in 2006 [WSL06]. Everything that has been described so far in this chapter is part of AthenaMP already. The work described in the next sections will be part of it eventually, but is not yet finished and sufficiently polished up.

OpenMP provides powerful constructs to parallelize regular programs, i.e., programs that execute a similar set of operations on different elements of a regular data structure such as an array. Irregular algorithms, in contrast, are difficult to parallelize using the existing OpenMP constructs. For irregular algorithms, the units of work (tasks) can usually not be mapped statically to a fixed number of threads (as explained in Section 2.1.2), because their number or size depends on the given input. Therefore it is often not possible to predict the amount of work to be done in a task for any particular input data.

According to Mattson [Mat03], one of the initial designers of the OpenMP specification, OpenMP was never meant for irregular applications. Other people have tried to use OpenMP for this kind of applications, though, and have gotten mixed results [HASP00, DVT00, NPA01].

One approach to achieve dynamic mapping is the use of task pools. A task pool is a data structure that stores tasks to support mapping them to a certain number of threads. Section 5.4.1 gives an overview about some task pool variants that we have implemented in OpenMP. In Section 5.4.2, we introduce three example applications that are used in Section 5.4.3 to assess the performance of the different task pool variants: quicksort, labyrinth-search and sparse cholesky factorization. Performance numbers gathered with the workqueuing model proposed by Shah et al. [SHPT99] are included for comparison in this section as well. Section 5.4.4 surveys related work and closes this part of our work on irregular algorithms.

```
1  task_data_t *task_data;
2  tpool_t *pool = tpool_init(num_threads, sizeof(task_data_t));
3  task_data = generate_initial_task();
4  tpool_put(pool, 0, task_data);
5  #pragma omp parallel shared(pool)
6  {
7    task_data_t *my_task_data;
8    int me = omp_get_thread_num();
9    while(TPOOL_EMPTY != tpool_get(pool, me, &my_task_data);
10     do_work(my_task_data); /* includes calls to tpool_put (...) */
11 }
12 tpool_destroy(pool);
```

Figure 5.20: OpenMP program using task pools

5.4.1 Overview

Task pools are used to achieve dynamic load balancing (mapping) in irregular applications. A task pool stores tasks that are either created dynamically at runtime, or where the size is not known in advance. It also provides a set of operations that allow threads to insert and extract tasks concurrently in a thread-safe manner. In the next few paragraphs, the high-level interface for the programmer used by all our task pool variants is introduced. Afterwards the different task pool variants are described.

Application Programming Interface

All implemented task pools use the same application programming interface. This API provides functions to initialize and destroy the task pool structure, as well as to insert and extract tasks concurrently. Figure 5.20 shows an example of the relevant part of an OpenMP program that uses our API. At the moment, this is still very C-ish. In ongoing work, this is changed to an object-oriented design using templates to turn it into a generic component useful for AthenaMP.

First, a task pool must be initialized by using the `tpool_init` function. This function must only be called once, and only by a single thread. Afterwards, the task pool can be used to store (`tpool_put`) and extract (`tpool_get`) tasks. The latter function blocks until it either successfully extracts a task from the pool, or discovers that the task pool is empty and all threads using the pool are idle. Finally, function `tpool_destroy` frees the memory used by the task pool.

Variants of Task Pools

We implemented several variants of task pools. Some of them (*sq1*, *sdq1* and *dq8*) were ported to OpenMP from existing POSIX threads and Java implementations described by

Korch and Rauber [KR04]. Others (*dq9* and *dq9-1*) have been developed by the authors as enhancements of the *dq8* variant. The remainder of this section explains the different versions in detail.

Central Task Queue: The simplest way to design a task pool, called *sq1*, is to use a single shared task queue. Each thread is allowed to access this queue with functions `tpool_put` and `tpool_get`. We used OpenMP lock variables to ensure that only one thread can access the task queue at a time. The variant has the drawback that when two or more threads are trying to access the task pool simultaneously, they have to wait for each other. Therefore, the task pool can become a bottleneck for applications that use a large number of threads or access the pool frequently. Nevertheless, this variant offers good load balancing capabilities and performs well for applications that create only few tasks or access the task pool rarely.

Combined Central and Distributed Task Queues: To reduce waiting times caused by access conflicts, the task pool variant *sdq1* uses distributed task queues. It manages a private task queue for each thread and permits only the owner thread to access the queue. Therefore no synchronization operations are needed for the private queues. An additional central queue is maintained for load balancing. Whenever a private queue is empty, the owner thread tries to fetch a task from the central queue. To ensure the exchange of tasks among the threads, the size of the private queues is limited. If a thread tries to enqueue a new task and discovers that its private queue is full, it will move the new task to the central queue.

Distributed Queues with Dynamic Task Stealing: In contrast to *sdq1*, the task pool variants *dq8*, *dq9* and *dq9-1* use multiple shared queues to reduce the possibility of access conflicts: each thread has its own private and its own shared queue. If a thread runs out of tasks in its private queue, it will take a task from its shared queue. If the shared queue is also empty, the thread will try to steal a task from the shared queue of another thread, and then return it from `tpool_get`.

Although the task pool variants *dq8*, *dq9* and *dq9-1* are conceptually similar, they use different strategies for filling the shared queues. Like *sdq1*, *dq8* uses private queues with a limited size. If a private task queue is full, the new tasks are moved to the shared queue.

Unlike *dq8*, variants *dq9* and *dq9-1* adjust the size of the private queues dynamically, based on the state of the shared queue. The size of a private queue in *dq9* and *dq9-1* is not limited to a certain value. The private queues of these task pool variants can contain an arbitrary number of tasks. The reason is that *dq9* and *dq9-1* try to keep most tasks in the private queues to reduce the number of operations on shared queues. A thread will move a task into its shared queue in `tpool_put` only if the shared queue is running empty. If both the private and the shared queue are empty, a new task will be inserted into the shared queue in *dq9*, but into the private queue in *dq9-1*.

Name	Num. Q.	Num. Shar. Q.	Size of Priv. Q.	Task-stealing Time
sq1	1	1	-	-
sdq1	*num_threads* + 1	1	limited (2)	-
dq8	*num_threads* ∗ 2	*num_threads*	limited (2)	priv. queue empty
dq9	*num_threads* ∗ 2	*num_threads*	unlimited	priv. queue empty
dq9-1	*num_threads* ∗ 2	*num_threads*	unlimited	priv. queue low (2)

Table 5.10: Comparison of implemented task pool variants

Another difference between *dq9-1* and *dq9* is the point in time, when task stealing is started. While *dq8* and *dq9* do not attempt to steal tasks before a private queue is empty, *dq9-1* initiates task stealing as soon as the number of tasks in the private queue drops below a predefined threshold value. This is done to prevent the private queue from running empty.

All of these different variants are summarized in Tab. 5.10. In the table, Q. stands for Queue, the number in braces stands for the actual number of tasks used in our tests.

5.4.2 Benchmarks

To compare the performance of our task pool variants, we have implemented three irregular applications: quicksort, labyrinth-search and sparse cholesky factorization. Quicksort is a popular sorting algorithm, initially invented and described by Hoare [Hoa62]. The labyrinth-search application finds the shortest path through a labyrinth using the breadth-first search algorithm. To ensure that all labyrinth cells with the same distance from the entry cell are visited before any other cells are processed, we use two task pools. The tasks in the first pool correspond to cells with distance d from the entry cell. The second task pool is used to collect tasks (cells) with distance $d+1$. The algorithm is explained in more detail in Section 6.2.1.

Cholesky factorization is an algorithm to solve systems of linear equations $Ax = b$. Information on cholesky factorization can be found, for instance, in the book by George and Liu [GL81]. To test our task pool variants, we have implemented only the most expensive part of cholesky factorization: numerical factorization.

5.4.3 Performance

Performance measurements were carried out on an AMD Opteron 848 class computer with four processors at 2.2 GHz, and on a Sun Fire E6900 with 24 dual-core Ultra Sparc IV processors at 1.2 GHz. On the AMD system, a maximum of four threads was used, while on the Sun system, a maximum of eight threads was used. Although more threads would have been possible on the latter machine, eight processors is the maximum number that this machine supports without encountering NUMA-effects (as it consists of multiple mainboards with 4 dual-core processors each). On the AMD system, the benchmarks were

compiled with the Intel C++ Compiler 9.0 using options `-O2` and `-openmp`. On the Sun Fire E6900, the Guide compiler with options `-fast --backend -xchip=ultra3cu --backend -xcache=64/32/4:8192/512/2 --backend -xarch=v8plusb` was used. We have not used the native SUN compiler, because it does not support the workqueuing extension (see next paragraph).

For comparison, we implemented quicksort and the cholesky factorization using Intel's proposed workqueuing model. It was first introduced by Shah et al. [SHPT99], as an integrated approach to achieve dynamic load balancing for irregular applications. Those authors suggest an OpenMP extension which allows to split the work into units (tasks) that are distributed dynamically to the threads of a program using a task queue. Since both the Intel C++ and the Guide compilers already support the workqueueing model, we implemented two benchmarks using this proposed OpenMP extension on the same set of machines. Unfortunately, we could not implement the labyrinth-search algorithm with this model, because we did not find a way to use two different queues and ensure that all tasks from one queue are executed before the program starts to execute tasks from the second queue.

Figure 5.21 shows the wall-clock times in seconds for the quicksort benchmark application with different task pool variants and the Intel taskq implementation. We have used an array with 100.000.000 elements as input data on the AMD Opteron system and an array with 10.000.000 elements on the Sun Fire E6900. The results for the cholesky factorization are shown in Figure 5.22. For the cholesky factorization, a 500x500 matrix was used as input. Figure 5.23 shows the results for the labyrinth-search benchmark application with different task pool variants.

Our experiments indicate that the performance of different task pool variants depends on the type of application. Quicksort and labyrinth-search, which create a large number of tasks, achieve better performance using task pools with distributed task queues. Cholesky factorization, in contrast, generates only a few tasks, and therefore good load balancing is crucial. The use of private queues turns out to be a drawback in this case, because all tasks remain in the private queues and the idle threads have no chance to fetch them. The performance of *dq9* is good, though, because this variant makes the distribution of tasks among the queues dependent on the number of tasks in the pool. If there are only a few tasks in the pool (shared queues are empty and at least one thread is idle), a new task will be inserted into a shared queue (and not, like e. g. for *dq9-1* into a private queue). If there are enough tasks in the pool, however, *dq9* will insert a new task into a private queue to avoid synchronization operations. Using this technique, *dq9* achieves much better performance than the other task pool variants with distributed queues.

Figure 5.22 shows that the only task pool variant that uses one central queue (*sq1*) achieves the best performance for cholesky factorization. The reason is the good load balancing offered by *sq1*: all tasks are kept in one central queue, where all threads can access them. Due to the small number of tasks generated by the algorithm, a central queue does not slow down the program because the application accesses the task pool only rarely.

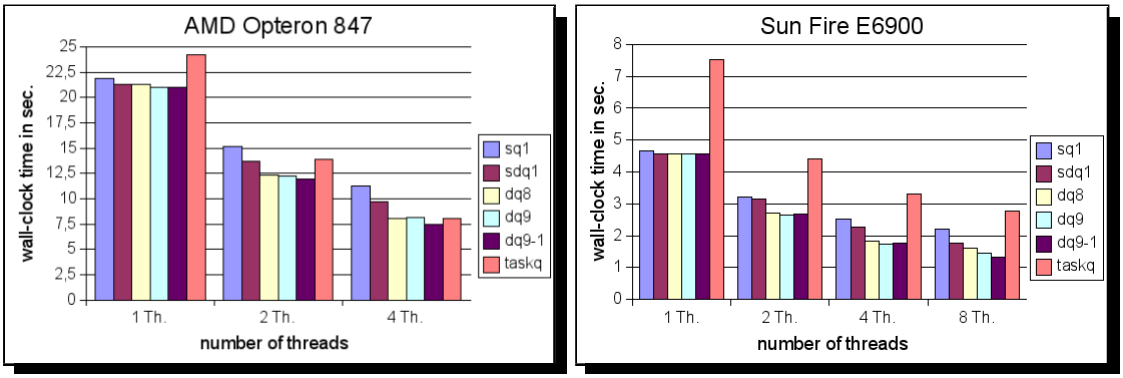


Figure 5.21: Wall-clock times for quicksort in seconds (best of three runs)

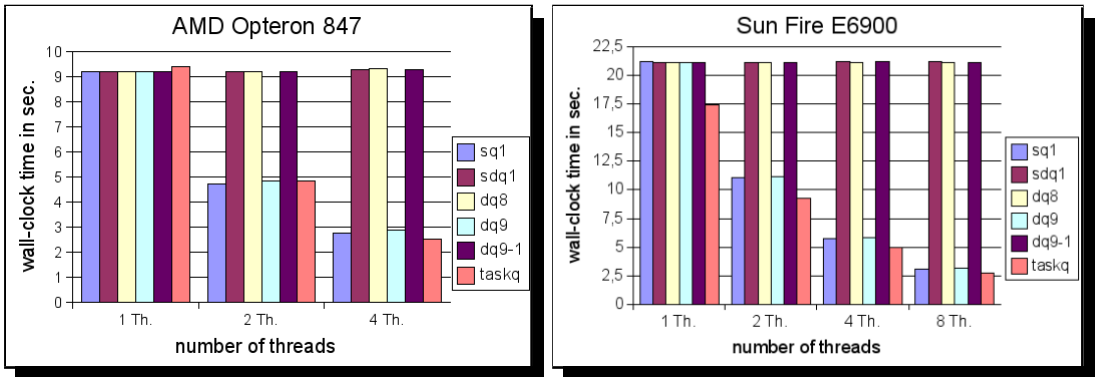


Figure 5.22: Wall-clock times for the cholesky factorization (best of three runs)

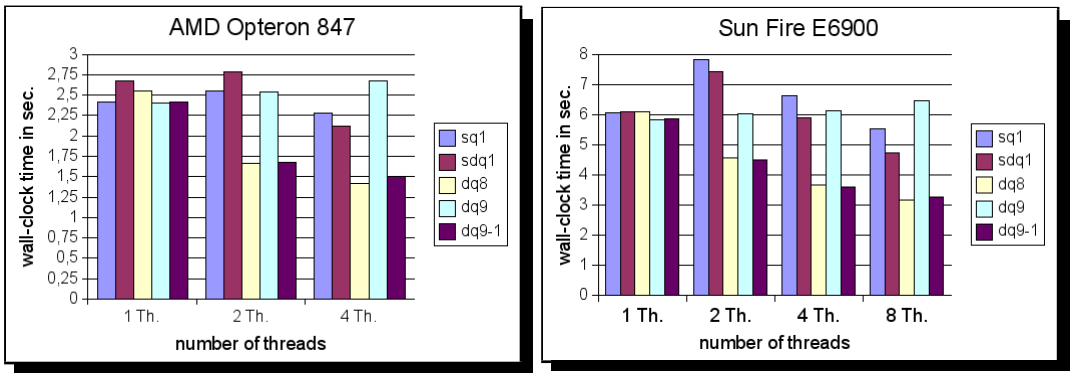


Figure 5.23: Wall-clock times for labyrinth-search (best of three runs)

The bottom line from our experiments is that there is no clear winning task pool implementation. It depends on the application, which task pool variant is suited best.

As can be seen, the performance of the task pools implemented inside the two compilers using Intel's taskq is comparable to (and in some cases even better than) our implementations for the cholesky example. When many tasks are generated and stored in the pools (as is the case for quicksort), our optimized task pools are able to outperform the Intel implementations, though.

5.4.4 Related Work

A detailed analysis of several task pool implementations with POSIX threads and Java threads can be found in the article of Korch and Rauber [KR04]. They conclude that a combination of private and a public queues for each thread works best for their three benchmark applications.

An OpenMP extension that could help to deal with irregular problems, the workqueuing model, has been suggested by Shah et al. [SHPT99], and performance measurements for this extension have already been discussed in Section 5.4.3. Since it is expected that a similar extension will be accepted into OpenMP 3.0 [ACD+07], this could make our task pools obsolete. Nevertheless, the experiences gained from implementing the different variants may help implementers of the extension in OpenMP compilers.

Another approach was proposed by Balart et al. [BDG+04]. They suggest to relax the specifications of the sections directive allowing a section to be instantiated multiple times. Additionally, they suggest to execute code outside of any section by a single thread. Each time this thread detects a section instance, it will insert this section into an internal queue. The section instances inserted into the queue are executed by a team of threads.

This closes our work on irregular algorithms for now. In Sections 6.1 and 6.2 enhancements to the specification are described that would make working with these kinds of algorithms even easier.

5.5 Other Patterns

In this section, some patterns that are too small for their own section are described. In detail, these are the observer pattern (Section 5.5.1), the RW-lock (Section 5.5.2), the shared queue (Section 5.5.3), the once pattern (Section 5.5.4), the pipeline pattern (Section 5.5.5), the thread-safe containers (Section 5.5.6) and the thread-storage (Section 5.5.7).

5.5.1 Observer

The work presented in this subsection was implemented by Florian Bachmann as part of his diploma thesis. We are only providing a short summary here, an extended description can be found in his thesis [Bac07].

The *observer* pattern (also known as *publish/subscribe*) is one of the original behavioral patterns described by Gamma et al. [GHJV95]. Like the Singleton shown in Section 5.2, it is an object-oriented pattern that has been frequently implemented. The difficulties and the reason we chose to implement it lay in its use in a multi-threaded setting.

As the name suggests, the observer pattern is mainly used to monitor the state of an object (called *subject*) and notify other objects (called *subscribers*) as soon as the subject changes its state. Since C++ does not have a native event handling system, this needs to be done manually.

Regarding implementation in a multi-threaded setting, the main difficulties lay in the protection of the subject's methods from concurrent access. For example, the subject contains a method that allows objects to subscribe. These subscribers are kept in an internal data structure of the subject, which of course needs to be protected from concurrent access, as multiple threads could attempt to subscribe objects at the same time. How this is achieved in detail can be found in the diploma thesis by Bachmann [Bac07].

5.5.2 RW-Lock

The work presented in this subsection was implemented by Christopher Bolte. The OpenMP memory model has so-called relaxed-consistency semantics, which implies that even to read a shared location in memory, use of a of synchronization construct is necessary (see Section 2.4.8 for details.

This restriction seriously limits concurrency in many cases, especially when values are frequently read, but only rarely changed. A so-called *Reader-Writer Lock* (short: *RW-Lock*) provides a valid workaround in this case. It allows multiple readers to enter the critical region protected by it. The lock is only exclusive, as long as a writer has acquired it. The interface of the corresponding component in AthenaMP is shown in Figure 5.24.

As can be seen in the interface, this lock type is as generic as the ones described in Section 5.1. Just specify the appropriate lock adapter (described in detail in Section 5.1.1) via the `LockPolicy` template parameter, and you can use this lock in any parallel programming system that supports lock adapters.

There are two important subclasses of RW-Locks. The distinction between them is most visible when there are readers still holding the lock and at least one writer is waiting to acquire it. *Reader-Preferred* RW-Locks let new readers acquire the lock, even when there are writers waiting for it already. *Writer-Preferred* RW-Locks are more fair to writers, as new readers have to wait in this case until the writers had their turn. Which one is more suitable depends on the application. Our RW-Locks support both strategies via the `Strategy` template parameter.

The main implementation strategy that enables our RW-Locks is the use of multiple locks, one for each reader and one for the writers. If a reader wants to acquire the lock, it acquires only one reader-lock, which is different from all other reader locks. Therefore, it can continue. If a writer wants to acquire the RW-Lock, it needs to acquire the writer

```
1  template <class LockPolicy=omp_lock_ad,
2            typename Strategy=is_reader_preferred>
3  class rw_lock {
4  public:
5    explicit rw_lock( const int threads = omp_get_max_threads(),
6      const double wait_time = 0.001);
7    ~rw_lock();
8    void set_r();
9    int test_r();
10   void set_w();
11   int test_w();
12   void unset();
13   bool try_upgrade();
14 };
```

Figure 5.24: RW-Lock interface

lock and all reader locks. Of course, this is a time-consuming operation when compared to normal locks, therefore careful profiling is needed when using it.

5.5.3 Shared Queue

The work presented in this subsection was inspired by Mattson et al. [MSM04] and implemented by Florian Bachmann as part of his diploma thesis. Note that the shared queue is not safe to use in general because of constraints in the OpenMP memory model, although it works just fine on x86 hardware. See the end of this subsection for details.

In Section 5.4, we have described irregular algorithms and the problems associated with implementing them. As a general solution, task pools were suggested and implemented. In the common case that merely two threads need to communicate, a task pool as described earlier is overkill, as it needs locking in many cases and also does not make sure that messages from one thread reach another thread in order. If you need this property or want to avoid the overhead associated with locking, a shared queue may be a solution.

It guarantees delivery of messages from exactly one thread to exactly one other thread in order, without locking. However, it is thread-safe only for this case. As soon as more than one thread is pushing messages into the queue or more than one thread pops messages out of the queue, the result is unspecified behavior.

The interface of the shared queue component is shown in Figure 5.25. The class has two template parameters: `Type` indicates the type of the messages to be delivered through the queue. `HasConstSizeRunTime` indicates, whether or not the `size` method is able to complete in constant time. If the parameter has *true* as its value, the queue keeps track of how many elements it presently holds by incrementing or decrementing an atomic counter in each `push`/`pop`- operation. This makes the `size` method very fast, but slows down

```
1  template<class Type, bool HasConstSizeRunTime = false>
2  class shared_queue
3    : private size_wrapper::size_wrapper<HasConstSizeRunTime>
4  public:
5    shared_queue();
6    shared_queue(const shared_queue& right);
7    shared_queue& operator=(const shared_queue& right);
8    ~shared_queue();
9    Type front() const;
10   bool is_empty() const;
11   int size() const;
12   void push(Type task);
13   void pop();
14 };
```

Figure 5.25: Shared Queue interface

the `push`/`pop`- operations. If `size` is only rarely used, it therefore makes sense to set `HasConstSizeRunTime` to false, which gets rid of the counter and makes the `size` method slower, because it has to count all elements each time it is called.

Implementation-wise, the shared queue is constructed around the idea that the data structure has two pointers internally: one for the `head` of the queue and one for the `tail`. The `head`-pointer is only accessed by the receiving thread (`pop`), the `tail`-pointer is only accessed by the sending thread (`push`). Therefore, no locking should be necessary.

Unfortunately, it does not work this way with the current OpenMP memory model for several reasons. First, even though both threads use different pointers, they may point to the same location in memory. Therefore, both threads may modify the same location in memory at the same time, leading to unspecified behavior. Second, the OpenMP memory model has the concept of the so-called *temporary view* of each thread, which can only be updated by carefully utilizing `flush` directives in combination with synchronization directives (see Section 2.4.8) – which we wanted to avoid here.

Last but not least, only careful placement of memory barriers (in the form of `flush` directives) can prevent the compiler and the processor from performing operation reordering to increase performance. These reorderings can also lead to unspecified behavior here. Therefore the shared queue can not be implemented correctly and in a portable manner in OpenMP without locks – thus we decided to not include it into AthenaMP. Our experiments indicate that it works on current x86-hardware, though, probably because the memory model of this architecture provides more guarantees than OpenMP's.

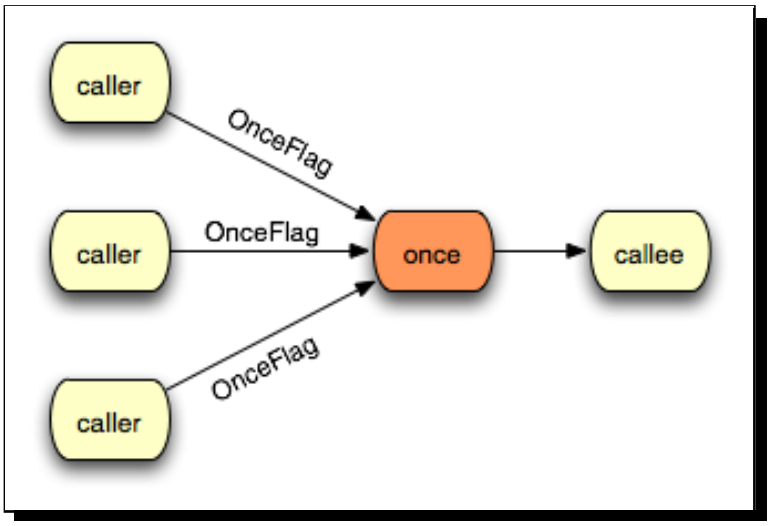


Figure 5.26: How the `once` function works in general

5.5.4 Once

One feature that is missing from OpenMP but frequently found in other parallel programming systems is the ability to specify a function that is supposed to be carried out exactly once. An important use case is e.g. initialization code. In POSIX threads this feature is called `pthread_once`, in the Boost library the name is `call_once`. OpenMP has the `single` directive, which is similar, but not quite equivalent. This work-sharing directive makes sure that only one thread carries out the enclosed code. This is different from `once`, because if the same `single` directive is encountered multiple times (e.g in a loop), the enclosed code will be carried out multiple times as well. When nested parallelism comes into play, `single` does not preserve once-semantics as well. Of course, all of these cases can be worked around by the programmer, but the same functionality can also be provided conveniently as a generic component called `athenamp::once`. Figure 5.26 shows how it is done in principle in AthenaMP.

The caller and the callee in the figure are both written by the application programmer (shown in yellow), while the once function is a library call that handles all the details of making sure that the callee is called only once (in orange). The variable of type `OnceFlag` serves a double role: first, it is an implementation detail - the value of the static variable is checked inside the library to see if the function has been called already. Second, it identifies which region to protect. If two different functions are specified with the same `flag`, only one of the two will be called. Figure 5.27 shows with a simple example, how the function can be used.

This example uses a C++-functor to demonstrate our functionality, in case a more C-like interface is needed, `once` also works on function pointers. The functor solution has the advantage, that it allows to pass parameters to the function via the functor, which

```
1  class OnceFunctor {
2    public:
3    void operator() () {
4      initialize_my_data ();
5    }
6  };
7
8  void main ()
9  {
10   #pragma omp parallel
11   {
12     static OnceFlag flag = ATHENAMP_ONCE_INIT;
13     OnceFunctor func;
14
15     athenamp::once(func, flag);
16   }
17 }
```

Figure 5.27: How to use the `once` function

is not possible for function pointers - except by using the ugly workaround of casting void-pointers. Another example has already been shown in Figure 5.11 in Section 5.2.2. Another solution to the problem can be found in Section 6.3.

5.5.5 Pipeline

The work presented in this subsection was inspired by Mattson et al. [MSM04] and implemented by Florian Bachmann as part of his diploma thesis. Pipelines are an important building block of many parallel programs. The functional decomposition described in Section 2.1.1 frequently results in a program structure resembling a pipeline. An example are e. g. image filters that are applied one after another in different threads. A pipeline comprises a linear sequence of stations. At each station, a function is carried out on a data item, before it is handed down to the next station. As soon as the pipeline is populated with work items, a speedup equivalent to the number of stations is possible.

The difficulties in implementing a multi-threaded pipeline lay in two fields: requirering as little synchronization as possible in-between the stations, and making the pipeline as usable as possible without requiring too much initialization work from the programmer.

We have tried to solve the first problem by using the shared queues described in Section 5.5.3 for synchronization. However, as is described there, the shared queues are not safe to use in general for OpenMP. Utilizing a lock-free queue as implemented e. g. in the Intel Threading Building Blocks [Int07] would be a better solution for this problem and is being investigated at the moment, although for this approach it becomes necessary to rely on architecture specifics, which we have tried to avoid so far to increase portability.

In order to solve the second problem, two different version of the pipeline were implemented: a non-typesafe one using void pointers to exchange data in-between the pipeline stations and a typesafe one using templates. The void-pointer version is very easy to setup, the pipeline is able to connect the stations without intervention from the programmer. Since the types passed into the queues cannot be checked for correctness by the compiler, this version is suboptimal. The typesafe pipeline solves this problem, but for initialization the programmer must connect the stations himself. Because of these problems, the pipeline is not yet part of AthenaMP.

5.5.6 Thread-safe Containers

The work presented in this subsection was implemented by Florian Bachmann as well. Thread safety is a concept every programmer of threaded programs must know about. As has been described in Section 2.4.9, a function (or library in general) is thread-safe, if it can be used from multiple threads simultaneously and still produces correct results. Also in this section, we have shown that OpenMP makes far-reaching guarantees with regards to the thread safety of the standard libraries.

Judging from those, one could conclude that e. g. C++'s Standard Template Library (STL) is thread-safe, since it is a library and part of the language. Unfortunately, this is not the case as of this writing. No C++ implementation we know of offers a thread-safe STL. For this reason, access to e. g. the containers offered by the STL must be protected from concurrent access. This can be done by the programmer, but of course it could also be done automatically using the well-known *Decorator* pattern [GHJV95].

As an example of this technique, we have implemented decorators for three STL-containers: `vector`, `deque` and `list`. They are called `vector_ts`, `deque_ts` and `list_ts`, respectively. Each time a method is called on these containers, an implicit lock is set to protect them from concurrent access. Note that there are better performing options available to achieve this (e. g. by using lock-free data structures), but unfortunately none of these are portable. Since the interface of these containers is the same as for the original ones, we are not showing it here. Performance numbers do not make much sense here either, since of course the original containers are faster in the sequential case, yet when using them in a multi-threaded systems the same techniques as in the decorators must be employed, resulting in exactly the same performance.

5.5.7 Thread Storage

The work presented in this subsection was implemented by Christopher Bolte. Having private data in each thread is a typical requirement in multi-threaded programs. In OpenMP, there are three ways to make data private: the `private` clause, declaring local variables inside a parallel region, and the `threadprivate` directive. Neither of them is applicable to an important use-case: when data are to be collected in a parallel region privately

```
1  #include <iostream>
2  #include <omp.h>
3  #include <vector>
4
5  const int N=10000000;
6  const int numthreads = 4;
7
8  void main()
9  {
10   std::vector<int> A (numthreads, 0);
11
12   double start = omp_get_wtime ();
13   #pragma omp parallel for num_threads (numthreads)
14   for (int i = 0; i < N; ++i) {
15     ++A[omp_get_thread_num()];
16   } // end for
17   double end = omp_get_wtime ();
18
19   std::cout << "Time (sec): " << end-start << std::endl;
20 }
```

Figure 5.28: An example showing false sharing

to each thread (with no protection from concurrent access provided), but need to be accessed either by a different thread later, or by the master thread after the parallel region ends. This may be necessary e. g. when implementing reductions manually. None of the techniques mentioned above allow this.

There is only one way to satisfy the use-case described above in OpenMP presently: by allocating a shared array (or vector) of data. Each thread can then access its own data in the array, while no protection from concurrent access is necessary. At a later point in the program, these restrictions may be lifted and access from different threads may be allowed, which is a very flexible and widely-deployed workaround. Unfortunately, it has a problem as well: *false sharing*. This problem and our solution to it are explained in the next paragraph.

Most shared memory architectures today are *cache-coherent*, meaning the caches on all processors are kept consistent. Caches do not operate on a single byte basis, but the smallest unit for a cache is a so-called *cache line*, which varies in size from architecture to architecture. A typical cache line size at the time of this writing is 128 bytes. Operating on cache-lines has significant disadvantages, as soon as many processors try to write to different data on the same cache line repeatedly. Each write will invalidate the whole cache line, which requires an update of the caches on the other processors. This results in significant traffic on the memory bus. An example will make this clearer and is shown in Figure 5.28.

```
1  #include <iostream>
2  #include <omp.h>
3  #include "misc/thread_storage.hpp"
4
5  const int N=10000000;
6  const int numthreads = 4;
7
8  void main()
9  {
10   athenamp::thread_storage<int> A (numthreads, 0);
11
12   double start = omp_get_wtime ();
13   #pragma omp parallel for num_threads (numthreads)
14   for (int i = 0; i < N; ++i) {
15     ++A();
16   } // end for
17   double end = omp_get_wtime ();
18
19   std::cout << "Time (sec): " << end-start << std::endl;
20 }
```

Figure 5.29: An example showing no false sharing, because of the `thread_storage`

Vector `A` has merely four integer elements, therefore it fits in a single cache line on most architectures. Each thread in the parallel region accesses exactly one element in the vector repeatedly - thereby constantly invalidating the caches of the other threads, that need to update their cache, before they can complete their operations. All threads share the same cache line and this is why this performance mistake is called *false sharing*.

The use-case described at the beginning of this subsection (putting private variables in a shared array, so they can be accessed from different threads later) has exactly this problem. The canonical solution to it is *padding*. This means that after each element in the array, a buffer is inserted, which pushes the next element into the next cache line. Of course, many programmers do not know about false sharing and the workaround is also a burden for them. For this reason, we have implemented a generic component called a `thread_storage` that employs the solution automatically. In Figure 5.29, our example program is shown again, this time with our generic component.

The only differences are in line 3 (where a different library is included), line 10 (where we initialize the `thread_storage` instead of a simple vector) and line 15 (where each thread accesses its part of the data structure). It is not even necessary to specify the thread number with the `thread_storage`, as it defaults to `omp_get_thread_num`.

Although the difference in-between the programs with regards to code is tiny, the performance difference is not: on our test system with 2 dual-core AMD Opteron processors

the version using the `thread_storage` needed 0.11 seconds to finish the program, while the version with false sharing took four times as long with 0.44 seconds.

The `thread_storage` is also able to solve the use-case sketched above: if data need to be accessed by a different thread later, this can be done by specifying the appropriate thread number to its `operator()` method. The thread storage itself provides no protection from concurrent access, though.

Since the component is generic, it is possible to store any type of data in it. It is also possible to specify a different cache line size (default is 128 bytes). The amount of padding to be inserted after each element is computed at compile time from the given cache line size and the size of the stored data type using template metaprogramming.

5.6 Chapter Summary

This section summarizes our efforts to create powerful generic components for OpenMP in a library called AthenaMP [Sue07a].

Locks are still an important building block for concurrent programming, but the locks provided by most parallel programming systems are rather basic and error-prone. In Section 5.1, we have presented higher-level classes that encapsulate the locking functionality and provide additional value, in particular automatic lock hierarchy checking, automatic setting of multiple locks in a safe order, as well as exception safety. We implemented our ideas in a generic way that is decoupled from the parallel programming system used.

In Section 5.2, we have described and evaluated different implementation possibilities for a thread-safe singleton with C++ and OpenMP, a work that has to our knowledge never been attempted using these systems before. The smallest and fastest implementation variant described there is the Meyers Singleton, which is unfortunately not yet safe to use with many compilers.

In Section 5.3, we have described how to implement data-parallel patterns modelled after the Standard Template Library with OpenMP, namely `modify_each`, `transmute`, `combine`, `reduce`, `filter` and `prefix`. Common characteristics were their iterator-based interface, the dynamic selection of the best-suited implementation depending on the iterator-type supplied, and their ability to nest inside each other. The patterns were described in detail and with examples, along with performance measurements for a simple benchmark and the game-like application OpenSteerDemo.

Efficient parallelization of irregular algorithms is an ambitious goal that often can be tackled with task pools. We have presented several variants of task pools along with their implementation in OpenMP in Section 5.4. To assess the performance of the variants, we have implemented three irregular algorithms: quicksort, labyrinth-search and cholesky factorization. Results show that the correct selection of a task pool variant has a significant impact on the performance of an application. There was no universally best variant, but the suitability depended on the pattern of accesses to the task pool. Applications that generated many tasks and access the task pool frequently benefited from the usage of dis-

tributed private queues. Applications that accessed the task pool infrequently, in contrast, needed good load balancing, and therefore gained more from a central shared task queue. This part is not yet included in AthenaMP.

The chapter closed with a summary of other patterns we have implemented, some of which were too small to be described in their own section, some of which were implemented by our students, and some had problems that prevented them from becoming part of AthenaMP just yet. We have described the observer pattern (Section 5.5.1), Reader-Writer locks (Section 5.5.2), shared queues (Section 5.5.3), the once-pattern (Section 5.5.4), pipelines (Section 5.5.5), thread-safe containers (Section 5.5.6) and the thread-storage (Section 5.5.7).

Chapter 6

A Specification Approach to Enhancing the Power of OpenMP

While OpenMP is a relatively easy-to-use and powerful parallel programming system, it has its problems as well. This chapter describes some of them, along with the solutions we have investigated. Some problems are worked around, where others can only be solved by extending the OpenMP-specification. Where the latter is necessary, we provide a reference implementation in the OMPi research compiler [DGLT03], before proposing the change to the OpenMP language committee. This work is part of our research on enhancing OpenMP, see Figure 6.1.

We have found the first problem with OpenMP while implementing the task pools described in Section 5.4: They are very hard to implement without *busy waiting*, a term that is explained along with a solution to the problem in Section 6.1.

Another problem related to irregular algorithms is the lack of easy ways to stop all threads from working in a parallel region, as soon as a solution is found. We address this problem in Section 6.2 with a proposed extension to OpenMP. The (smaller) problem of how to go to the end of the parallel region as soon as a solution is found is addressed there as well.

In Section 6.3 I will shortly sketch some of the work done while serving in the OpenMP language committee. This is all put into a single, short section because I can hardly be considered the driving forces behind these efforts and have neither performance numbers nor reference implementations for any of the proposals described there. A short summary of what has been achieved closes the chapter in Section 6.4.

6.1 Avoidance of Busy Waiting

The work presented in this section includes material from a workshop-publication in 2004 [SL04] and a workshop-publication in 2006 [WSL06]. The implementation of the task pools sketched in Section 5.4 was relatively straightforward with OpenMP, but we encountered a problem for which OpenMP does not provide an adequate solution: Each time a thread tries to extract a task but detects an empty task pool, it has to wait until another thread inserts a new task. Korch and Rauber [KR04] solved the problem for

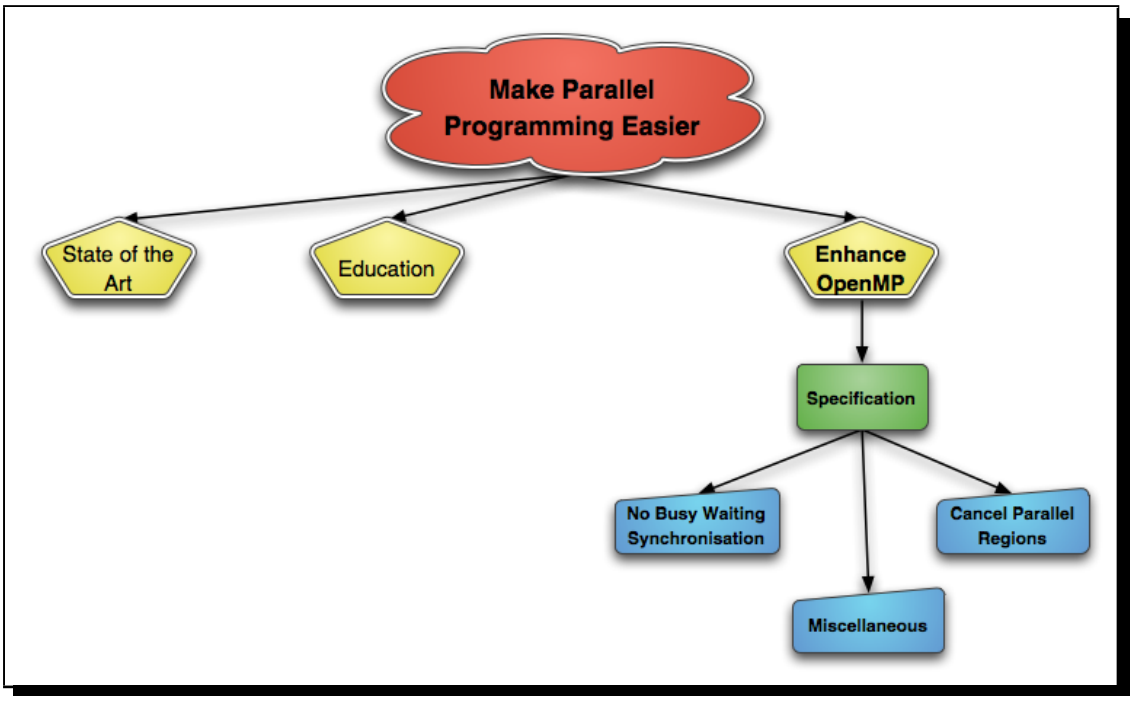


Figure 6.1: Objectives, aims and contributions of this thesis (OpenMP specification)

- **# pragma omp yield**:
  release the processor so that another thread can run on it
- **#pragma omp sleepuntil (scalar_expression)**:
  sleep until scalar_expression becomes true

Figure 6.2: Scheduling in a nutshell

their POSIX threads task pool implementations using condition variables, and the `wait-notify` methods in Java, respectively.

Unfortunately, there is no mechanism in OpenMP to put a thread to sleep until an event occurs or a condition becomes true. Another way to put this is: there is no point-to-point synchronization available in OpenMP. In our task pool implementations, we therefore had to fall back to *busy waiting*, which means polling a condition repeatedly, resulting in unnecessary idle cycles. For this reason, we suggest OpenMP extensions to solve the problem, which are sketched in Figure 6.2. These simple extensions can be used to avoid busy waiting in a task pool, and are also helpful in other contexts.

The problem is explained on a broader scale in Section 6.1.1. Afterwards, Section 6.1.2 specifies our proposed solution, and Section 6.1.3 gives our reasons for the design. Finally, in Section 6.1.4, the specification is applied to our examples from Section 5.4.2,

and some more ways to use the new functionality are shown. A reference implementation of the suggested changes to the OpenMP functionality can be found in a special version of the OMPi Compiler [DGLT03] that is available from the authors on request.

6.1.1 Problem Description

The problem of busy waiting manifests if a thread has to wait for a condition to become true before it can continue. Most of the time, this condition is made true by a different thread – therefore those two threads need a way to communicate without impacting all other threads, which is called point-to-point synchronization*point-to-point synchronization*. In the case of our task pools, for instance, function `tpool_get` is supposed to return an element from the pool, but if there is no element left, it has to wait for work to become available. The most sparing way for the computing resources to implement this waiting is to put the thread to sleep until the condition becomes true. Unfortunately, there is no functionality available in OpenMP to support this, though.

As a valid workaround, the programmer may poll a condition repeatedly, thereby wasting processor time. This approach is known as *busy waiting* or *spinning*. To give another example, busy waiting is also required for pipelined algorithms and the pipeline component described in Section 5.5.5, where a stage has to wait until a previous stage has completed its work. Busy waiting is best avoided, especially when other threads are waiting for the processor to become available, or when power consumption is an issue, e.g. in embedded systems.

Novice OpenMP programmers may resort to using locks to solve the problem. In their approach, the waiting thread tries to set an already set lock, and is put on hold as a result. As soon as work is available, a different thread will unset the lock, thereby enabling the waiting thread to continue. Although this approach often works, it is not compliant with the OpenMP specification, because the lock is unset by a different thread than the owner thread, which leads to unspecified behavior (see Section 3.2.2 for details on this common mistake). Furthermore, there is no guarantee that a thread waiting on a lock is put to sleep at all in OpenMP (spinning is also allowed), and therefore this approach is even more flawed.

The problem described above cannot be solved in OpenMP satisfactory as of now, since there are no directives for scheduling available. Therefore, Section 6.1.2 suggests a possible addition to the OpenMP specification that makes the suggested workarounds (busy waiting or non-compliant use of locks) obsolete. The problem has already been noticed by Lu et al. [LHZ98], who suggested the introduction of condition variables (as found in POSIX threads) in 1998. A different solution centered around point-to-point synchronization is sketched by Ball and Bull [BB03]. Our solution tries to combine the power of condition variables with the ease of use of OpenMP.

Let us make one more fact perfectly clear: The newly proposed functionality is not useful for the common case in computing centers today, where one processor is exclusively available for each thread. It is intended for the more general case that multiple threads are

competing for the available processors. With the advent of multi-core CPUs in common desktop systems and the expected shift to multi-threaded applications, we soon expect this case to be the dominant one.

6.1.2 Specification

We suggest two new directives:

#pragma omp yield
Similar to the POSIX function `sched_yield`, this directive tells the scheduler to pick a new thread to run on the current processor. If no new thread is available, it returns immediately. The directive provides a simple way to pass on knowledge on what is important and what not from the programmer to the runtime system and operating system scheduler. As a second new directive, we propose:

#pragma omp sleepuntil (scalar_expression)
This directive puts the current thread to sleep until the specified scalar expression becomes true (non-zero). The expression is occasionally tested by the runtime system in the background. Before each test, a `flush` is carried out automatically, to keep the temporary view of the thread consistent with memory. An implementation of the directive is not required to wake up the sleeping thread immediately after the expression becomes true, nor does it have to wake it up if the expression becomes true and becomes false again shortly afterwards. Not all threads waiting on the same expression have to wake up at the same time either. It is unspecified, how many times any side-effects in the evaluation of the scalar expression occur.

6.1.3 Rationale

The `yield` directive is inspired by its POSIX counterpart, `sched_yield`. It offers an easy to use way to influence the scheduling policies of the operating system. This can be important when computing resources are sparse and the programmer wants to optimize program throughput. An example of this would be calling the `yield` directive at the end of every pipeline step in a pipelined application, to get values through the pipeline as fast as possible.

We know of no scheduling primitive in any other parallel programming system that is as powerful and easy to use as the proposed `sleepuntil`.This directive is as powerful as condition variables, yet it lacks their difficult usage. The directive can be emulated by wasting time in a loop, but this would be busy waiting and wasteful to the available computing resources, as outlined in Section 6.1.1.

The proposed changes are fully backwards compatible to the existing OpenMP specification, since no behavior of existing OpenMP functionality is altered in any way.

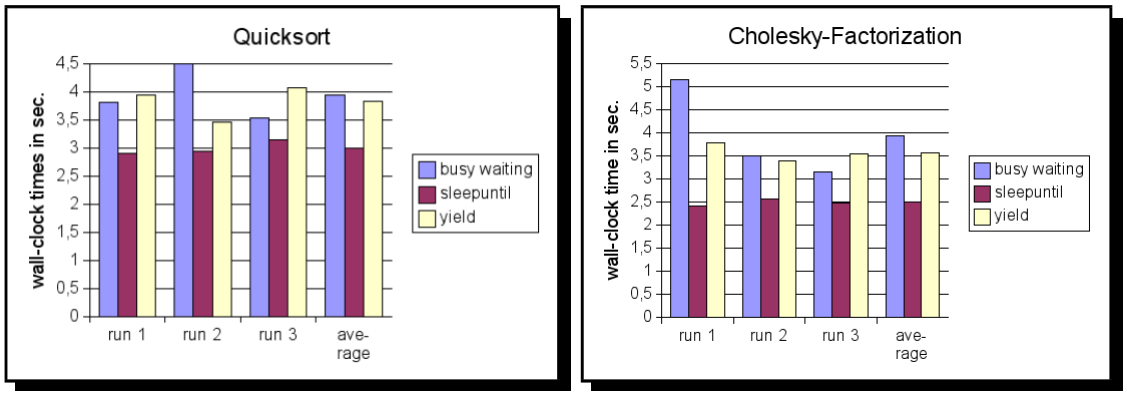


Figure 6.3: Wall-clock times in seconds for task pool sq1 on oversubscribed system

6.1.4 Application

We have emphasized in Section 6.1.1 that there is no way for a parallel algorithm using task pools to wait for a new element out of an otherwise empty pool, without constantly polling the pool. There are two approaches to solve this problem with our newly proposed directives. The first one calls the `yield` directive whenever there is no work in the pool, which will put the thread to sleep if another thread is waiting for a processor to become available. Chances are, that a different thread will produce work for the task pool. If there is no other thread from the same application, a context switch may occur and a different application will run on the processor, allowing for a higher throughput on the machine. Finally, if there is no other thread waiting for the processor, the call to the `yield` directive will just return and no harm is done. A second possible solution is the following:

```
#pragma omp sleepuntil (!tpool_is_empty (pool))
```

This solution offers a more fine-grained control over when the thread is supposed to wake up again, as the thread will sleep until something has been put into the task pool and not just an unspecified amount of time as with the `yield` solution. After wake-up, it is still necessary to check if the task pool is not empty again, as no locking of any sort is involved here. The thread might have been woken up at a time when the pool was not empty, but when it tries to actually get a task from the pool, a different thread might have already popped the task.

It is difficult to measure the impact of the proposed directives, as they are most useful on fully loaded systems. We have therefore oversubscribed a system (Sun Fire E6900 with 8 processors) by starting our benchmark applications with 32 threads using the most simple task pool sq1. The results are shown in Figure 6.3.

A different use case for both new directives is testing. When testing OpenMP compilers or performing tests for OpenMP programs, it is often useful to force the scheduler into

 # pragma omp cancelregion:
request cancellation of parallel region

 # pragma omp exitregion:
take current thread to end of parallel region

int omp_get_cancelled (void):
has the current region been cancelled ?

pragma omp barrier **oncancel**:
execute scope if thread is cancelled in barrier

 # pragma omp onbarriercancel:
execute scope if thread is cancelled in implicit barrier

Figure 6.4: Thread cancellation in a nutshell

certain timing behaviors that could not be tested otherwise (e. g. stalling one thread, while all other members of the team go ahead and run into a barrier). This is not possible with the present OpenMP specification (except with busy waiting again), and can be very useful to test for hard to catch errors. An example to stall execution of one thread for 100 milliseconds is shown below:

```
double now = omp_get_wtime(); /* save current time into now */
#pragma omp sleepuntil (omp_get_wtime() >= now + 0.1)
```

This closes our work on avoiding the busy waiting problem, in the next section a solution for a different problem with OpenMP is introduced – the problem of how to cancel work in a parallel region.

6.2 Cancelling Work in Parallel Regions

The work presented in this section is derived from a conference publication [SL06b]. It explores an important issue in developing irregular parallel algorithms with OpenMP, which is the missing ability to cancel threads in a parallel region. While a (not completely functional) workaround for the issue is suggested in Section 6.2.1, the main contribution of this section is a proposal for new functionality to solve the problem in a convenient and easy to use way on the language level. The suggested additions to OpenMP are previewed in Figure 6.4. A working implementation can be found in a special version of the OMPi compiler [DGLT03], which is available from the authors on request.

```
1  #pragma omp parallel
2  {
3    while (!exit_found) {
4      while ((task = pop (taskpool)) != NULL) && (!exit_found)) {
5        mark (labyrinth, task);
6        if (!inspect_field_for_exit (task))   // inspect all neighbors
7          push (neighbors (task), next_taskpool); // no exit was found
8        else
9          exit_found = true;  // an exit was found
10       #pragma omp flush (exit_found)
11     }
12     #pragma omp barrier
13     #pragma omp single
14     {
15       taskpool = next_taskpool; // switch the taskpools
16       next_taskpool = NULL;
17     } // implicit barrier (includes flush)
18   }
19 } // end of parallel region with implicit barrier
```

Figure 6.5: Parallel breadth first search using a flag for thread cancellation

As a running example, we use breadth-first search on a labyrinth. The algorithm and its implementation are explained shortly in Section 6.2.1, where we will also explain why thread cancellation is a problem. Furthermore, a first (but inconvenient) workaround for the issue is presented in this section. Section 6.2.2 shortly introduces a few basic terms often used when talking and writing about thread cancellation. An actual specification of the new functionality to solve the problem on the language level is given in Section 6.2.3, followed by the rationale for some of our design decisions in Section 6.2.4 and a short discussion on implementation and performance issues in Section 6.2.5. Section 6.2.6 puts the specification in perspective, by applying it to the labyrinth example.

6.2.1 Problem Description

In labyrinth-search, the objective is to find the shortest path through a labyrinth, from a given entry to a single exit. This problem is not merely a theoretical one, but has practical relevance e. g. for mapping electrical circuits on a chip. We consider a breadth-first search algorithm, which is not necessarily the fastest choice, but is simple enough to serve as an example here and to still include all the problems we want to illustrate. A very broad sketch of the algorithm is presented in pseudocode in Figure 6.5.

The algorithm starts by putting the entry position of the labyrinth into the task pool (not shown in the pseudocode). Afterwards, it spawns a parallel region (line 1). Then, one of the threads takes a position out of the task pool (line 4), marks it on a map as processed

(line 5), evaluates all neighbors by checking the four possible directions for walls (line 6), and checks if an exit is found on any of them. If no exit was found and the neighbor-positions have not been evaluated before (this check is not shown in our pseudocode), the neighbors are put into the task pool to be processed in the next step (line 7), possibly by a different thread. If an exit is found, a flag is set that indicates this fact (line 9). We need to be careful with the different positions in the task pool, since only positions with the same distance to the start should be evaluated together, or else the breadth-first search will degenerate. Therefore, only positions with the same distance to the entry are kept in the task pool, while the neighbors are put into a different one (called `next_taskpool`). As soon as the task pool is empty, both task pools are switched by a single thread, and the computation proceeds with the former `next_taskpool` (lines 15-16). When the algorithm depicted in Figure 6.5 is done, a single thread follows the marks set in the labyrinth (line 5) from the exit point back to the entry point and identifies the shortest way.

In the listing above, a flag is used to indicate when the threads in the parallel region should finish their work, because an exit was found (indicated by `exit_found` == true). We know of no other way in OpenMP to indicate that the threads should end their work in a parallel region. In the next few paragraphs, we will point out the problems with this approach. Section 6.2.3 will present an extension of OpenMP that leads to an easier solution, which we will discuss in Section 6.2.6.

The Problem with Flags

When using flags to indicate that the parallel region should be aborted, great care has to be taken with checking these flags by the programmer. In our example, it might happen that one thread enters the while loop (line 3), finds an exit, sets the appropriate flag, and afterwards hangs in the barrier (line 12), because another thread does not enter the next iteration of the while loop at all, as the flag is indicating now that an exit was found! The program will exhibit undefined behavior in this case (most likely a deadlock), because in OpenMP the sequence of barrier constructs encountered must be the same for every thread in the team. Thus, the code in Figure 6.5 is not correct, and it is not safe to use without further adjustments that would make it even harder to read and explain!

Flags that indicate when a parallel region is to be cancelled give rise to yet another problem: Due to the OpenMP memory model, the flags have to be updated with a `flush` directive before their values are guaranteed to be up to date. This step is frequently missed by inexperienced OpenMP programmers, as was shown in Section 3.2. The consequence is similar as sketched above: the program will potentially deadlock, because the thread which set the cancel flag has got its current correct value and will exit the loop, whereas other threads might still use the old value and continue with it. This is not an issue in our example, as there is a flush included in many OpenMP directives (e. g. in the implicit barrier on line 17). Nevertheless, when the code is only slightly altered, the problem may surface.

Let us summarize the problems we have identified so far with thread cancellation in OpenMP:

- there is no easy way to branch out of a parallel region, the only possible workaround is to use flags
- it is difficult to work with flags indicating that a region should end, at least as soon as barriers come into play
- if one forgets to flush a flag, a deadlock may arise

While we have presented a workaround for the main problem (flags manually set and checked by the programmer), it is still cumbersome and error-prone. Therefore we will present another possible solution in Section 6.2.3, based on a proposal to add thread cancellation to OpenMP. The proposal is also useful for the following common scenarios, which could benefit from thread cancellation:

- a cancel button from a user interface was pressed
- a solution has been found in a speculative algorithm

But before we go further, we will review the terms used in this section in the next few paragraphs.

6.2.2 Terms

We speak of *forceful cancellation* when a thread has the ability to cancel another thread from the outside. The cancelled thread may get the opportunity to clean up after itself, yet it does not have the power to decide when to be cancelled, nor to prevent cancellation at all. Asynchronous cancellation in POSIX threads is an example of forceful cancellation. *Deferred cancellation* is an important subcase of asynchronous cancellation, in which the cancelled thread is not terminated immediately, but only at certain predefined cancellation points. Deferred cancellation is supported in POSIX threads as well. With *cooperative cancellation*, in contrast, a thread can only ask for the cancellation of another thread. The cancelled thread has the opportunity to honor this request and cancel itself, to process the request at a later time, or even to ignore it altogether. Java threads support cooperative cancellation.

6.2.3 Specification

The following directives to support cooperative thread cancellation in OpenMP are proposed:

#pragma omp cancelregion
This directive asks all threads in the team to stop their work and go to the end of the parallel region, where only the master thread will continue execution as usual. The emphasis

```
1 #pragma omp barrier oncancel
2 {
3   /* This code is executed only when the region is cancelled before the
4    * thread reaches the barrier or while it is waiting there.
5    * This scope can be used to free thread resources. */
6
7   #pragma omp exitregion  /* end the execution of the thread */
8 }
```

Figure 6.6: Use of the `oncancel` clause

here is on *asks*. The threads in the team are not cancelled immediately, but merely an internal cancel flag is set. The threads are not interrupted in any way and have to poll the flag using one of the constructs described below. An exception is the thread that called the directive: it is cancelled immediately by an implicit call of the `exitregion` directive (explained below). Invoking the `cancelregion` directive on an already cancelled region has no effect except for the implicit call to `exitregion`. It is the task of the programmer to check if the cancel flag has been set, using a new OpenMP runtime library function:

int omp_get_cancelled (void)
This function returns 1 (true) if the cancellation of the enclosing parallel region was requested, and 0 (false) otherwise.

#pragma omp exitregion
This directive is not only useful for thread cancellation, but can be invoked at any point in a parallel region to immediately end the execution of the calling thread. This is accomplished by jumping to the end of the present parallel region, right into its closing implicit barrier (which is of course honored).

There is a problem with the proposal so far: barriers. If a region containing barriers is cancelled, at least one thread (the one calling the `cancelregion` directive) will never reach that barrier. Without further adjustment, one or more of the other threads in the region could hang in the barrier and never recover, since the barrier is not completed.

#pragma omp barrier oncancel
A solution to this problem is proposed in the form of the `oncancel` clause for the barrier directive. A new scope is optionally added to the barrier directive by specifying the `oncancel` clause. The commands in this scope are carried out only if the present parallel region has been or is being cancelled while the thread is waiting on the barrier. This can be seen in Figure 6.6.

It is now possible to use barriers in combination with thread cancellation. It remains the task of the programmer to do the right thing when a thread waiting on a barrier is cancelled, although most of the time he will just free the resources associated with the thread and exit the parallel region afterwards (using the newly proposed `exitregion`

```
1  #pragma omp for
2  for (...) {
3    /* for-loop code */
4  } /* implicit barrier at end of for-loop */
5  #pragma omp onbarriercancel
6  {
7    /* This code is executed only when the region is cancelled before the
8     * implicit barrier or while the thread is waiting there.
9     * This scope can be used to free thread resources. */
10
11   #pragma omp exitregion  /* end the execution of the thread */
12 }
```

Figure 6.7: Use of the `onbarriercancel` directive

directive). Note that if the thread is not finalized with `exitregion`, it will hang in the barrier again (or phrased differently: there is an implicit barrier at the end of the `oncancel` clause). The reasons for this design decision are given in Section 6.2.4. The `oncancel` code is carried out at most once per barrier and thread. Furthermore, if the region is already cancelled when a thread enters the barrier, it will immediately proceed with the `oncancel` code.

For implicit barriers (at the end of work-sharing constructs), a similar construct is proposed:

#pragma omp onbarriercancel
The usage of this directive is similar to the `oncancel` clause suggested above, except that `onbarriercancel` is a standalone construct and must be specified immediately after the implicit barrier it references. This is shown in Figure 6.7.

If the directive is present, all commands in its scope are carried out if the region is cancelled before or while the thread is waiting on the barrier. A `nowait` clause on the referenced work-sharing construct and the `onbarriercancel` directive cannot be specified together. The directive also cannot be specified after a combined parallel work-sharing construct (e. g. `#pragma omp parallel for`), the reasons for this design decision are also given in Section 6.2.4.

OpenMP allows for nested parallelism, i.e., when a member of a team inside a parallel region encounters a new parallel construct, a new subteam is formed. Our proposed extensions apply to nested parallelism as follows: Cancellation requests from inside the subteam only cause members of the subteam to have their cancellation flag set. If another member of the original team requests cancellation however, the cancellation flags for all members of all subteams are set as well, although technically they are not in the same team.

6.2.4 Rationale

Some of the suggested changes could be emulated manually by the experienced OpenMP programmer (such as keeping track of the cancel state of each thread). As has been explained in Section 6.2.1, this is, however, an unnecessary burden and gets difficult when barriers are involved at the latest. Therefore, our proposal introduces the new functionality on a language level.

The `exitregion` directive can be seen as a convenient shortcut, but even without thread cancellation, it is useful as soon as one gets into deeply nested functions inside parallel regions. It allows the programmer to jump to the end of the parallel region immediately, thereby potentially saving many lines of code of conditional statements. If barriers are involved in the parallel region, care has to be taken with `exitregion` for the reasons described in Section 6.2.3, or else the program might deadlock.

We have decided against forceful cancellation as in POSIX threads. On one hand, asynchronous cancellation makes resource deallocation practically impossible. Since one never knows when a thread is cancelled, there is no place to put cleanup code, not even POSIX's cleanup stacks are save to use with asynchronous cancellation. The concept of having cancellation points and deferred cancellation in OpenMP, on the other hand, seemed like overkill, as the amount of functions which are cancellation points is difficult to handle for programmers. Therefore, this proposal suggests cooperative cancellation, which can be found in a similar way e.g. in Java threads. Other good arguments for the use of cooperative cancellation can be found in the Java documentation [Sun99].

A major problem with cooperative cancellation are the `barrier` constructs. The suggested solution (`oncancel` clause, `onbarriercancel` directive) may seem like a lot of overhead to cope with barriers, but the proposal is still easier and more natural than the possible alternatives (such as disallowing barriers with thread cancellation, putting the burden on the programmer to carefully work around them with flags, cancelling barriers forcefully).

We have also decided against automatically including an `exitregion` directive at the end of an `oncancel` or `onbarriercancel` scope. The main reason for this is consistency, as automatically including the directive would cancel the threads waiting on barriers forcefully. This would be inconsistent with the rest of the proposal, where cooperative cancellation is employed. Another reason is nested parallelism. We have specified in Section 6.2.3 that cancelling a parallel region will cancel all subregions as well. But as a subregion might be presently doing uninterruptible work and may contain barriers, the decision not to cancel on barriers automatically allows these subregions to complete their work when interrupted from threads in the upper parallel region, while properly shutting down when cancelled from inside their subregion.

The reason for not allowing the `onbarriercancel` directive after combined parallel work-sharing constructs is that the two main reasons for applying the directive are not valid after a combined directive. There is no need to take care of left over threads hanging in the implicit barrier at the end of the combined construct, as these threads are exactly

where they would be if an `exitregion` clause was specified. There is also no need to clean up any resources, as the programmer must have already done this before the end of the parallel region.

During our internal discussions on the topic of thread cancellation, we have worked out a checklist that each and every proposal we came up with had to pass. This checklist and some explanations of why our proposal passes it are spelled out here to make our design decisions yet more clear:

1. **Backwards Source Compatibility**
   Old code must run unchanged, when translated with a compiler that understands thread cancellation. This is the case, as the behavior of existing OpenMP-constructs is not changed, but only new clauses or directives are added.

2. **Nested Parallelism**
   Each proposal must clearly state how thread cancellation and nested parallelism play together. Our proposal does so, by declaring that when a parallel region is cancelled, all parallel regions that were created by a thread from the cancelled region have their cancel flag set as well.

3. **Barriers**
   Each proposal must cope with the case that a region is cancelled while one or more threads are waiting on a barrier (including implicit barriers), without producing deadlocks. Our proposal does so with the introduction of the `oncancel` clause and the `onbarriercancel` directive.

4. **No Resource Leaks**
   The programmer must have the option to free any resources before a thread is cancelled. Our proposal takes care of this by advocating cooperative cancellation, where the programmer checks if a cancellation request has been put up and can therefore deallocate/free all resources before exiting from a thread. Even resource deallocation while waiting on barriers is allowed with the introduction of the new `oncancel` clause and `onbarriercancel` directive.

5. **C/C++/Fortran Compatibility**
   Each proposal must apply to all three supported languages of the OpenMP specification. Although our proposal only spells out the C syntax of the proposed changes, we believe that these are adaptable to C++ and Fortran as well.

6. **Simplicity**
   Each proposal must be as simple and easy to understand as possible, staying in line with the original OpenMP philosophy. Especially the barrier constructs made this a difficult task, but we think to have met that goal with the introduction of only three new directives, one new runtime library function and one new clause.

6.2.5 Implementation and Performance Issues

We have used the OMPi compiler [DGLT03] as a testing ground for our implementation. One of the benefits of employing cooperative cancellation is ease of implementation, and most of our changes were straightforward:

- adaptation of the compiler frontend to the new directives and clauses
- addition of new runtime library functions for `exitregion`, `cancelregion`, `onbarriercancel` and `omp_get_cancelled`
- a few more minor and locally restricted changes in the runtime library

The most difficult part was the implementation of `exitregion`, which must be able to jump out of deeply nested functions to the end of the parallel region. This was solved using `setjmp` and `longjmp`. The second difficulty was adapting the barriers to the `oncancel` clause. A total rewrite of the runtime support function for barriers was required.

Great care was taken not to impact performance with our changes. Our choice of cooperative cancellation enabled us to implement thread cancellation without any measurable impact on performance. None of our test applications showed any notable slowdown. Neither did the OpenMP Microbenchmarks [BO01], which we used to measure performance of our adapted barrier implementation.

6.2.6 Application

In this section, we apply the thread cancellation functionality to our labyrinth-search example from Section 6.2.1. We had isolated three main problems there:

- there is no easy way to branch out of a parallel region, the only possible workaround is to use flags
- it is difficult to work with flags indicating that a region should end, at least as soon as barriers come into play
- if one forgets to flush a flag, a deadlock may arise

All these issues have been solved, as can be seen in Figure 6.8. Firstly, it is easy now to branch out of a parallel region, as the `cancelregion` directive is a natural fit for the problem (see line 9). Just one directive, and the code will branch to the end of the parallel region on line 26. If barriers are involved like in our case, `oncancel` clauses have to be added (line 12), as well as an `onbarriercancel` clause at the end of the single work-sharing construct (line 21). The second problem is also solved, as there is no need to work with programmer-managed flags to indicate that a parallel region should be finished. Last but not least, the third issue has been made obsolete: there is no need

```
1  #pragma omp parallel
2  {
3    while (!omp_get_cancelled ()) {
4      while ((task = pop (taskpool)) != NULL) && !omp_get_cancelled ()) {
5        mark (labyrinth, task);
6        if (!inspect_field_for_exit (task))   // inspect all neighbors
7          push (neighbors (task), next_taskpool); // no exit was found
8        else {
9          #pragma omp cancelregion  // an exit was found
10       }
11     }
12     #pragma omp barrier oncancel
13     {
14       #pragma omp exitregion
15     }
16     #pragma omp single
17     {
18       taskpool = next_taskpool; // switch the task pools
19       next_taskpool = NULL;
20     } // implicit barrier
21     #pragma omp onbarriercancel
22     {
23       #pragma omp exitregion
24     }
25   }
26 } // end of parallel region with implicit barrier
```

Figure 6.8: Parallel breadth first search using proposed language constructs

anymore to flush any cancel flags, as they are managed automatically by the OpenMP runtime system. We believe that this change alone will make errors less common in irregular parallel applications.

6.3 Miscellaneous Changes to the OpenMP Specification

In this section, I highlight some enhancements to the OpenMP specification that I found useful during the course of my work on AthenaMP, during my work teaching students, and while contributing in the OpenMP language committee. Many of them have been sketched earlier in this publication and are summarized here to have them all in one place. I have no reference implementations for any of them and not all of them were originally suggested by me, therefore this list is merely an indication of issues I have contributed to in some way.

Controlling the Total Number of Threads: In one of my papers [SL04], I have parallelized a recursive version of the quicksort algorithm using nested parallelism. As one of the results of this paper I found out that there is no easy way to limit the number of threads in an application using nested parallelism in OpenMP. This is necessary, because creating a new active parallel region for each step in the recursion will oversubscribe the machine quickly. Relying on `omp_get_num_threads` is pointless in this case, because it only returns the number of threads in the current team and not the total number of threads in a program. For this reason, I have suggested the introduction of the `omp_get_num_all_threads` function. Its proposed return value is the number of threads created by OpenMP in the program, allowing their effective limitation, especially in but not limited to recursive algorithms.

Lock Initialization with OMP_LOCK_INIT: I have shown in Section 5.2.2 that it would make sense to initialize OpenMP locks not only by using the `omp_init_lock` function, but also with a macro, e. g. `OMP_LOCK_INIT`. This is adapted from POSIX threads, where `PTHREAD_MUTEX_INITIALIZER` can be used to initialize a lock. If this idea was accepted into the standard, it would be possible to initialize static locks using static variable initialization, a facility that is guaranteed to happen only once in OpenMP.

Declaring Static Member Variables threadprivate: In Section 5.2.2 I have shown that there is benefit in allowing static member variables to be declared threadprivate. This would be a very simple change in the specification and has already been implemented in the Intel Compiler.

Flushing Reference Variables not allowed: Although it is not explicitly forbidden in the specification to flush reference variables, most compilers do not allow it either. A very

common use case for reference variables is to pass parameters to functions by reference, either because they need to be changed inside the function or because the object is large and copying it would include a performance penalty. This is a recommended practice in many textbooks about C++. I have hit this problem when implementing the once functionality touched in Section 5.2.2, where the second parameter to the `once` function is passed by reference to const and I would like to flush it inside the function.

The specification only allows to flush pointers and not pointees. This is a problem in my case, because reference variables are most likely implemented as pointers. The OpenMP specification should therefore explicitly allow the special case of flushing reference variables to be more conforming to recommended C++ practices.

Allowing atomic assignments: Right now the specification is very restrictive when it comes to what operations are allowed to be protected by the `atomic` directive. It is allowed to update a storage location, yet it is not allowed to assign a value to it. This leads to the grotesque situation sketched below:

```
int i = 0;

#pragma omp atomic
i += some_function_call (params);   // this is allowed

#pragma omp atomic
i = 0;                              // this is NOT allowed
```

Neither the OpenMP memory model, nor the memory models of the languages that OpenMP is based on (C/C++/Fortran) make any guarantees about what kinds of assignments are guaranteed to be atomic either. Therefore, you can e.g. increment a counter, but you cannot reset it using `atomic`. As soon as a reset is required, the `critical` directive needs to be employed. And since the `critical` directive and the `atomic` directive cannot be intermixed to protect the same storage location, the `atomic` directive cannot be used at all in this case. I have therefore proposed to allow assignments to be protected by the `atomic` directive.

Thread Safety: The OpenMP specification makes very strong guarantees with regards to thread safety (see Section 2.4.9). Unfortunately, most compilers and standard libraries I am aware of currently do not satisfy those guarantees. Since many compiler vendors do not even have control over the libraries used by their compilers, they are hesitant to guarantee anything with regards to thread safety. This has lead to the situation that users count on their libraries and base languages being thread-safe, yet most of them are not in practice, potentially leading to broken programs on current compilers. I have contributed many proposals to change this situation, e.g. by taking some guarantees out of the specification to at least make it consistent with what is actually provided by compilers today.

Iterator Loops in C++: C++ programmers often use iterators in loops. As an example, to add one to all elements in a vector, a C++ programmer will most likely write code similar to this:

```
std::vector::iterator it;
std::vector::iterator first = container.begin();
std::vector::iterator last = container.end();

for (it = first; it < last; ++it)
{
    /* use *it here, e.g.: */
    *it += 1;
}
```

Parallelizing this loop with OpenMP is presently not allowed, except by requiring the programmer to rewrite it into a more C-ish form using simple integers as index variables. Yet, this transformation can be done automatically by the compiler, e.g. by turning the loop into this form:

```
iterator it;
iterator first = container.begin();
iterator last = container.end();

#pragma omp parallel for
for (long omp_i=0; omp_i < std::distance(first, last); ++omp_i)
{
    iterator priv_it = first;
    std::advance (priv_it, omp_i);

    /* use *priv_it here, e.g.: */
    *priv_it += 1;
}
```

It has therefore been proposed to allow the parallelization of iterator loops, at least for random access iterators (although technically, the transformation sketched above works for other iterator types as well, albeit slower). I have contributed to this issue in multiple discussions, e.g. by describing the loop transformation shown above.

Runtime Scheduling: Another issue I have contributed to is the lack of support for changing the scheduling policy for parallel loops at runtime. With this feature, it would be possible to pass a parameter to e.g. the data-parallel patterns described in Section 5.3 and let the user decide which scheduling policy works best for the particular problem. As an example, when the functors supplied to the patterns take a different amount of time to execute depending on their input, dynamic or guided scheduling usually work best.

Currently, the user has to change the parameter inside the library or copy the code into the program, where it can be modified.

Changing the scheduling policy at runtime is only possible with an environment variable in OpenMP at the moment. This feature is mostly used as a debugging aid, because one can only change the schedule of all loops (with a schedule set to `runtime`) at once. Use of environment variables is also not practical for a library. Letting the `schedule` clause take a user-supplied parameter (e. g. a string) would solve this problem and is presently discussed in the OpenMP language committee.

Performing Operations Exactly Once: In Section 5.5.4 I have shown how to make sure a function or a functor is called exactly once. An important usecase for this was initialization of resources. Although the solution I have shown there is workable and makes the life of the programmer easier, it is still not as easy as it could be if the functionality was integrated into the language, e.g. like this:

```
#pragma omp once (A_NAME)
{
    /* put whatever code you want carried out once here */
}
```

This solution requires no scaffolding, no extra functors or functions with parameters and could be implemented by the compiler writers as described in Section 5.5.4. This idea was initially suggested by me in a blog-article [Sue06c].

6.4 Chapter Summary

In this chapter, we have discussed our proposals for changing the OpenMP specification. We started in Section 6.1 with a proposal to solve the busy waiting problem and therefore enable point-to-point synchronization in OpenMP. We suggested two new directives: `yield` and `sleepuntil`. Both enable the programmer to influence the scheduling process, and to put threads to sleep on demand. By using these directives, the need for busy waiting is eliminated. The effectiveness of the approach was demonstrated by benchmarking oversubscribed systems.

In Section 6.2 we have discussed a major problem with parallelizing irregular applications: lacking support for thread cancellation in OpenMP. A workaround and an extension to OpenMP have been suggested, whose main part is the `cancelregion` directive that enables cooperative cancellation. A quick way to exit a parallel region is provided for each thread with the `exitregion` directive. The applicability of both directives has been demonstrated using a breadth-first search example.

Section 6.3 of this chapter highlighted miscellaneous changes to the OpenMP specification I have proposed or worked on in the OpenMP language committee. The list

includes changes for getting the total number of threads, lock initialization, threadprivate static member variables, flushing of reference variables, atomic assignments, thread safety, iterator loops, runtime scheduling and `once` functionality.

Chapter 7

Closing Remarks and Perspectives

This closing chapter of the thesis is divided into two parts. The first one (Section 7.1) summarizes our research, findings and contributions. In the second part (Section 7.2), possible directions for future work are described.

7.1 Thesis Summay

In this thesis, we have explored ways to make parallel programming easier for programmers. A quick overview over the objectives, aims and steps to pursue these aims can be found (one last time) in Figure 7.1.

After a short introduction in Chapter 1, we took a look at foundations for this work in Chapter 2. The chapter contained a broad introduction to the steps involved in parallelizing a program and some common problems while doing so. At the same time, it explained our reasons to build a lot of material in this thesis on top of the parallel programming system OpenMP, which already solves some of the problems sketched there. A short overview over some common parallel programming systems in use today was also provided, along with a short tutorial on OpenMP, which was meant to make the more advanced parts of this thesis more understandable.

Chapter 3 started the main part of this thesis with an evaluation of the current state of the art with regards to parallel programming. It described the results of a survey that has been carried out among more than 250 parallel programmers and above all, highlighted what parallel programming systems and languages are known and used among programmers. C and MPI were the winners for our survey group. Although not statistically sound, the survey allowed us to form eight hypotheses about these topics, which may be proven in the future.

The chapter continued with an evaluation of fifteen frequently made mistakes with OpenMP. The most frequently made mistake for our study was to not properly protect shared variables from concurrent access. These mistakes have been gathered by observing groups of students. We suspect that knowing about the mistakes alone makes them less likely to occur, resulting in better programs that are easier to write in the first place. It has also been shown that current compilers are no big help in diagnosing these errors, but there are tools available (e. g. the Intel Thread Checker) that help at least for some. During

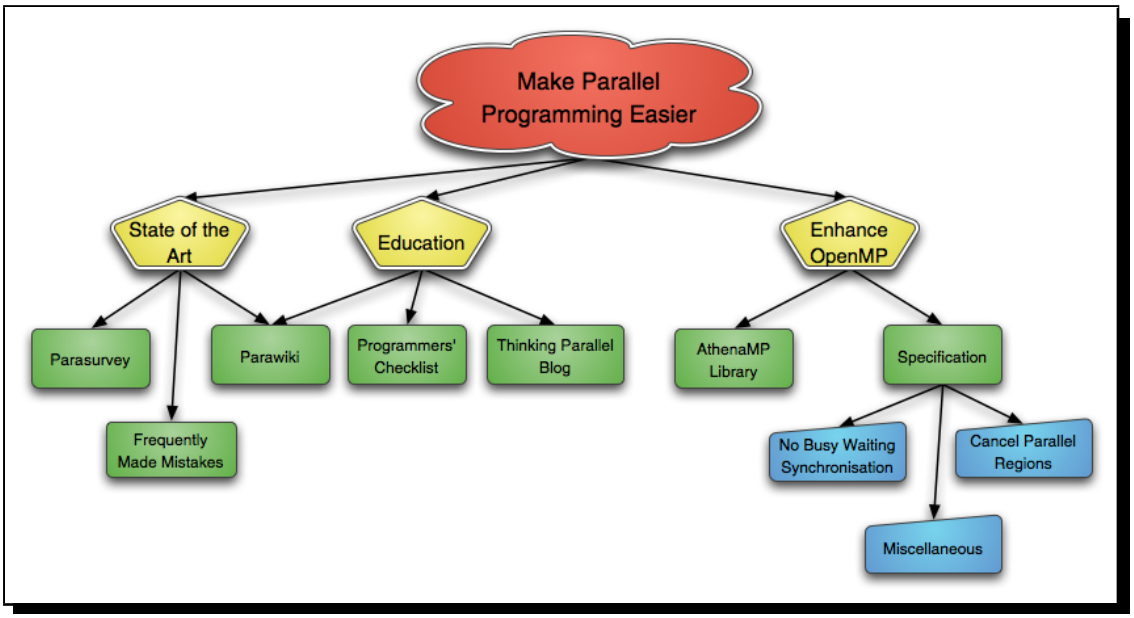


Figure 7.1: Objectives, aims and contributions of this thesis

the course of this work, we frequently returned to these mistakes and presented solutions to them. The chapter closed with a description of the Parawiki, our attempt to gather knowledge and experiences in the field of parallel programming in one single location.

Chapter 4 described our efforts to make parallel programming easier by educating programmers. Its main contribution was a checklist for OpenMP programmers that was built on the mistakes described in the previous chapters and our own experiences from using OpenMP. In this chapter, we also described the Thinking Parallel Weblog, where we are trying to educate programmers about topics related to parallel programming. More than 50.000 readers have read our articles there, which probably makes this resource the one with the biggest impact from this thesis.

Chapter 5 has been the most technical part. It described a library called AthenaMP, in which we have implemented several well-known patterns for parallel programming as so-called generic components. Our contributions for this chapter included research on and implementations of task pools (useful for implementing irregular algorithms), generic locks, deadlock-avoidance functionality in locks, thread-safe singletons, several data-parallel patterns and some more (shown in Figure 7.2). It was shown, how these patterns/components raise the level of abstraction and safe the programmer using them time and errors. Where applicable, performance metrics were also provided.

Chapter 6 was dedicated to enhancing the OpenMP specification on a language level. Some problems we have encountered in earlier chapters could not be solved by a library and therefore we have researched and implemented proposals to change the OpenMP specification, which were described there. Our main contributions included a proposal to avoid the need for busy waiting synchronization and a proposal to enable thread cancel-

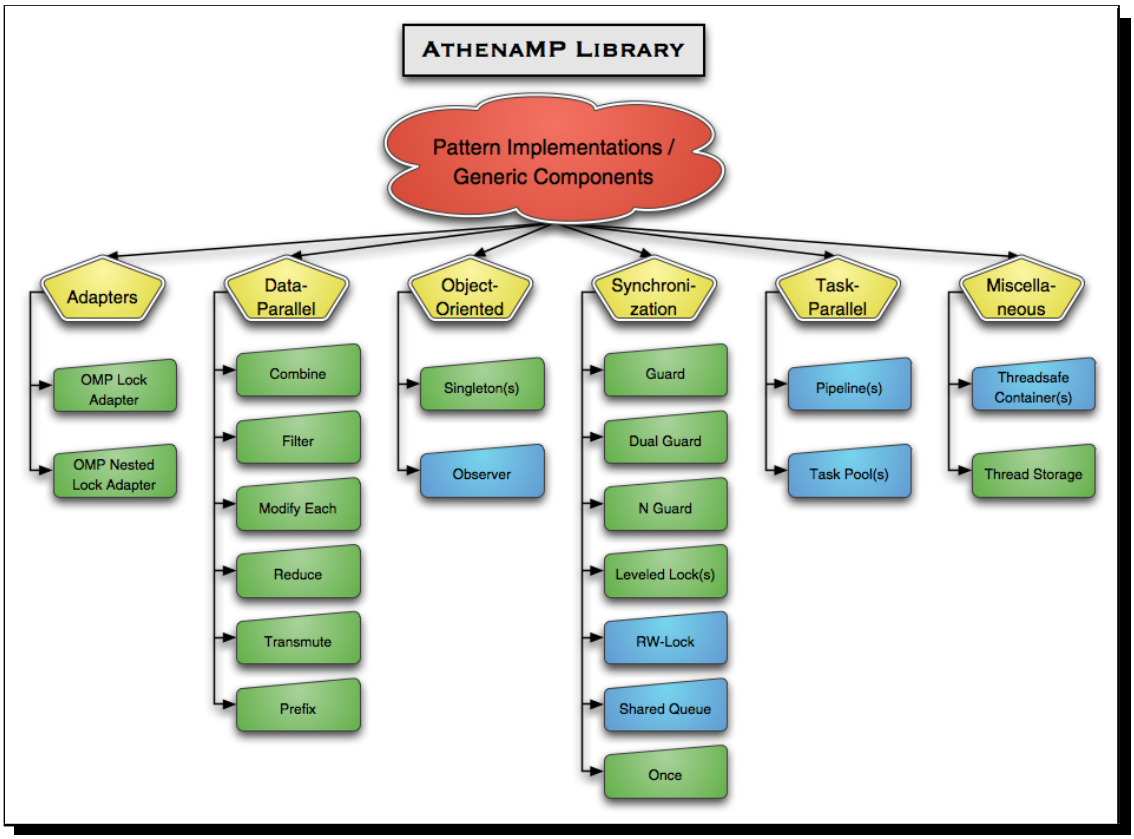


Figure 7.2: The AthenaMP library in a nutshell

lation in OpenMP. The chapter also included proposals for other issues, which are either not as well-tested or not proposed by us originally.

7.2 Future Work

Every part of this thesis can be worked on and extended in future work. The topic of finding the current state of the art was a mere snapshot at this point. Valuable insights could be gained e. g. by repeating the Parasurvey described in Section 3.1 over time, monitoring the changes in used and known systems. Since the survey has merely yielded hypotheses at this point, the next step should be to turn it into a full-blown and statistically useful research study. The Parawiki and the list of frequently made mistakes can easily be extended with more content to make them more useful, the same is true for our programmers' checklist and the Thinking Parallel weblog. Other steps to find out about the state of the art today include monitoring the programs carried out on supercomputers. Education of programmers is a wide area as well, other approaches to research this topic further could be through online courses, tutorials or even e-learning, etc.

The work in Chapter 5 can be extended in many ways. There are a multitude of other parallel programming patterns available that would benefit programmers when turned into generic components. A good example for this are the task pools presented in Section 5.4. An outcome of this section was, that we have not yet found a single task pool that supports both: a large number of tasks as created by e. g. quicksort (where it becomes important to make the task pool operations as fast as possible) and a small number of tasks (where good load balancing is key and the cost of individual task pool operations is not as important). Work on an adaptive task pool that supports both cases has been started already and can be expected to finish in the future. Moreover, the components already present in the library can be tuned for more performance or for ease-of-use.

Extending the library to different parallel programming systems is also a much-needed research direction. An example of how this could be achieved has been presented in Section 5.1 already: the synchronization patterns shown there are not dependent on a specific parallel programming system, but are generic by relying on a common interface. Different patterns already implemented in this thesis could use this technique to become independent of OpenMP, e. g. the task pools shown in Section 5.4 or the ThreadStorage described in Section 5.5.7.

Changing the OpenMP specification is not a finished work as well, as the changes proposed are not yet accepted into the specification - and will probably require some rework before they are, depending on the feedback given by the language committee members.

List of Publications

Parts of this thesis or versions thereof have been published in separate research publications already. The following list contains the publications that I have been involved with during the work on this thesis, either directly or as a contributing author.

[LS06] Claudia Leopold and Michael Suess. Observations on MPI-2 Support for Hybrid Master/Slave Applications in Dynamic and Heterogeneous Environments. In *EuroPVM/MPI 2006, Bonn*, pages 285–292, 2006.

[LSB06] Claudia Leopold, Michael Suess, and Jens Breitbart. Programming for Malleability with Hybrid MPI-2 and OpenMP: Experiences with a Simulation Program for Global Water Prognosis. In *European Conference on Modelling and Simulation, Bonn*, pages 665–670, 2006.

[SL04] Michael Suess and Claudia Leopold. A User's Experience with Parallel Sorting and OpenMP. In *Proceedings of the Sixth European Workshop on OpenMP - EWOMP'04*, October 2004.

[SL05] Michael Suess and Claudia Leopold. Evaluating the State of the Art of parallel programming systems, 2005.

[SL06a] Michael Suess and Claudia Leopold. Common Mistakes in OpenMP and How To Avoid Them. In *Proceedings of the International Workshop on OpenMP - IWOMP'06*, June 2006.

[SL06b] Michael Suess and Claudia Leopold. Implementing Irregular Parallel Algorithms with OpenMP. In *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference*, 2006.

[SL07a] Michael Suess and Claudia Leopold. Generic Locking and Deadlock-Prevention with C++. In *Proceedings of the International Conference on Parallel Computing (PARCO 2007), Juelich, Germany*, 2007.

[SL07b] Michael Suess and Claudia Leopold. Implementing Data-Parallel Patterns for Shared Memory with OpenMP. In *Proceedings of the International Conference on Parallel Computing (PARCO 2007), Juelich, Germany*, 2007.

[SL07c] Michael Suess and Claudia Leopold. Problems, Workarounds and Possible Solutions Implementing the Singleton Pattern with C++ and OpenMP. In *Proceedings of the Third International Workshop on OpenMP - IWOMP 2007*, June 2007.

[SPL07] Michael Suess, Alexander Podlich, and Claudia Leopold. Observations on the Publicity and Usage of Parallel Programming Systems and Languages: A Survey Approach, 2007.

[WSL06] Alexander Wirz, Michael Suess, and Claudia Leopold. A Comparison of Task Pool Variants in OpenMP and a Proposal for a Solution to the Busy Waiting Problem. In *Proceedings of the International Workshop on OpenMP - IWOMP'06*, June 2006.

Bibliography

[ACD+07] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang. A Proposal for Task Parallelism in OpenMP. In *Proceedings of the Third International Workshop on OpenMP - IWOMP 2007*, June 2007.

[AJR+01] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. *Workshop on Languages and Compilers for Parallel Computing*, pages 193–208, 2001.

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001.

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[Bac07] Florian Bachmann. Software Pattern mit C++ und OpenMP, 2007.

[Bal02] J.H. Baldwin. Locking in the Multithreaded FreeBSD Kernel. In *Proceedings of BSDCon*, pages 11–14, 2002.

[BB03] C. Ball and M. Bull. Point-to-Point Synchronisation on Shared Memory Architectures. In *Proceedings of the Fifth European Workshop on OpenMP - EWOMP'03*, September 2003.

[BDG+04] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos Mercurium: a Research Compiler for OpenMP. In *Proceedings of the European Workshop on OpenMP 2004*, October 2004.

[BO01] J. Mark Bull and Darragh O'Neill. A microbenchmark suite for OpenMP 2.0. *SIGARCH Comput. Archit. News*, 29(5):41–48, 2001.

[But97] David R. Butenhof. *Programming with POSIX Threads*. Addison–Wesley, 1997.

[CDK00] Rohit Chandra, Leonardo Dagum, and Dave Kohr. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000.

[CJP07] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using Openmp: Portable Shared Memory Parallel Programming*. Mit Press, 2007.

[Com] Computer Dictionary Online. http://www.computer-dictionary-online.org/.

[Cra00] Eric Crahen. ZThreads. http://zthread.sourceforge.net/, 2000.

[DGLT03] Vassilios V. Dimakopoulos, Alkis Georgopoulos, Elias Leontiadis, and George Tzoumas. OMPi Compiler Homepage. http://www.cs.uoi.gr/~ompi/, 2003.

[dmo] dmoz - open directory project. http://dmoz.org/.

[dS06] Bronis R. de Supinski. [Omp] A memory model question. http://openmp.org/pipermail/omp/2006/000479.html, July 2006.

[Duf06] Joe Duffy. No More Hangs. *MSDN Magazine*, 21(5), 2006.

[DVT00] Eugen Dedu, Stéphane Vialle, and Claude Timsit. Comparison of OpenMP and Classical Multi-Threading Parallelization for Regular and Irregular Algorithms. In Hacène Fouchal and Roger Y. Lee, editors, *Software Engineering Applied to Networking Parallel/Distributed Computing(SNPD)*, pages 53–60. Association for Computer and Information Science, may 2000.

[Fos95] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[Fre] Free On-Line Dictionary of Computing. www.foldoc.org.

[Gei94] Al Geist. *Pvm: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GL81] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive-Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[GNL98] William Gropp, Bill Nitzberg, and Ewing Lusk. *Mpi: The Complete Reference*. MIT Press, 1998.

[GPB+06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.

[Gra03] Grama, A. and Gupta, A. and Karypis G. and Kumar, V. *Introduction to Parallel Computing*. Addison-Wesley New York, 2003.

[HASP00] Dixie Hisley, Gagan Agrawal, Punyam Satyanarayana, and Lori Pollock. Porting and performance evaluation of irregular codes using OpenMP. *Concurrency: Practice and Experience*, 2000.

[HdS05] Jay P. Hoeflinger and Bronis R. de Supinski. The OpenMP Memory Model. In *Proceedings of the First International Workshop on OpenMP - IWOMP 2005*, June 2005.

[Hoa62] C.A.R. Hoare. Quicksort. *The Computer Journal*, 5:10–15, 1962.

[Int07] Intel. Threading Building Blocks. http://www.intel.com/cd/software/products/asmo-na/eng/threading/294797.htm, 2007.

[JG97] E. Johnson and D. Gannon. HPC++: experiments with the parallel standard template library. *Proceedings of the 11th international conference on Supercomputing*, pages 124–131, 1997.

[Kaw] Guy Kawasaki. Bona tempora volvantur. http://blog.guykawasaki.com/.

[Kem01] William E. Kempf. The Boost.Threads library. http://www.boost.org/doc/html/threads.html, 2001.

[KL07] Bjoern Knafla and Claudia Leopold. Parallelizing a Real-Time Steering Simulation for Computer Games with OpenMP. In *Proceedings of the International Conference on Parallel Computing (PARCO 2007), Juelich, Germany*, 2007.

[KR04] Matthias Korch and Thomas Rauber. A Comparison of Task Pools for Dynamic Load Balancing of Irregular Algorithms. *Concurrency and Computation: Practice and Experience*, 16(1):1–47, 2004.

[Lab03] Lawrence Livermore National Laboratory. OpenMP. http://www.llnl.gov/computing/tutorials/openMP/, 2003.

[LCTaM04] Yuan Lin, Nawal Copty, Christian Terboven, and Dieter an Mey. Automatic Scoping of Variables in Parallel Regions of an OpenMP Program. In *Proceedings of the Workshop on OpenMP Applications & Tools - WOMPAT 2004*, May 2004.

[Leo01b] Claudia Leopold. *Parallel And Distributed Computing: A Survey of Models, Paradigms, and Approaches*. John Wiley & Sons, 2001.

[LHZ98] Honghui Lu, Charlie Hu, and Willy Zwaenepoel. OpenMP on Networks of Workstations. In *Proc. of Supercomputing'98*, 1998.

[Lin05] Yuan Lin. Reducing the Complexity of Parallel Programming. http://blogs.sun.com/roller/page/yuanlin, 2005.

[MA04] Scott Meyers and Andrei Alexandrescu. C++ and the Perils of Double-Checked Locking - Part 1. *Dr. Dobb's Journal*, pages 46–49, July 2004.

[Mat03] Timothy G. Mattson. How good is OpenMP. *Scientific Proramming*, 11:81–93, 2003.

[Mey05] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Professional, 3. edition, 2005.

[MSM04] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming (Software Patterns Series)*. Addison-Wesley Professional, 2004.

[NPA01] Dimitrios S. Nikolopoulos, Constantine D. Polychronopoulos, and Eduard Ayguade. Scaling irregular parallel codes with minimal programming effort. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)*. ACM Press, 2001.

[Ope05] OpenMP Architecture Review Board. OpenMP Specifications. http://www.openmp.org/specs, 2005.

[Qui03] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Science/Engineering/Math, 2003.

[Rey04] Craig W. Reynolds. OpenSteer website. http://opensteer.sourceforge.net, 2004.

[Sad] Yariv Sadan. Yariv's Blog. http://yarivsblog.com/.

[SHPT99] Sanjiv Shah, Grant Haab, Paul Petersen, and Joe Throop. Flexible Control Structures for Parallelism in OpenMP. In *Proceedings of the First European Workshop on OpenMP - EWOMP'99*, September 1999.

[Sin] Eric Sink. Eric.Weblog(). http://software.ericsink.com/.

[SL] Michael Suess and Claudia Leopold. Evaluating the state of the art of parallel programming systems, text = Technical Report KIS No. 1 / 2005, University of Kassel, year = 2005, url = http://opus.uni-kassel.de/handle/urn:nbn:de:hebis:34-20050712196,.

[SL04] Michael Suess and Claudia Leopold. A User's Experience with Parallel Sorting and OpenMP. In *Proceedings of the Sixth European Workshop on OpenMP - EWOMP'04*, October 2004.

[SL06a] Michael Suess and Claudia Leopold. Common Mistakes in OpenMP and How To Avoid Them. In *Proceedings of the International Workshop on OpenMP - IWOMP'06*, June 2006.

[SL06b] Michael Suess and Claudia Leopold. Implementing Irregular Parallel Algorithms with OpenMP. In *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference*, 2006.

[SL07c] Michael Suess and Claudia Leopold. Implementing Data-Parallel Patterns for Shared Memory with OpenMP. In *Proceedings of the International Conference on Parallel Computing (PARCO 2007), Juelich, Germany*, 2007.

[SL07d] Michael Suess and Claudia Leopold. Problems, Workarounds and Possible Solutions Implementing the Singleton Pattern with C++ and OpenMP. In *Proceedings of the Third International Workshop on OpenMP - IWOMP 2007*, June 2007.

[Sod05] Angela C. Sodan. Message-Passing and Shared-Data Programming Models - Wish vs. Reality. In *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications (HPCS'05)*, pages 131–139, Washington, DC, USA, 2005. IEEE Computer Society.

[SPL07] Michael Suess, Alexander Podlich, and Claudia Leopold. Observations on the Publicity and Usage of Parallel Programming Systems and Languages: A Survey Approach, 2007.

[Spo00] Joel Spolsky. Joel on Software. http://www.joelonsoftware.com, 2000.

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2 – Networked and Concurrent Objects*. John Wiley and Sons, 2000.

[Str97] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[Sue] Michael Suess. Thinking Parallel. http://www.thinkingparallel.com.

[Sue05a] Michael Suess. Parallel Programming Survey. http://www.plm.eecs.uni-kassel.de/parasurvey/, February 2005.

[Sue06a] Michael Suess. A Short Guide to Mastering Thread-Safety. http://www.thinkingparallel.com/2006/10/15/a-short-guide-to-mastering-thread-safety/, 2006.

[Sue06b] Michael Suess. A Vision for an OpenMP Pattern Library in C++. http://www.thinkingparallel.com/2006/11/03/a-vision-for-an-openmp-pattern-library-in-c/, November 2006.

[Sue06c] Michael Suess. How to do it ONCE in OpenMP. http://www.thinkingparallel.com/2006/09/20/how-to-do-it-once-in-openmp/, 2006.

[Sue07a] Michael Suess. AthenaMP. http://www.athenamp.org/, 2007.

[Sue07c] Michael Suess. Please Don't Rely on Memory Barriers for Synchronization! http://www.thinkingparallel.com/2007/02/19/please-dont-rely-on-memory-barriers-for-synchronization/, 2007.

[Sun99] Sun Microsystems. Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated. http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html, 1999.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2. edition, 2001.

[Thea] The Language Guide. http://www.engin.umd.umich.edu/CIS/course.des/cis400/.

[Theb] The Parawiki. http://parawiki.plm.eecs.uni-kassel.de.

[Tro07] Trolltech. QT Concurrent. http://labs.trolltech.com/page/Projects/Threads/QtConcurrent, 2007.

[Wik] Wikipedia The Free Encyclopedia. http://www.wikipedia.org/.

[Wol96] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.

[WSL06] Alexander Wirz, Michael Suess, and Claudia Leopold. A Comparison of Task Pool Variants in OpenMP and a Proposal for a Solution to the Busy Waiting Problem. In *Proceedings of the International Workshop on OpenMP - IWOMP'06*, June 2006.

[Yeg04] Steve Yegge. Singleton Considered Stupid. http://steve.yegge.googlepages.com/singleton-considered-stupid, 2004.

Index

Statement

Erklärung

Hiermit versichere ich, dass ich die vorliegende Dissertation selbständig und ohne unerlaubte Hilfe angefertigt und andere als die in der Dissertation angegebenen Hilfsmittel nicht benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen sind, habe ich als solche kenntlich gemacht. Kein Teil dieser Arbeit ist in einem anderen Promotions- oder Habilitationsverfahren verwendet worden.

Kassel, September 19, 2007

Michael Süß