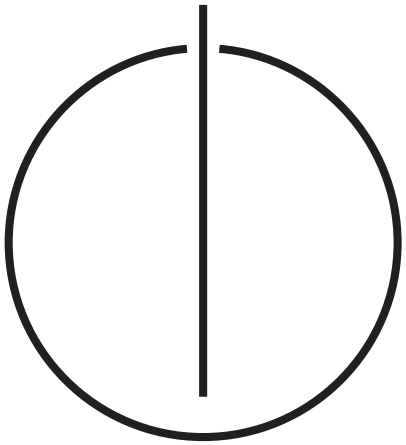
Technische Universität München

Fakultät für Informatik



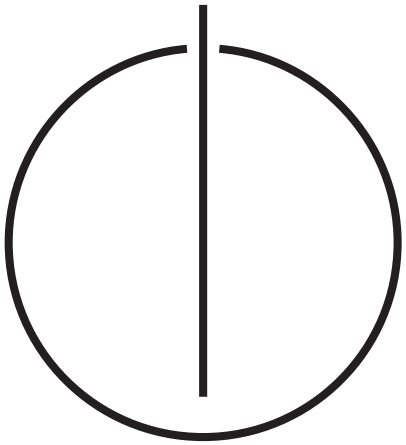
Master's Thesis in Informatik

Performance Prediction to Assist Parallelization

Robert Guder

Technische Universität München

Fakultät für Informatik



Master's Thesis in Informatik

Performance Prediction to Assist Parallelization

Performanzvorhersage zur Unterstützung von Parallelisierung

Author: Robert Guder

Supervisor: Prof. Dr. Michael Gerndt

Advisor: Andreas Wilhelm

Submission Date: 28.01.2016

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching, January 28, 2016 Robert Guder

Abstract

In this Master's thesis we present an annotation based tool to predict the possible parallel speedup and the scalability of sequential programs. Program parallelization is sophisticated, time-consuming and error-prone. Hence, this tool assists architects and developers to decide whether it is worth the effort to parallelize existing sequential programs. In this context, the effort to estimate the parallel speedup is usually a fraction of the effort to parallelize the complete program.

The user has to annotate parallel executable tasks, loops and critical sections in a sequential source code. This source code is profiled to gather timing information, which is used to create a task graph. To simulate the parallel behaviour of the program, the task graph is traversed and each task is scheduled using static scheduling. Additionally, the overhead is considered in the parallel simulation to achieve accurate prediction results. For this purpose, we present a sophisticated approach to determine the overhead accurately in little time.

Experimental results demonstrate the accuracy of the tool, which is able to consider workload imbalance, nested parallelism and the synchronization of critical sections. The parallel speedup of all presented examples are predicted with an maximum inaccuracy of 46.6%. Three out of four of the examples are predicted with an averaged inaccuracy of 2.1%.

Contents

1 Introduction

In 1965 Moore's law was formulated by Gordon Moore. It claims that the number of transistors on the same area of a chip doubles every 12 to 24 months. Up to the mid 2000s the increasing number of transistors led directly to an increasing clock frequency. Consequently, developers could rely on a performance improvement of their programs without changing anything. However, due to several reasons, like heat generation, a clock frequency limit of about 4 GHz per single processor was reached. Now, developers have to use the advantages of multi-core processors to further reduce their program execution time. For this reason, parallelization of programs becomes more and more important.

Parallel programming, however, brings many challenges. Due to the involved interleaving and synchronization complexity, it is quite subtle and error-prone. This makes it difficult for developers to write code efficiently. Moreover, developers are not used to parallel programming. As most programming languages have predominantly sequential syntax, programmers usually do not think in parallel. In addition, tools which support efficient sequential coding, like code editors, debuggers and tools for code analysis and verification were subject of research for many decades. In contrast to that, the development of tools supporting parallel programming is more complex and subject of research for only a short time. Hence, less useful tools are available for parallel programming. Consequently, writing parallel programs is very time consuming.

In contrast to write completely new parallel programs, it is often more time consuming to parallelize legacy code. Dealing with legacy code leads to a diversity of problems. For example, the whole program architecture was designed for sequential execution. Thus, instead of writing a parallel program from scratch, many parts of the source code have to be transformed to parallel code. This is a complex task, because usually the source code was written by someone else which makes it hard for the developers to understand all parts of the code. Moreover, these former developers, who were involved writing the source code, usually do no longer work in the same team or even the same company and cannot be asked for advice. This combination leads to increased complexity in parallelizing legacy code.

As a consequence, managers and developers are interested in a quick estimation of the parallel potential of existing sequential source code before spending much effort in parallelizing the code. Simple methods to approximate the parallel speedup of programs, like Amdahl's law [1], use the parallel and sequential execution time and the number of cores for the prediction. However, they do not consider the underlying memory architecture and the overhead of parallel execution to estimate the speedup in real parallel systems. Consequently, the prediction results are inaccurate and cannot be used to estimate the risk of parallelization for critical projects. To achieve more accurate results, a parallel

performance prediction tool which considers these additional factors is helpful, like it is developed in this thesis. By using the prediction of a maximum, realistic speedup, possible risks can be estimated and programmer can decide which parts of the program are worth the effort of parallelization.

Although there is a high demand for such prediction tools, only a limited number of tools exist. All of them have some drawbacks, however. Kismet [25], for example, which is built upon a hierarchical critical path analysis, predicts the speedup based on a very small granularity. As a developer is not able to parallelize on such a granularity the tool is mainly developed for automatic parallelization. Two other tools, Parallel Prophet [27] and Suitability in Intel Advisor [24], are using a similar approach as the tool presented in this thesis. However, Intel Advisor is a proprietary tool and neither the source code nor detailed information about used prediction techniques are published. In contrast to that, some scientific work [27] [26] regarding Parallel Prophet exists. This is interesting from a scientific point of view, however, there is no use for a developer, as neither the source code nor a executable file of this tool is available.

In this thesis we present a tool to estimate the speedup of coarse grained parallelization. The tool has high usability so that the user needs to invest only a fraction of the effort of a real parallelization. For this reason, six annotations are introduced to express parallel sections, tasks, critical sections and expected synchronization behavior. The user can insert these annotations to mark parts of the code which he would like to parallelize. Afterwards, the annotated code is executed and the results are used to create a task graph, which is necessary for the scheduling algorithm of the tool. Finally, the total speedup of the program is calculated considering parallel overhead. The goal of this process is a good balance between high accuracy, moderate execution time, and low memory overhead of the prediction tool. In the current version the tool can predict the speedup of parallel loops and tasks and it supports workload imbalance, lock synchronization, nested parallelism, and recursive parallelism.

To get an idea how the parallel performance prediction tool works, a small code snippet is presented in Code 1.1. The code is doing some dummy calculation within a for loop. In the third and fourth iteration of the loop, a critical read and write to the same variable is made. The developer would like to know the parallel potential of this code. Hence, the developer inserts appropriate annotations which can be seen in Code 1.2. By inserting PAR_SEC_BEGIN() and PAR_SEC_END(), a parallel section is defined. Within a parallel section every task can be executed in parallel. Every loop iteration is marked as a single task and the critical read and write is synchronized with PAR_LOCK_BEGIN() and PAR_LOCK_END().

Using this information, the prediction tool measures the time between two annotations and creates a task graph as shown in Figure 1.1. The beginning and the end of a section is represented with boxes, tasks are represented as ellipses, sequential parts as octagons and locks as diamonds. Moreover, the information is used to estimate the overhead being considered by the scheduling. For the estimation the parallel execution time of tasks, the number of cores and the number of loop iterations is necessary. The overhead is added

Code 1.1: *Code snippet including a critical region. The parallel potential of this code should be predicted.*

```
for (int k = 0; k < 4; ++k)
{
   //do some calculations
   if(k > 1)
   {
      //read and write. Critical
          access.
   }
}
```

Code 1.2: *Annotations are inserted at the parts of the code which may run in parallel.*

```
PAR_SEC_BEGIN("sec", true);
for (int k = 0; k < 4; ++k)
{
   PAR_TASK_BEGIN("task");
   //do some calculations
   if(k > 1)
   {
      PAR_LOCK_BEGIN(0);
      //read and write. Critical
          access.
      PAR_LOCK_END(0);
   }
   PAR_TASK_END();
}
PAR_SEC_END();
```

to the execution time of the according task in the task graph. In the following step, the task graph is used to calculate the scheduling. We only model the execution times of sequential regions and locks. Hence, Figure 1.2 depicts a possible scheduling using two cores. In the figure, overhead is marked red and critical sections are marked gray. The critical sections may not be executed at the same time. Finally, the predicted speedup is presented to the user. In this example, the prediction tool estimates a speedup of 1,5. To check the scalability of the program, the whole process is repeated using an increasing number of cores.

The exact functionality of the tool is described in the following chapters. The thesis starts with an introduction to parallel programming in Chapter 2. Relevant details of the hardware are explained, and parallel programming models are presented. Moreover, an introduction to performance theory is given including the presentation of different analytic performance prediction models. In addition, basic terms, definitions and concepts of parallel programming are described. Some related work is mentioned in Chapter 3. In this chapter, existing prediction tools are introduced and existing scientific work is named. Following, Chapter 4 describes the developed tool in detail. The utilized model is introduced and the exact meaning of the annotations is defined. Furthermore, the theory and implementation of the used profiling approach, the task graph creation, the overhead determination and the scheduling is explained. After the whole tool chain is presented, the final results are shown in Chapter 5. The features of the prediction tool are evaluated by different examples. These examples are described in detail and the prediction results are presented. Additionally, an overview of the execution time of the prediction tool is given. Finally the thesis is summarized in Chapter 6 and an outlook is given.

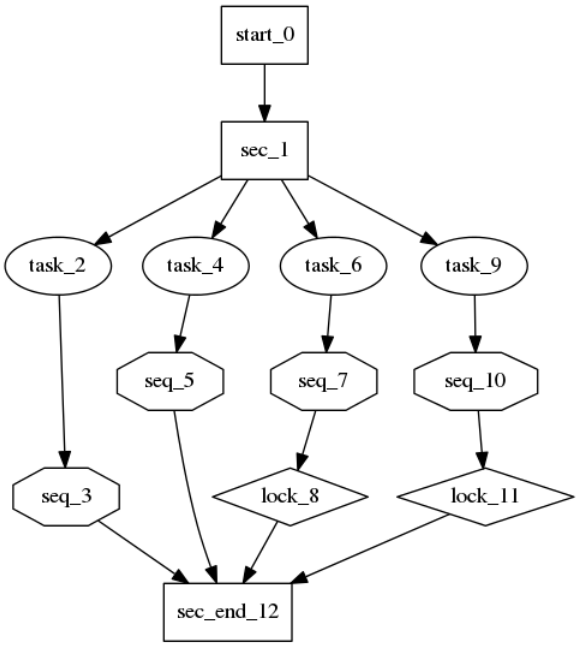


Figure 1.1: *This task graph is created out of Code 1.2. The task graph is used for the scheduling in the prediction tool.*

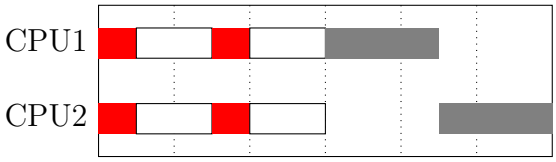


Figure 1.2: *Possible schedule of the four tasks of the parallelized code. The according task graph is shown in Figure 1.1. The overhead in the scheduling is marked red. Critical sections are marked gray and may not be executed at the same time.*

2 Parallel Programming

As already mentioned in the introduction, the limit of single-core processor performance is reached since the mid 2000's. Since this event, new hardware trends occurred and a common technique to further improve program performance is the development of parallel programs. For parallel programming in general, and for the prediction of parallel performance in particular, a basic knowledge about this topic is necessary. This chapter gives a broad overview about these basics. First of all, different approaches of parallelism in the hardware are introduced. Afterwards, concepts of parallel programming and programming models are explained. Additionally, an overview of performance theory is given. This includes the description of some metrics, as well as the presentation of well known performance prediction models from Gene Amdahl, John Gustafson and the work-span model. Parallel Programming, however, has some difficulties. Consequently, some pitfalls of parallel programming are introduced at the end of this chapter, like race conditions, critical sections, load imbalance and parallel overhead.

2.1 Hardware

The hardware underlies every computation in a computer system. It allows the parallel execution of instructions and is a complex system of different units holding and manipulating data. Due to this complexity, parallel execution of programs behave differently on different underlying hardware. However, it is neither possible, nor necessary to know the details of every single system to parallelize software. Nevertheless, some basics have to be introduced to the reader to understand the proposed prediction model in this thesis in detail.

2.1.1 Memory Model

First of all, the memory model in computer systems is presented. Every system has to hold data. To store a big amount of data, cheap memory is used. However, this memory - usually the harddisk of a system - is slow. Consequently, the data access is the performance bottleneck and the processor speed to manipulate data cannot be exploited. Using faster memory would solve this problem, however, it is to expensive to store Gigabytes of data. As a result, a hierarchical memory model was developed.

It follows the idea that the fastest memory units - registers - are placed close to the functional units of the CPU and can hold only a very small amount of data. The next level of the memory hierarchy is the cache. It has slightly more capacity, however, it is slower than the registers. Usually computers have multiple levels of cache, each level having more capacity and being slower than the level above. These caches are named L1

cache and L2 cache. Most of the systems have an additional L3 cache, which is shared across all cores in a CPU. The two bottom most levels of the memory hierarchy are the main memory and the disk storage. According to [38], access to the main memory is typically ten times slower than the acces to the last level cache, however, it has more than 1000 times bigger capacity. Figure 2.1 illustrates the relationship between speed and memory size of the different kinds of memories.

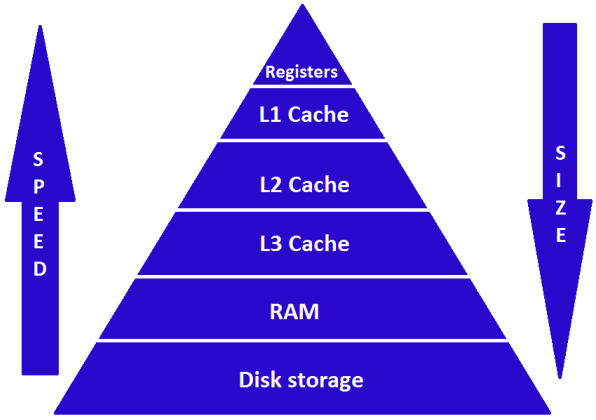


Figure 2.1: *Illustration of the memory hierarchy. The fastest and smallest memory is at the top of the hierarchy, whereas the slowest memory with the most capacity is at its bottom. Additionally to lower levels, frequently used data is stored in higher levels of the memory hierarchy.*

To take advantage of this system, a coherent caching strategy is used. In general, the complete data is saved at the bottom most level of the memory hierarchy, where memory is cheap. To speed up the access, data which is frequently used, or data which is used in near future with high probability, is additionally saved in higher hierarchy levels.

2.1.2 Classification of Computer Architectures

Independent of memory hierarchy, computer architectures can be classified. In the context of this thesis, computer architectures are classified according to Flynn's Taxonomy [14]. Using this taxonomy, it is possible to characterize parallelism. It considers how the control flow and the data management is implemented in the architectures. In general, there are four categories, which are summarized in Figure 2.2. **Multiple Instruction, Single Data (MISD)** is more of a theoretical nature and not used in real systems. Consequently, it is not described in detail here. This is different with **Single Instruction, Single Data (SISD)**, which represents the standard von-Neumann-Architecture and was used in non-parallel processors in the last decades. Systems classified to this category usually consist of one processor manipulating a single data element after the other sequentially. In contrast to that, parallelism occurs in **Single Instruction, Multiple Data (SIMD)** architectures. In SIMD architectures, a single instruction is able to execute operations on multiple data elements at the same time. For example, vector processing units are typical representatives of SIMD systems. The fourth category in Flynn's taxonomy is **Multiple Instruction, Multiple Data (MIMD)**. Among others, common multi-processors are

classified to this category, because several instructions can manipulate multiple data at the same time.

- **SISD:** Single Instruction Single Data - describes the Von-Neumann-Architecture
- **SIMD:** Single Instruction Multiple Data - e.g. vector processing units
- **MISD:** Multiple Instruction Single Data - more a theoretical descritpion
- **MIMD:** Multiple Instruction Multiple Data – e.g. multi-core CPUs

Figure 2.2: *Categories of Flynn's taxonomy.*

2.1.3 Instruction Level Parallelism

Parallelism - the process of speeding up something by doing two or more things simultaneously - in computer programs can be realized using multiple processors. It can also occur in single-processor systems on the instruction level, however. This section presents different kinds of instruction level parallelism.

One kind of instruction level parallelism is **instruction pipelining**. The instruction pipeline consists of the following four steps:

- **Instruction Fetch (IF):** The instruction is loaded from the memory.
- **Instruction Decoding (ID):** The instruction is decoded and required data is loaded from the memory and the registers.
- **Execution (EX):** The instruction is executed.
- **Write Back (WB):** The result is written back to the memory or to the registers.

Figure 2.3 illustrates the sequential and parallel execution of instructions. As long as there is no dependency among single instructions, execution of instructions can overlap (see Figure 2.3a). Compared to sequential execution (see Figure 2.3b), the throughput of parallel execution is obviously higher.

A technical extension of instruction pipelining is realized in **superscalar CPUs**. Instead of only being able to execute one IF, one ID and so on at the same time, superscalar CPUs are capable of executing two or more instructions of the same type concurrently. This is possible, because processors usually have several functional units of the same type, like Arithmetical Logical Units (ALU) or Floating Point Units (FPU).

Vector processors are another kind of instruction level parallelism. They enable the use of instruction sets which operate on a one-dimensional array called vectors. Thus, the execution of a single instruction on multiple data at the same time is possible. Acccording to Flynn's taxonomy, this is a typical SIMD architecture. For example, Intel's Advanced Vector Extension (AVX) [35] provides 256 Bit registers in which bytes and words can be stored. It allows the execution of the same operation on the complete data of a single register.

Instruction Number	Pipeline							
1	IF	ID	EX	WB				
2		IF	ID	EX	WB			
3			IF	ID	EX	WB		
4				IF	ID	EX	WB	
5					IF	ID	EX	WB
Clock Cycle	1	2	3	4	5	6	7	8

(a) *Parallel execution of complete instructions. The following instruction starts, before the last instruction has finished.*

Instruction Number	Pipeline															
1	IF	ID	EX	WB												
2					IF	ID	EX	WB								
3									IF	ID	EX	WB				
4													IF	ID	EX	WB
Clock Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

(b) *Sequential execution of complete instructions. The following instruction starts after the last instruction has finished.*

Figure 2.3: *Instruction pipelining.*

2.1.4 Multi-Processor Systems

Besides parallelism in a single-processor architecture, parallel execution is typically realized using more than one processor for computations. Variants of this kind of parallelism are multi-core systems and multi-processor systems. To explain the difference, Flynn's taxonomy presented in Section 2.1.2 has to be expanded to enable a more precise classification of MIMD systems. Both, multi-processor systems, as well as multi-core systems are classified as MIMD systems. Within MIMD systems it can be further distinguished between shared memory systems and distributed memory systems, as introduced in Figure 2.5.

A multi-core architecture uses **shared memory**. In a multi-core architecture usually many cores are included in a single processor. Each of these cores makes its own calculation and has its own L1- and L2-cache. The L3-cache and the main memory, however, is usually shared. This means, different processor cores (processing units) have access to the same shared, local memory. Figure 2.4 illustrates a typical multi-core-architecture.

Shared memory systems can be subclassified according to the layout of the memory. Systems which share memory uniformly are called uniform memory access (UMA) systems. This means, the memory access time is the same for each core. If the access time is not uniform the system is called a not-uniform memory access (NUMA) system. In this system the memory access time depends on the location of the memory relative to the processing unit. The closer a memory is located to the processor, the faster is the access.

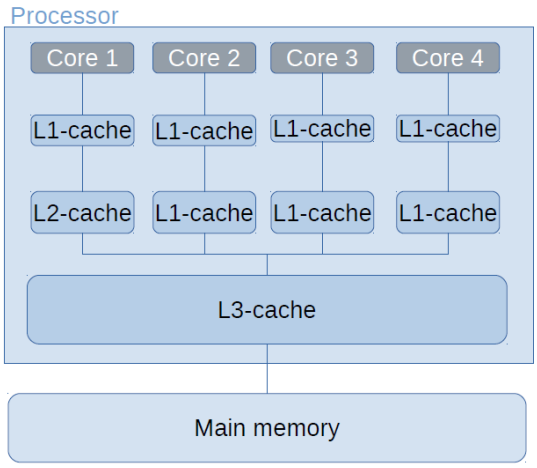


Figure 2.4: *A typical multi-core-architecture. Here, four cores are integrated in a single processor. Each core has its own L1 and L2-cache and shares the L3-cache and the main memory.*

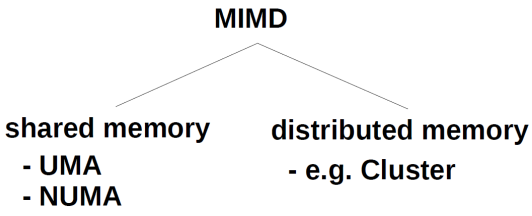


Figure 2.5: *Detailed classification of MIMD architectures.*

In contrast to multi-core processors, multi-processor systems are distributed in a network. Each processor has its own directly accessible local memory. Regarding the network as a whole, however, we talk about **distributed memory**. In parallel applications, data of neighboured processors have to be exchanged using network communication. A typical example of a distributed system is a cluster. Moreover, any multi-processor system can consist of several multi-core processors.

Figure 2.5 gives an overview about the detailed classification of MIMD architectures, including shared and distributed memory systems.

2.2 Concepts of Parallel Programming

Parallel program execution requires appropriate hardware. But moreover, the executed software must be parallelized. For this reason some fundamental concepts of parallel programming are introduced in this section.

First of all, some terms are defined. A **task** is a piece of work. It can be executed sequentially or in parallel. **Work** is the general term for all operations a processor executes to follow a specific goal. A processor can execute **processes**, which can consist of different **threads**. Processes are instances of programs running on the computer. Threads, in general, are entities, which execute tasks. The difference between threads and processes is that all threads belonging to a process share the same data of the process, whereas a process has no access to the data of another process. This data has to be distributed using inter process communication. Threads can be distinguished between **hardware threads** and **software threads**. A hardware thread is a hardware entity, which can execute a program by itself. In contrast to that, a software thread is only a virtual hardware thread. The number of hardware threads depends on the number of existing cores. Each core can

run at least on hardware thread. Depending on the CPU architecture, a core can run multiple hardware threads. However, more software threads than hardware threads can be created. If the number of software threads exceeds the number of hardware threads, software threads are mapped to hardware threads by the operating system. In this thesis it is assumed that every core of a processor has exactly one hardware thread and every hardware thread consists of exactly one software thread. Consequently, core and thread is used in the same context in the following.

A concept to parallelize a sequential program is to decompose it in tasks. These tasks are assigned to threads. If two or more tasks are not dependent on each other, they can be executed concurrently.

To determine concurrency in sequential programs different strategies exist. Most commonly it is tried to indentify **data parallelism** and **functional parallelism**. In data parallelism the same operation is is executed on different items of data simultaneously. In contrast to that, functional parallelism follows the approach to execute different functional blocks - mostly represented by single functions - in parallel. Besides, it can happen that further tasks are created during the execution of an existing task. This may be the case in recursive function calls and is usually known as **nested parallelism**.

2.3 Parallel Programming Models

Program parallelization is a sophisticated task and it is hard to handle this on fine granularity for larger programs. For this reason, some parallel programming models and libraries exist to support the programmer. This section introduces some basic programming models and corresponding examples. Moreover, some clear definitions are presented, because the names used for programming models differ in literature.

According to [46] it can be generally distinguished between implicit and explicit parallelism. The automatic parallelization of sequential source code by parallelizing compilers or interpreter is called implicit parallelism. Hence, the developer has no direct influence on the parallel program created by the compiler. In contrast to that, human support is possible and even necessary using explicit parallelism. The developer is responsible for the majority of the parallelization part and has strong influence on the parallel performance of the program.

Having clarified the difference between implicit and explicit parallelism, the classification of programming models can be defined. Most standard programming languages like C, C++, Fortran or Java are classified into the **sequential model**, because these programming languages are originally designed for sequential programming but can be parallelized using parallelizing compilers. Automatic parallelizing compilers, like Polaris [3], SUIF [6], RawCC [32] an the auto-parallelizing feature of the Intel C++ Compiler (ICC) have different approaches to parallelize existing program code. Nevertheless, they all have in common that the developer usually need not spend much effort in the parallelization phase. Compilers are able to do all the steps of the parallelization. This includes the

detection of parallelizable parts, the transformation of code and finally some parallel runtime management. Usually, they do a lot of optimization work on the instruction level, like vectorization and optimizing the instruction pipeline. In addition, such parallelizing compilers are able to distribute calculations and tasks to multiple threads. For this purpose, they statically analyze the code to detect dependencies. If they can guarantee that two or more tasks are not dependent on each other, the compilers parallelize these tasks. In contrast to humans, a compiler is not error-prone. The parallelization done with a compiler, however, is restricted to loops [18] and (as already mentioned before) parallelization on instruction level. Moreover, the auto parallelization mode of the ICC for example, only detects a small percentage of the parallelizable tasks even in a simple parallelizable code like the Rodinia [11] benchmark suite. A well educated human developer is able to find roughly six times as many parallelizable tasks [16]. Hence, for efficient parallelization human support is necessary.

Using language extensions and a special high-level API, the developer can parallelize programs efficiently in most favored and well known programming languages. Such APIs and language extensions can be distinguished between a **shared memory programming model** and **message passing**. The difference between shared memory systems and distributed memory systems is already explained in Section 2.1.4.

For the parallelization of programs for distributed systems, data and instructions have to be synchronized among the different processes running on different computers, using messages over the network. The Message Passing Interface (MPI) is the most common API, which supports the programmer to exchange synchronization messages easily.

Code 2.1: *Selection of some directives in OpenMP.*

```
#include "omp.h"

//a group of threads is spawned
#pragma omp parallel
{
   //loop iterations are divided dynamically and are assigned to the threads
   #pragma omp for schedule(dynamic, 1)
      for (int i = 0; i < 1000; i++)
      {
         //some calculations
      }
}
```

Other language extensions and APIs like Cilk++ [33], Intel Threading Building Blocks (TBB) [41] and OpenMP [40] do not offer functions for communication among physically distributed processors. They are typical examples for shared memory programming models and execute several pieces of work in parallel in different threads on a shared memory system. OpenMP defines compiler directives to manage parallel execution. Using special directives, the developer can explicitly parallelize applications. OpenMP is mainly used to distribute the execution of loops to different threads. For this purpose, the developer

can choose different scheduling strategies, as well as special synchronization directives. Additionally, newer versions of OpenMP also support tasks, which can be executed concurrently in parallel sections. Moreover, critical sections can be defined and some further attributes can declare data as shared or private. A critical section is a part of the program, which cannot be executed concurrently and has to be protected. Code 2.1 shows a small snippet how OpenMP directives are used in C++ programs. To enable the use of the OpenMP API, the library has to be included. With "#pragma omp parallel" a group of OpenMP threads is spawned. The loop presented in the code snippet is parallelized using "#pragma omp for". In contrast to OpenMP, TBB and Cilk do not use directives to express parallelism. TBB is a portable C++ template library, which provides additionally concurrent data structures and parallel algorithms. Moreover, no special compiler support is required for TBB. Cilk++ introduces new C++ keywords to express parallel code. Additionally, Cilk++ supports array notations to express data parallelism in a natural way.

Using message passing and shared memory models, the developer has to detect dependencies and parallel sections on his own. Sometimes, additional code transformations have to be made to resolve dependencies. The developer can mark parallel sections, scheduling strategies, common data and so on, using the provided interfaces of the presented language extensions. Afterwards, the runtime system of the language extension is responsible for the parallel execution. Consequently, these parallel programming models are examples of explicit parallelism.

2.4 Performance Theory

In the last sections basic knowledge necessary for program parallelization is summarized. The focus of this thesis is the prediction of the parallel performance of a sequential program. In this connection, we use the term performance as the fastest possible execution time of a program using available (hardware) resources. For the process of performance prediction, some metrics are defined in this section. Additionally, several existing analytic models are described which estimate the parallel performance of programs under the assumption of ideal machines, like Amdahl's law, Gustafson's law and the work-span model.

2.4.1 Metrics

First of all, some metrics are defined. In parallel programming, programmers usually are interested in the speedup and the scalability of their programs. The **speedup** is defined as follows:

$$S_p = \frac{T_1}{T_p}. \tag{2.1}$$

With:

- p being the number of cores,
- T_1 being the execution time of a program on a single-core processor ,

- T_p being the parallel execution time of a program on a multi-core processor with p cores and
- S_p being the the speedup on p cores.

Consequently, the **efficiency** E_p is defined:

$$E_p = \frac{S_p}{p} \tag{2.2}$$

The efficiency is an indication of the relative reduction of the execution time normalized by the number of processors p. Efficiency 1 is ideal efficiency, which corresponds to linear speedup. In contrast to that, the scalability of a program indicates the ability of a system to handle either a larger amount of data in linear time or to reduce the execution time of a fixed problem size adding further processors. A problem scales well, if it is suitably efficient and practical when applied to large situations

2.4.2 Amdahl's Law

After having presented the definitions of several metrics, different performance models can be described. One model was presented 1967 by Gene Amdahl [1]. He makes an estimation of the possible speedup of a program, assuming a fixed problem size and varying the number of cores p. If all effects leading to superlinear speedup (an efficiency greater than 100%) are excluded, a maximum speedup of $S_p = p$ is possible. Usually the whole program cannot be completely parallelized, as there are several parts which have to be executed sequentially, like the initialization of processes or the memory allocation. As a result, the maximum speedup is rarely reached. In Amdahl's law, the sequential part of a program is called s ($0 \leq s \leq 1$). The parallel part of the program can be expressed as $1 - s$. Hence, the parallel execution time T_p of the program on a total number of p cores can be expressed as follows:

$$T_p = s * T_1 + \frac{1-s}{p} * T_1 \tag{2.3}$$

Consequently, the speedup S_p calculated by Gene Amdahl can be computed using Equation 2.1:

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{s * T_1 + \frac{1-s}{p} * T_1} = \frac{1}{s + \frac{1-s}{p}} \tag{2.4}$$

If an infinity large number of cores p is used, it can be shown that the speedup S_p is limited by the sequential part s:

$$\lim_{p \to \infty} S_p = \lim_{p \to \infty} \frac{1}{s + \frac{1-s}{p}} = \frac{1}{s} \tag{2.5}$$

Assume, for example, that a program with a complete execution time of 10 minutes has a sequential part of 1 minute. Theoretically, the parallel part of 9 minutes can disappear parallelizing the program using an infinite large number of cores. Consequently, the speedup is limited by the sequential part ($s = \frac{1}{10}$) and can reach a maximum value of $S_\infty = 10$ in this case. Figure 2.6 illustrates this example, showing the speedup estimated

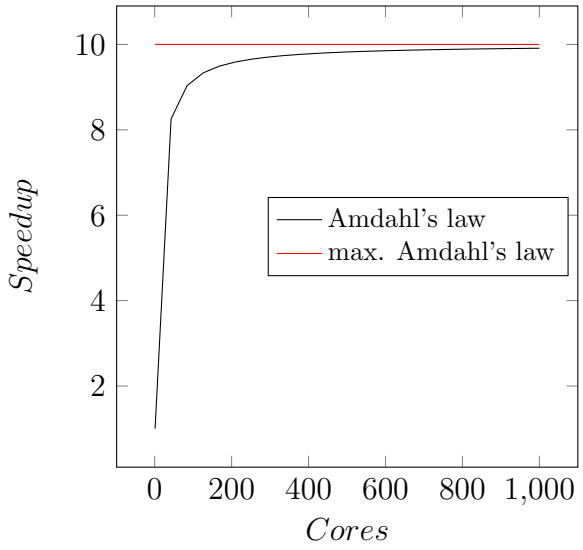


Figure 2.6: *The theoretical speedup of a program with a sequential part of 10%, based on the assumptions of Gene Amdahl. With an increasing number of cores, the speedup converges to $\frac{1}{s}$ (illustrated with the red graph), with s being the sequential part of the program.*

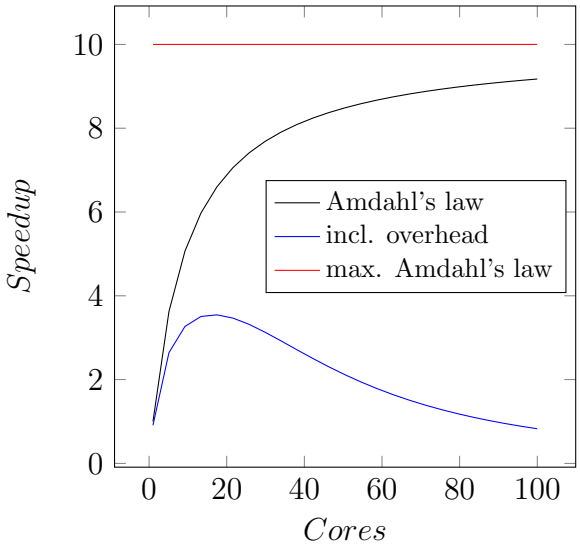


Figure 2.7: *Illustration of Amdahl's law considering overhead, presented in Equation 2.6. Like in Figure 2.6, the program has a sequential part of 10%. Compared to the prediction of Amdahl's law, the speedup estimation of the adapted model does not increase that quickly and again decreases after about 15 cores.*

with Amdahl's law using an increasing number of cores.

Amdahls assumptions, however, have some weaknesses. With an increasing number of cores, the communication, synchronization and scheduling costs increase as well. Hence, the speedup is influenced, as it is lower as predicted by Amdahl. In detail, that means that for a constant problem size and a sufficient large number of cores, the speedup is not converting to $\frac{1}{s}$ but usually again decreasing after a certain number of cores. This behavior can be modeled adding an overhead $o(p)$ of the communication cost to Amdahl's law, ike it is described in [22]:

$$Sp = \frac{1}{s + o(p) + \frac{1-s}{p}} \tag{2.6}$$

In Figure 2.7 this behavior is illustrated using the same example that is illustrated in Figure 2.6. When the speedup predicted by Amdahl is compared to the speedup estimation of the adapted model of Equation 2.6, it can be seen that the speedup of the adapted model doesn't further increase using roughly about 15 cores and more. As the overhead is increasing fast, the speedup even decreases using a higher number of cores.

2.4.3 Gustafson's Law

In contrast to Gene Amdahl, who considers decreasing the execution time of a fixed problem size using an increasing number of cores, John Gustaffson assumes constant

execution time of a program. With increasing number of cores, he also increases the problem size [19]. That means, doubling the parallel problem size of a program and running this program in parallel using twice as much cores leads to the same (fixed) execution time as before. John Gustaffson claims that any sufficient large problem can be parallelized efficiently. Thus, the speedup can be expressed as follows:

$$S_p = s + p * (1 - s) \tag{2.7}$$

He assumes that with increasing p, the sequential part s gets smaller and the parallel part $1 - s$ bigger. For infinitely large p, the speedup increases linearly.

Although Amdahl's law and Gustafson's law describe two different approaches, both are correct. It depends only on the perspective of the developer, whether a program should run faster with the same workload or whether the program should run in the same time with more workload. Nevertheless, both models are only rough approximations of the real speedup.

2.4.4 Work-Span Model

Compared to Amdahl's law and Gustafson's law, the work-span model is more useful, as it estimates an upper bound and a lower bound of the parallel speedup of a program. In the work-span model, all tasks of a program are determined and formed to a directed acyclic graph (DAG). A DAG considers dependencies between tasks, that means a task must not run as long as all its predecessors haven't stopped executing. The speedup calculations of the work-span model are completely based on T_1 and T_∞. In this model, T_1 is called work and T_∞ - the minimum execution time the algorithm would need with infinitely large number of cores - is called span. The span is equivalent to the length of the critical path. The critical path is the longest path in a DAG from start to end. In Figure 2.8 a DAG of an arbitrary example program is shown. The red boxes represent a critical path in the DAG. In this example, the work is 15 and the span is 5, assuming uniform node cost.

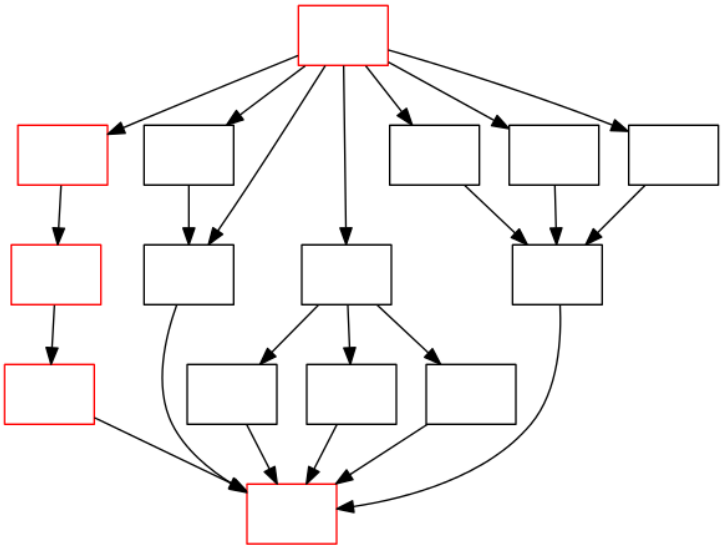
According to [38], the work-span model ignores any communication costs. Moreover, a scheduling algorithm has to be chosen for the work-span model, which immediately assigns a task to a core, whenever a core is idle and a task is ready to be executed. An implementation of such a scheduling algorithm is called a greedy scheduler. In the work-span model no superlinear speedup may occur. Consequently, the upper bound of the speedup S_p is determined as follows:

$$S_p = \frac{T_1}{T_p} \leq \frac{T_1}{T_1/p} = p. \tag{2.8}$$

On an ideal machine with infinitely large number of cores, the speedup is saturated after a certain number of cores:

$$speedup \leq \frac{work}{span}. \tag{2.9}$$

In contrast to that, Brent's Lemma [4] determines a bound T_p



;

Figure 2.8: *A directed acyclic graph. Arrows denote dependencies between tasks.*

$$T_p \leq \frac{T_1 - T_\infty}{p} + T_\infty, \tag{2.10}$$

which leads to a lower bound of the speedup S_P, when inserted in Equation 2.8:

$$S_p \geq \frac{T_1}{(T_1 - T_\infty)/p + T_\infty}. \tag{2.11}$$

The bounds can be illustrated regarding the example in Figure 2.8. Inserting $p = 4$, $T_1 = 15$ and $T_\infty = 5$ in Equation 2.11, the lower bound of the model guarantees a speedup of 2 for four cores, even if the worst possible scheduling is applied. Figure 2.9 compares Amdahl's model with the work-span model for the DAG in Figure 2.8. The sequential part s of this task graph is 2/15. Consequently, Amdahl's law estimates a maximal possible speedup of 7.5. Besides a lower bound, the work-span model provides an upper bound. The estimation for the maximal possible speedup for the given example is at most 3. This means, the work-span model provides a tighter upper bound than the model of Gene Amdahl. The reason for this is that in contrast to Amdahl's model, the work-span model considers how parallizable the parallel parts are.

Although the work-span model provides good upper and lower bounds of the speedup, this model has some drawbacks. The model assumes greedy schedulers and memory bandwidth is not limited. Furthermore this model does not consider critical sections, as they could make scheduling non-greedy.

2.5 Pitfalls

A broad overview of parallel programming was given so far. In parallel programming and especially in the process of speedup prediction, however, developers face some more difficulties. This sections describes race conditions and how they can be avoided using

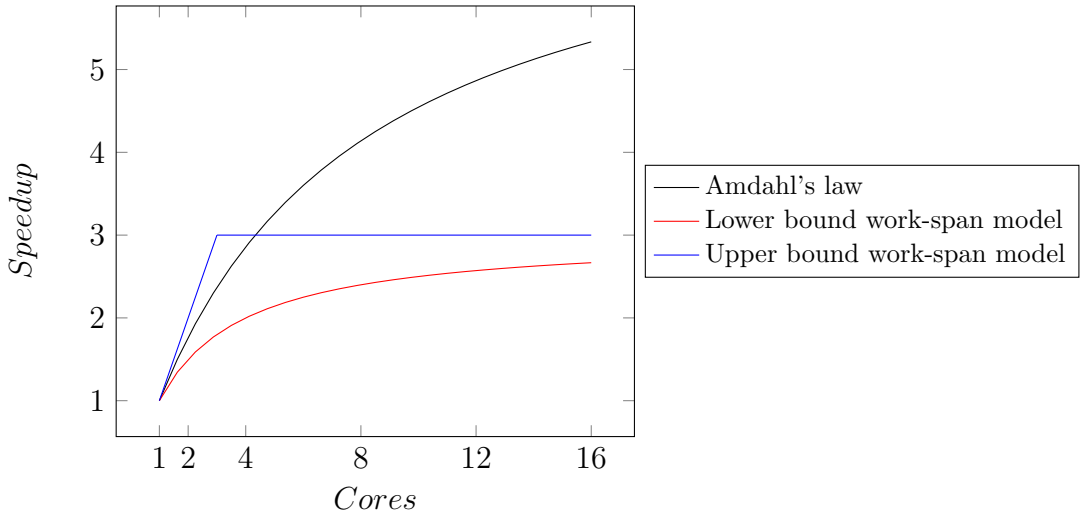


Figure 2.9: *Comparison of Amdahl's law and the work-span model.*

locks. Moreover, the term of load imbalance is explained and a first insight in parallel overhead is given.

2.5.1 Race Conditions

A race condition can occur when two or more threads perform operations on shared data and at least one operation is a write on this data. The single operations have to be executed in a certain chronological order to produce the correct result. Figure 2.10 illustrates a typical example of a race condition. On the left side a small code snippet is shown. The variable x is shared among all available threads and it is initialized with 0. In the next step, Thread 1 and Thread 2 increment the variable. The desired value of x after these two increments is 2.

```
incr(Variable x){
    x := x+1
}
---------------------------------------
Thread thread1 = createNewThread()
Thread thread2 = createNewThread()
x := 0
thread1.incr(x)
thread2.incr(x)
```

x = 0		
Thread 1	**Thread 2**	*time steps*
read(x)		1
	read(x)	2
	x:=x+1	3
x:=x+1		4
	write(x)	5
write(x)		6
x = 1		

Figure 2.10: *This example illustrates an unexpected behavior of a program due to race conditions. The pseudo code snippet on the left side intends to increment x two times - both incrementations are executed in two different threads, which are running in parallel. Consequently, x is intended to be 2 at the end of the program. The right side shows a possible execution order of operations. In contrast to the expected result, x has the value 1.*

On the right side of Figure 2.10, a possible behavior of the program is illustrated, if the

access to x is not synchronized. In time step 1 Thread 1 reads the value of the variable, which is 0. In time step 2 Thread 2 reads the value of x, which is also 0. Now, both Threads increment x locally and for both threads the value of x is 1. This value is written back to the global variable x in time step 5 and 6. However, this behavior was not intended from the developer. For the correct behavior, write(x) of Thread 1 must be finished before read(x) of Thread 2. To avoid race conditions, a parallel programmer has to identify parts of the code, where race conditions can occur and has to develop strategies to avoid them.

2.5.2 Locks

A lock is a low-level strategy to avoid race conditions. It is the implementation of a mutual exclusion and sometimes also called mutex. Usually, a lock is a global variable which has exactly one of the following two states: locked or unlocked. If task A needs exclusive access to memory, it can request a lock. This means, the global variable is set to locked and the task owns the lock until it is released by the task. For this period of time, no other task can hold this lock. That means, if task B wants to have access to the same data and requests the lock, it has to wait until A releases the lock to continue with its computations.

Code 2.2 and 2.3 show the usage of locks to synchronize the access to x in Figure 2.10. In Code 2.2, the access to x in the incr() method is synchronized and in Code 2.3 an alternative implementation of the code snippet of Table 2.3 is represented to enable the desired behavior. Using the implementation, either thread1 or thread2 holds Lock L until x is incremented and the other thread does not have access to x for this period of time.

Code 2.2: *A possible implementation of the incr() function of Table 2.10 to avoid race conditions using locks.*

```
incr(Variable x, Lock L){
   L.lock()
   x := x + 1
   L.unlock()
}
```

Code 2.3: *A synchronized version of the pseudo code snippet of Figure 2.10 using the incr() function from Code 2.2.*

```
Lock L = unlocked

Thread thread1 = createNewThread()
Thread thread2 = createNewThread()

x := 0
thread1.incr(x,L)
thread2.incr(x,L)
```

Locks are a great way to synchronize the access to shared memory. However, locks can lead to deadlocks. A deadlock is a situation, where two threads wait for each other and both threads cannot continue until the other thread proceeds. Moreover, the frequent use of locks leads to bad performance of parallel programs. Consequently, developer should use locks carefully in parallel programs. For this reason, a field of research is the development of lock free algorithms.

In the process of estimating the parallel potential of a program, however, a developer doesn't want to spend the time to implement lock free algorithms. For this reason, locks are the most important method in the prediction model presented in Section 4 to

synchronize access to shared data.

2.5.3 Load Imbalance and Overhead

Another difficulty that has to be considered in program parallelization is load imbalance. Load imbalance is the uneven distribution of tasks (regarding executiontime) across threads. Figure 2.11 shows four tasks, which are assigned to four threads. As the tasks vary in execution time, each thread has a different amount of work to do, which is called load imbalance. In general, it should be assigned roughly the same amount of work to every thread. As a result processors can be used optimally, are not idle for a long time and parallel performance is maximized.

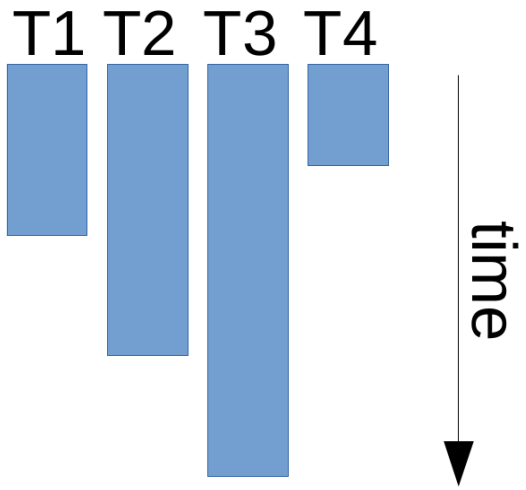


Figure 2.11: *Four tasks with different lengths are assigned to four threads. Due to the different lengths of the tasks, the load is imbalanced.*

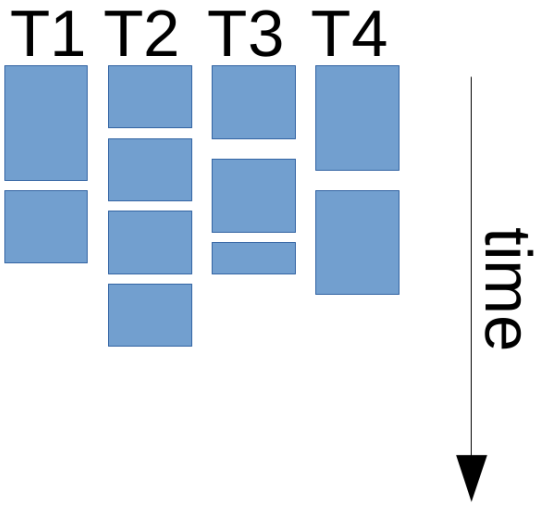


Figure 2.12: *The tasks of Figure 2.11 are divided in smaller tasks, so that the load is more balanced. This method is called over-decomposition.*

Achieving optimal load balancing is the task of the scheduler. Especially, when more tasks than threads exist. However, in the example of Figure 2.11 there are only four tasks and four threads. Nevertheless, the developer can improve the load balance by dividing tasks in several smaller tasks, if this is possible. This method is called over-decomposition [38] and is illustrated in Figure 2.12. As overhead occurs in parallel programs, over-decomposition has to be used carefully. Parallel overhead is the result of coordination and communication work among different cores to share information and enable parallel execution. This means overhead is an additional execution time that has to be added to each task. Different kinds of overhead, as well as methods to measure overhead are explained in detail in Section 4.5. If the length of tasks in the process of over-decomposition is chosen very small, it could be that the overhead is quite large compared to the task length. Consequently, over-decomposition slows down the parallel performance. As a result, usually a tradeoff between load balance and overhead has to be made.

3 Related Work

This chapter gives an overview of related research regarding parallel overhead and prediction of parallel speedup. Additionally, existing parallel speedup prediction tools are presented.

3.1 Overhead Determination

A lot of research regarding overhead measurement has happened for the last couple of years. With the EPCC Microbenchmark Suite [8] [7] [42] and CLOMP [5], two benchmark suites were developed to determine the overhead of OpenMP directives including synchronization, loop scheduling and handling of thread-private data. The EPCC Microbenchmark Suite implements an approach to compare the execution time of parallelized code with equivalent non parallelized code to determine the overhead. CLOMP extends the EPCC Microbenchmark Suite. Using the CLOMP implementation, it is possible to configure some further parameters of the benchmark, which influence the overhead. Thus, it is possible to model realistic code structures. In further research, approaches based on the EPCC Microbenchmark Suite were developed to measure task overhead [30] and overhead of nested parallelism [12] in OpenMP. Both approaches were integrated in the current version of the EPCC Microbenchmark Suite [9]. Moreover, [43] presents an approach to estimate thread start-up overhead and predicts speedup in parallelizing compilers.

3.2 Speedup Prediction

Related work describes many different approaches to predict parallel performance. In this section different works are presented and prediction tools like Kremlin, Cilkview, Cilkprof, Suitability and Parallel Prophet are compared to the prediction tool presented in this thesis.

In [21] an approach is presented, which is mainly intended to be applied in parallelizing compilers. Based on a task graph, the overhead of different scheduling approaches is predicted and a clustering algorithm is presented, which guarantees a minimum real world speedup per core for communication-free task graphs. This work mainly focuses on the scheduling in a parallelizing compiler. Moreover, the relevant task length used in this work is only a few thousand clock cycles. In contrast to that, the prediction tool developed in this thesis focuses on the speedup prediction of higher abstraction levels with bigger tasks.

The speedup estimation tool called Kismet [25] works on unmodified programs, too. Consequently the user needs not invest any effort to get a parallel speedup estimation of the

program. It builds upon the hierarchical critical path analysis [17] to localize parallelism in sequential code. To compute upper bounds of the performance, a parallel execution time model is used. Information about the program are gathered using LLVMs [31] static instrumentation. To predict the speedup of programs, which are heavily influenced by memory effects, additionally memory accesses are instrumented. Moreover, Kismet is able to transform programs, such that the parallel performance is optimized. All these operations are based on very fine grained tasks. Practically it is not realistic, however, that programmers parallelize in such fine granularity. Moreover, the fine grained instrumentations lead to a huge memory overhead and the heavy analysis to a slowdown of usually more than 100.

Another prediction approach is implemented in the Cilkview scalability analyzer [20]. In contrast to the approach presented in this thesis, Cilkview focuses on the estimation of the scalability of a program already parallelized with Cilk++ [33]. Cilkview runs under the Pin [36] dynamic instrumentation framework and dynamically gathers information about the program in sequential execution. Using these information, as well as the work and span of the task graph, an upper and a lower bound of the parallel speedup is determined and visualized. Cilkprof [44], the successor of Cilkview, additionally collects data about every call site (the call site is the location in the code where a function is either called or spawned) to estimate how much each call site contributes to the overall work and span. Consequently, scalabiliy bottlenecks in a Cilk program can be detected.

Two further tools, Parallel Prophet [27] and Suitability, which is integrated in Intel Parallel Advisor [24], also deal with parallel speedup prediction. Both tools use an approach quite similar to that used in this thesis. The user has to annotate parallel and critical regions in the sequential source code. Based on the source code, timing information is collected. Finally, the parallel behavior is simulated. In the simulation step, the tools differ. In Suitability, the parallel behavior is emulated using an interpreter, which uses a priority queue to fast forward a pseudo-clock to the next event. As Suitability is proprietary software, the source code is not open source and moreover, no detailed research work is published. Consequently, no further information is available about the technical details of the emulation. In contrast to that, the details of Parallel Prophet are described in [26].

Parallel Prophet creates a so called program tree including the information about the execution time of every task after profiling. In the program tree, it is possible to model workload imbalance, loops, tasks, critical sections and nested parallelism. The program workflows, which can be expressed with this program trees are limited, however. For example, the program presented in Code 4.3 and Figure 4.3 cannot be expressed with the program tree of Parallel Prophet, because modelizing the dependency of tasks on sequential parts in a parallel section is not supported. In contrast to that, we developed an alternative task graph in this thesis to enable the prediction of more complex programs. In Parallel Prophet, the program tree is traversed to emulate the parallel behavior of the program. To improve accuracy and consider memory effects, a memory performance model is additionally integrated in the emulator. In the prediction tool developed in this thesis, no memory model is integrated to keep the execution time low. Parallel Prophet

implements two different kinds of emulators. The fast forward approach is similar to the approach used in Suitability. A priority queue of tasks is created and the tasks are assigned to physical cores one after the other dynamically. During this step, the parallel overhead of the program is considered to achieve realistic predictions. Experimental results presented in [26] show, however, that parallel speedup is predicted inaccurately, especially in situations with high overhead. Consequently, an additional approach to predict the parallel behavior is implemented in Parallel Prophet. A so called synthesizer creates an executable parallel program out of the program tree. This program is executed on the target platform and the parallel speedup is estimated. However, the complete execution of a parallel program should be avoided in the prediction tool developed in this thesis. The disadvantage of this approach is, that the prediction is only possible on the target platform. Consequently, a speedup estimation for more cores available on the target platform is not possible. Hence, we implement a scheduler with a sophisticated overhead determination approach to estimate accurate prediction results in our prediction tool. Although this overhead determination approach has to be executed on the target platform, too, the schedeuling enables a speedup estimation for an arbitrary number of cores, if the overhead is neglected. In [27] and [26] it is not exactly explained, how the overhead factors are determined in Parallel Prophet. Moreover, the source code is not available. Consequently, a detailed analysis of the overhead determination method in Parallel Prophet is not possible.

4 Performance Prediction Tool

In the previous chapters reasons showing the need for parallelization of sequential programs were presented. However, program parallelization can be quite difficult. Concurrent parts of the code have to be identified and dependencies, which may prevent parallel execution, have to be detected. Moreover, work has to be partitioned equally among processors and parallel overhead has to be considered to enable the optimal execution time. But even when concurrency and dependencies are detected, the code must often be refactored to enable parallel execution. This is a lot of work and additionally only a minority of programmers have experience in parallel programming. Hence, program parallelization is expensive. For this reason, companies are interested in the the parallel potential of existing sequential programs. If they could expect high parallel speedup, they are facing a lower risk to spend the effort and money to parallelize the program. Consequently, there is a high need for tool support in this area. As only a small number of tools for speedup prediction exist at the moment (see Chapter 3), each of them having some drawbacks, the main subject of this thesis is the development of a performance prediction tool, which is described in detail in this chapter. First of all, an overview of the prediction tool is given, before each component is described in detail.

4.1 Overview

The performance prediction tool developed in this thesis is able to estimate the potential of parallel speedup and scalability of existing programs. The effort for the user of the tool to estimate the potential of a program must be a fraction of the actual program parallelization. We present a tool, which estimates a tight upper bound. But not every parallel implementation is optimal and for every single prediction step optimistic assumptions were made. Consequently, the predicted speedup usually exceeds the real speedup. For the prediction process, finding a good balance between prediction accuracy and program overhead is one of the key tasks in creating a model for the tool. Approaches, which lead to higher accuracy, like a memory simulation, and a more accurate approach to determine the parallel overhead are not realized in the tool, because they would increase the execution time to much.

The prediction tool developed in this thesis follows an annotation approach. This means, the user annotates concurrency in the sequential code. Moreover, critical sections have to be identified by the user and must be marked using annotations.

Using these annotations the prediction tool supports the precise prediction of

- **loops** and

- **tasks**.

Loops and tasks cover most of the parallelizable parts of sequential programs. Within parallel sections, the tool can handle

- **workload imbalance**,
- **critical sections** and
- **nested parallelism**.

The main focus in this thesis lies on the most accurate prediction of programs parallelized with OpenMP. Programs parallelized with other programming models can also be predicted with this tool, however, the same prediction accuracy as programs parallelized with OpenMP cannot be guaranteed, because they differ in scheduling, overhead and parallelization concepts. The support of these programming models is not part of this thesis, but due to the design of the tool, it can easily be integrated in future program versions.

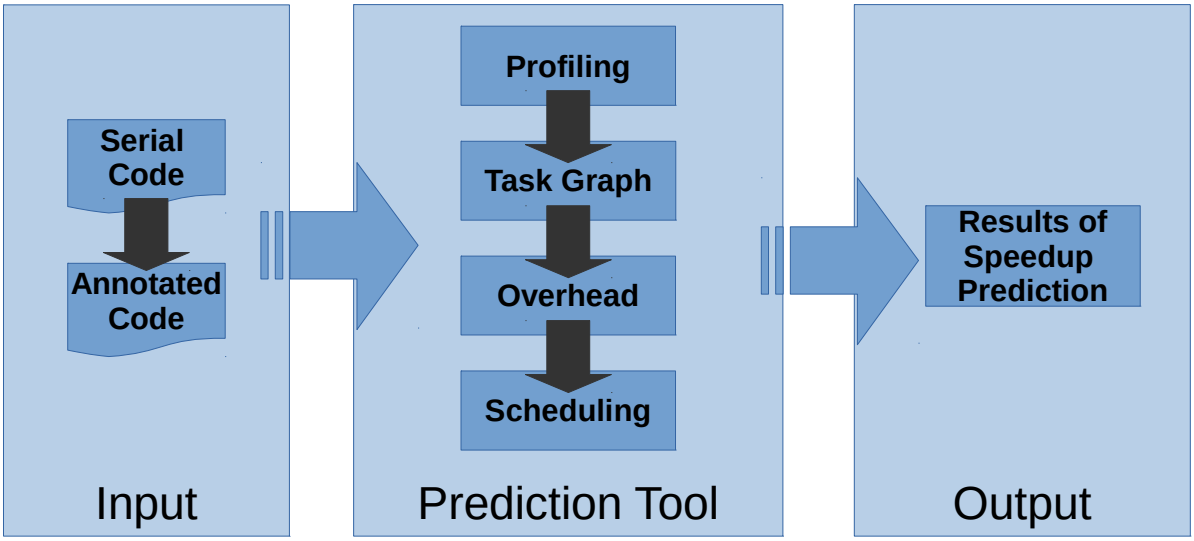


Figure 4.1: *Workflow of the performance prediction tool.*

The concepts and methods used to achieve accurate speedup prediction are presented in detail in the following sections. Figure 4.1 gives an overview of the workflow of the prediction tool. The tool expects an annotated sequential program as input using the annotations presented in Section 4.2. Figure 1.2 shows an example usage of different annotations. Each identified region consists of a starting point and an end point. Thus each pair of annotations represents an instruction interval. The execution time of these intervals is determined using interval profiling. Details of the interval profiling are explained in Section 4.3. With the information of the annotations and the interval profiling, a task graph is created (see Section 4.4). Now all required information is available to determine the parallel overhead and to schedule the tasks. As the overhead is machine dependent, it is determined at each run of the prediction tool, as described in Section 4.5. For this

reason the speedup and scalability of a program can only be determined on the machine on which the tool runs. The overhead is added to the appropriate nodes in the task graph. Afterwards scheduling using the Modified-Critical-Path algorithm [49], described in Section 4.6, is performed. This algorithm is implemented in the current version of the tool, because it provides very good scheduling results in little execution time [29]. Finally, the simulated execution time for a specific number of threads is compared to the sequential execution time to determine the speedup, which is presented to the user.

4.2 Annotations

The performance prediction tool needs information about which parts of the program can be executed in parallel and about critical sections. The user is responsible to insert annotations into the source code of a sequential program. The concept of using annotations is the result of finding the right balance between accuracy and user overhead. Parallelizing the source code completely without predicting its speedup would lead to the highest accuracy. As already mentioned, this method is very time consuming and complex, however. In contrast to that, a concept letting the compiler identify all the necessary information by its own would result in little, or even no user overhead. As described in Section 2.3, however, this method has some drawbacks in parallel performance and accuracy. Consequently, using annotations is a reasonable tradeoff, because the user has to spend little effort and knowledge to insert the annotations and the rest of the prediction is done by the tool. A short overview of the supported annotations of the tool can be found in Table 4.1. The annotations are technically implemented as function calls.

Annotation	**Short description**
PAR_BEGIN(int min, int max)	Initialization of the prediction tool. Start point of time profiling.
PAR_END()	Completion of the prediction tool. End point of profiling.
PAR_SEC_BEGIN(String sec_name, bool isLoop)	Start of parallel section.
PAR_SEC_END()	End of parallel section.
PAR_TASK_BEGIN(String task_name)	Start of independent task.
PAR_TASK_END()	End of independent task.
PAR_LOCK_BEGIN(void *lock)	Request a lock.
PAR_LOCK_END(void *lock)	Release the owned lock.

Table 4.1: *Overview of the annotations in the prediction tool.*

4.2.1 Description of the Different Annotations

In the prediction tool eight annotations are used. Using *PAR_BEGIN()* and *PAR_END()*, the prediction tool can be started and stopped. *PAR_SEC_BEGIN()*, *PAR_SEC_END()*, *PAR_TASK_BEGIN()* and *PAR_TASK_END()* mark parallel executable work and enable the expression of dependencies between tasks, respectively. Moreover, it is possible to

define critical sections with *PAR_LOCK_BEGING()* and *PAR_LOCK_END()*. Each of these annotation is described in detail below:

- **PAR_BEGIN(int min, int max):** The prediction tool starts with this annotation. This includes the initialization of the tool and initialization of the interval profiling. Moreover, the profiling of the sequential versions starts at this point. The parameters *min* and *max* indicate the minimal and maximal number of cores that shall be predicted. Note that the tool only predicts the power of two number of cores. For example *PAR_BEGIN(3,8)* starts the tool, which is going to predict the parallel speedup for four and eight cores.

- **PAR_END():** Indicates the endpoint of the source code that shall be predicted. Consequently the interval profiling stops with this annotation. Within this function call, the prediction process starts.

- **PAR_SEC_BEGIN(String section_name, bool isLoop):** A parallel section with the ID *section_name* is started. All tasks created within this section may run in parallel. The variable *isLoop* indicates whether the tasks created within this section are handled as loop iterations or normal tasks. This influences the overhead determination of the tasks within this parallel section.

- **PAR_SEC_END():** The current parallel section is closed. Additionally, an implicit barrier is defined at this point. This means, further program execution has to wait, until the complete work within this section is terminated.

- **PAR_TASK_BEGIN(String task_name):** Starting point of the task with the ID *task_name*. All tasks within the same parallel section can be executed in parallel. This implies, that tasks within this section are independent of each other. A task interval (the interval between *PAR_TASK_BEGIN()* and *PAR_TASK_BEGIN()*) can also mark loop iterations that may run in parallel to each other. This annotation must occur within a parallel section.

- **PAR_TASK_END():** End point of the current task. *PAR_TASK_END()* must occur within a parallel section.

- **PAR_LOCK_BEGIN(void *lock):** A critical section is started and the lock &lock is requested. This critical section may be executed of at most one thread at the same time until the lock is released again. *PAR_LOCK_BEGIN(void*)* must occur within a parallel section. Moreover, multiple locks are supported in the prediction tool. For this reason it is necessary to specify a pointer to the lock which is requested.

- **PAR_LOCK_END(void *lock):** This is the end of the critical section protected by the lock &lock. Thus the lock &lock is released. The annotation must occur within a parallel section.

4.2.2 Dependencies and Synchronization Using Annotations

After the annotations are defined exactly, some examples are introduced in the following to illustrate the correct usage of the annotations.

First of all, we explain how loops can be annotated. In the PAR_SEC_BEGIN(String section_name, bool isLoop) annotation, the user can enable or disable the *isLoop*-flag. If the *isLoop*-flag is set, the tasks within this parallel section are handled as loop iterations, otherwise they are handled as tasks. This influences the parallel overhead used by the speedup prediction. In Section 4.5 it is described that the scheduling of tasks has a higher overhead than the scheduling of loop iterations. The loop annotation is illustrated using two examples. In Code 4.1 the boolean *isLoop*-flag is "*false*" whereas it is "*true*" in Code 4.2. Consequently, the prediction result of these examples can differ. But nevertheless, each task interval is executed in parallel in both examples and both code snippets lead to the same task graph presented in Figure 4.2.

In this paragraph, some dependency and synchronization issues are discussed. This section should highlight the impact of implicit barriers and sequential regions. A sequential block is a sequence of instructions, which is not surrounded by a task interval. If sequential blocks are within a parallel section, the subsequent tasks have to be scheduled such that they start executing after the sequential block has finished. Code 4.3 illustrates such an example and the snippet in Code 4.4 represents the related annotated code. In this example the sequential execution of f2() leads to a dependency. Obviously f3() has to be executed after f(2). However, there is no special policy when to execute f1(). Using two cores, the following parallel program executions could be possible (cores can be exchanged):

- **core1:** f1() before f2() - **core2:** f3()
- **core1:** f2() - **core2:** f1() before f3()

Thus, the annotations in Code 4.4 would result in the creation of a task graph being similar to the task graph in Figure 4.3.

A similar example is given in Code 4.5. In this code snippet the loop iterations consist of a parallel executable task and a sequential part, which results in the task graph shown in Figure 4.4. The sequential part depends on the sequential part of the previous loop iteration. This means the sequential part of iteration i is definitively executed before the sequential part of iteration i+1. In contrast to the sequential parts, more possibilities exist to schedule parallel tasks. However, the task of iteration i+1 must not start before the sequential part of iteration i has terminated. In OpenMP similar behavior can be expressed using the *ordered* directive.

An example using the OpenMP *barrier* directive is shown in Code 4.6. To keep the number of annotations small, the prediction tool does not support explicit barriers. Consequently, synchronization must be expressed using the implicit barrier of $PAR\_SEC\_END()$. An appropriate annotated version of Code 4.6 is illustrated in Code 4.7. A task graph expressing this behavior is illustrated in Figure 4.5. The behavior of the barrier implies that

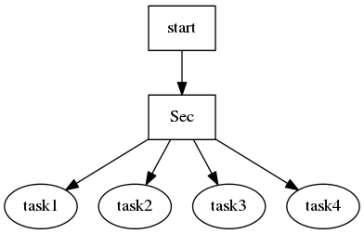
Code 4.1: *Four tasks that can be executed completely in parallel.*

```
PAR_BEGIN(4,4);
  PAR_SEC_BEGIN("Sec",
      false);
    PAR_TASK_BEGIN("t1");
      //do some work
    PAR_TASK_END();
    PAR_TASK_BEGIN("t2");
      //do some work
    PAR_TASK_END();
    PAR_TASK_BEGIN("t3");
      //do some work
    PAR_TASK_END();
    PAR_TASK_BEGIN("t4");
      //do some work
    PAR_TASK_END();
  PAR_SEC_END();
PAR_END();
```

Code 4.2: *Four tasks that can be executed completely in parallel. Each loop iteration represents a task.*

```
PAR_BEGIN(4,4);
  PAR_SEC_BEGIN("Sec",
      true);
 for(int i=0; i<4; i++)
    {
      PAR_TASK_BEGIN("t$i");
        //do some work
      PAR_TASK_END();
    }
  PAR_SEC_END();
PAR_END();
```

Figure 4.2: *The task graph for Code 4.1 and Code 4.2. Four tasks are executed in parallel.*



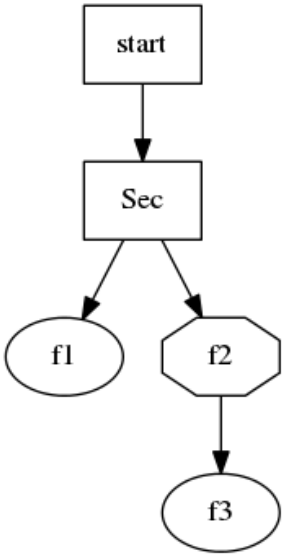
Code 4.3: *An OpenMP example with an expected synchronisation after the sequential part.*

```
#pragma omp parallel
{
  #pragma single
  {
    #pragma omp task
    {f1()}
    f2()
    #pragma omp task
    {f3()}
  }
}
```

Code 4.4: *Anotations for equivilant behavior as the OpenMP example in Code 4.3.*

```
PAR_BEGIN(2,2)
  PAR_SEC_BEGIN(1)
    PAR_TASK_BEGIN(t1)
      f1()
    PAR_TASK_END(t1)
      f2()
    PAR_TASK_BEGIN(t2)
      f3()
    PAR_TASK_END(t2)
  PAR_SEC_END(1)
PAR_SEC_END()
```

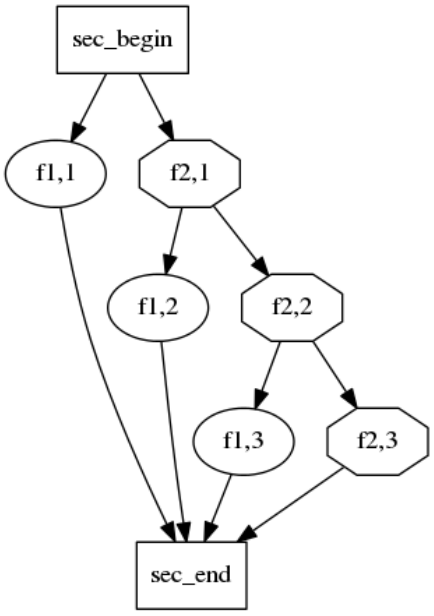
Figure 4.3: *The appropriate task graph for the example in Code 4.4.*



Code 4.5: *A loop containing concurrently executable work and a sequential part.*

```
PAR_SEC_BEGIN("sec", true);
   for(int i=1; i<4; i++)
   {
    PAR_TASK_BEGIN("task");
      f1();
    PAR_TASK_END();
    f2();
   }
PAR_SEC_END();
```

Figure 4.4: *Task graph of a loop containing sequential parts. The appropriate annotated code can be found in Code 4.5.*



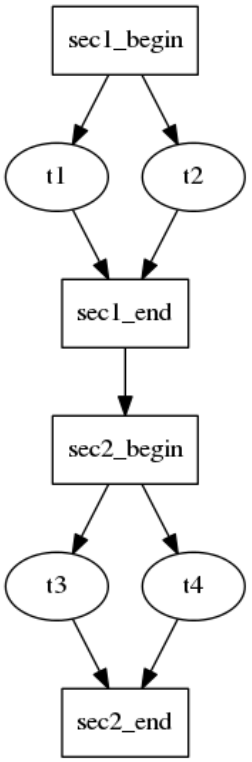
Code 4.6: *An OpenMP example using a barrier after executing the first two functions.*

```
#pragma omp parallel
{
  #pragma single
  {
    #pragma omp task
    {f1()}
    #pragma omp task
    {f2()}

    #pragma omp barrier

    #pragma omp task
    {f3()}
    #pragma omp task
    {f4()}
  }
}
```

Code 4.7: *Example of annotated code with similar behavior like the example in Code 4.6.*

```
PAR_SEC_BEGIN("sec1",
    false)
  PAR_TASK_BEGIN("t1")
    f1()
  PAR_TASK_END()
  PAR_TASK_BEGIN("t2")
    f2()
  PAR_TASK_END()
PAR_SEC_END()
PAR_SEC_BEGIN("sec2")
  PAR_TASK_BEGIN("t3")
    f3()
  PAR_TASK_END()
  PAR_TASK_BEGIN("t4")
    f4()
  PAR_TASK_END()
PAR_SEC_END()
```

Figure 4.5: *Task graph of the example in Code 4.7.*



the start node of sec2 must start after the end node of sec1 has terminated. This means f1() and f2() have finished their calculations before f3() and f4() start.

4.3 Interval Profiling

The annotations inserted by the user provide the initial information for the prediction. To predict the speedup, however, information about the execution time of tasks, critical sections and sequential parts of the program is needed. For this purpose, a highly accurate tracing implementation is used in the prediction tool and is extended to an interval profiler.

4.3.1 Profiling Model

The profiler handles each annotation as an event. During the execution of the annotated sequential program, the time since the last event is determined and saved. As the time between two events is measured, which represents intervals, this method is called interval profiling. The right part of Figure 4.6 shows a possible example output of the interval profiler for the annotated source code shown on the left part of the figure. Each line gives information about the occurred event and the number of clock cycles elapsed since the last event. For example, 3120 clock cycles elapsed between the annotations PAR_SEC_BEGIN() and PAR_SEC_END(). To determine the execution time of the task in the code snippet - that means the execution time for each event between $PAR\_TASK\_BEGIN()$ and $PAR\_TASK\_END()$ - the results of the profiling for each event after *task_begin* up to the according *task_end*-event must be summed up. For example, the task in Figure 4.6 has an execution time of 42140 + 1200 + 14000 = 57340 clock cycles.

```
//...
PAR_SEC_BEGIN("sec_begin", false);
   //do something
   PAR_TASK_BEGIN("task_begin");
      //do something
      PAR_LOCK_BEGIN(&lock);
         //do something
      PAR_LOCK_END(&lock);
      //do something
   PAR_TASK_END();
   //do something
PAR_SEC_END();
//...
```

```
...
sec_begin 30108
   task_begin 3120
      lock_begin 42140
      lock_end 1200
   task_end 14000
sec_end 120984
...
```

Figure 4.6: *The right side shows an example output of the interval profiler after having profiled the annotated code on the left. The profiler saves the event and the clock-cycles elapsed since the last event.*

4.3.2 Time Measurement Implementation

If a source code consists of a large number of very small tasks, small deviations in time profiling can lead to high inaccuracy in the prediction results. Thus, the time measurement should be as precise as possible. Modern processors offer timing devices like real-time clock (RTC), High Precision Event Timer (HPET) and Advanced Programmable Interrupt Controller (APIC). However, access to all of these timers involve a system call. This includes significant overhead, which must be reduced to achieve accurate measurements. For this reason, the interval profiler implemented in the prediction tool uses the Time Stamp Counter (TSC), which can be accessed using an assembly instruction. As a result no additional context switch by the operating system is necessary. When reading the TSC, the number of elapsed clock cycles since processor reset is returned. Moreover, the TSC is not directly influenced by frequency boosting technologies, like AMD's Turbo Core or Intel's Turbo Boost, but runs at a constant rate. The program execution time itself, however, is influenced by frequency boosting. In order to achieve accurate prediction results, frequency boosting technologies have to be disabled.

The time measurement in the prediction tool is implemented using the function __rdtscp() of the TSC to determine a timestamp of the last occurred event and the current event. The difference is the number of interval clock cycles elapsed. Additionally, the average number of cycles to call the __rdtscp() routine is determined and subtracted from the interval time:

$$duration = timstamp_{last\ event} - timestamp_{current\ event} - avg\_time_{\_\_rdtscp()}. \quad (4.1)$$

This ensures, that the method call overhead is not included in the profiling results. Knowing the processor clock frequency enables the conversion from clock cycles to time. The formula for the conversion is

$$execution\ time[s] = \frac{clock\ cycles[\frac{1}{s}]}{processor\ frequency}. \quad (4.2)$$

4.4 Task Graph Creation

Using the information from the interval profiling and the annotations, a task graph is created. A task graph is a DAG, as it is described in Section 2.4.4. It is necessary to model dependencies between different regions of the program. The task graph consists of the following four types of nodes:

- section-begin-node
- section-end-node
- sequential-node
- task-begin-node

The **section-begin-node** is created, as soon as a parallel section is entered and the **section-end-node** is created, when a parallel section is left. Information whether the parallel section is a loop or not is saved in the section-begin-node.

In Section 4.3 it is described that each annotation is implemented as an event. Whenever an event occurs, a **sequential-node** is created, saving the elapsed time since the last event. The sequential-node is the only node, which saves information about the execution time.

Whenever a task interval is entered, additionally a **task-begin-node** is created. This node does not save execution time information of the task, only information about the overhead. In the task graph, the task-begin-node represents the beginning of a parallel executable task. Each task-begin-node is followed by one ore more sequential-nodes, which are contained in the task region. These nodes contain the necessary timing information of the current task.

To additionally represent critical sections in the task graph the sequential-node which is created when a critical section is entered, saves the lock_ID of the lock it should be synchronized with.

```
int i = 1;
PAR_TASK_BEGIN("Task");
   //do some work 5000 cycles
   PAR_LOCK_BEGIN(&i);
      //do some work 400 cycles
   PAR_LOCK_END(&i);
PAR_TASK_END();
```

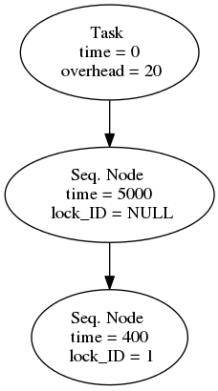


Figure 4.7: *The task graph on the right illustrates the internal representation of the task, which is annotated on the left side. A task interval consists of one or more sequential-nodes and saves no timing information, but overhead information. In contrast to that, the sequential-nodes save information about the execution time and moreover can mark a node as critical section, if the lock_ID is initialized.*

To illustrate the internal representation of tasks and critical sections in the task graph Figure 4.7 shows a code snippet and the appropriate representation of the task graph. In the snippet, a task is annotated. Within the task interval some work is done, which takes 5000 clock cycles, before a critical section is executed, which takes 400 clock cycles. In the task graph a task-begin-node is created, saving its overhead and no execution time. Additionally, we know that the task interval contains two sequential regions. Both are represented as sequential-nodes and belong to the task-begin-node. They save the execution time of each region. Additionally, the sequential-node which represents the critical section has initialized the lock_ID with 1. The overall execution time of the task is

the sum of the execution times of the sequential-nodes belonging to the task-begin-node. Consequently, the task "Task" has an execution time of 5400 clock cycles (neglecting the overhead). The complete execution time of a task is also called task length in the following.

The goal of the task graph is to model dependencies between the nodes. Consequently, nodes have to be connected with edges, if they depend on another node. Dependency means, a node may not be executed before its predecessor has finished. The algorithm to create and connect nodes is shown in Code 4.8. In the algorithm the current active section and node is saved. Additionally, a stack exists, which saves open tasks. When a task interval is entered, a task is pushed to the stack. When a task interval is left, the topmost task is poped. The topmost task of the stack represents the current active task. This approach helps to keep track of open tasks, when nested parallelism occurs.

Outgoing and incoming edges are only connected with the current active node. In general, the algorithm is based upon the following rules:

- Whenever a node is created, it is connected with the current active node.
- Whenever a sequential node, a task-begin-node or a section-end-node is created, this node becomes active, because the following nodes depend on the execution of this node.
- When $PAR\_TASK\_END()$ is called, the parent node of the active task-begin-node becomes active.
- Each node inside a parallel section, which has no successor must be connected with the section-end-node.

Code 4.8: *Algorithm to create a task graph.*

```
initialization of the task graph:
   1. create root node: <start>
   2. set active node: cur_node = <start>
   3. set active section: cur_s = NULL

when encountering a...

a. sequential region (time):
   1. create node: u = <seq, cur_s, time>
   2. create edge: cur_node -> u
   3. cur_node = u
   4. add u to cur_sec.nodes
   5. if cur_node == lock:
       set lockID

b. section_begin (name):
   1. create node: u = <sec_begin, cur_s, name>
   2. create edge: cur_node -> u
```

```
   3. cur_node = u
   4. create Section s
   5. set active section: cur_s = s
   6. add u to cur_sec.nodes

c. section_end (name):
   1. create node: u = <sec_end, cur_s, name>
   2. for each t in cur_s.nodes:
         if t has no predecessor:
            create edge: t -> u
   3. set cur_node = u
   4. update cur_s

d. task_begin (name, time):
   1. create node: u = <task_begin, cur_s, name, time>
   2. create edge: cur_node -> u
   3. cur_node = u
   4. add u to cur_sec.nodes
   5. push u to open_tasks

e. task_end():
   1. set cur_node = the parent of the appropriate task_begin (topmost element
      of open_tasks)
   2. update open_tasks
```

4.5 Overhead

Even if parallel work can be distributed equally among p processors, this usually does not lead to an execution time of $\frac{1}{p} *$ *sequential execution time*, which results in a linear speedup. The real execution time of the program on each core is usually a bit higher, because parallel overhead has to be considered. Parallel overhead is the result of coordination and communication work among different cores to share information and enable parallel execution. In general, different sources of overhead exist in a parallel program.

One source of overhead is the starting and terminating process of a parallel library, like OpenMP. This overhead occurs only once during a run of a program. Moreover, the thread startup overhead has to be considered.

In OpenMP, the thread startup overhead is the time to create OpenMP threads. Usually a thread pool is implemented in OpenMP libraries. This is a pool of threads, which is created at the start of a parallel program. Whenever a task is ready to be executed, a thread is requested and the task is assigned to this thread. As soon as a task has finished, the thread can be released and it is passed back to the thread pool. The advantage of this method is that threads need not be created and terminated for every parallel section or even for every task. That means, also the thread startup overhead must only be considered once at the beginning of the parallel program and at the end, when the threads have

to be terminated. For this reason, as well as the fact that the thread startup overhead and the library startup overhead is only a very small part of the overall overhead [34], these two kinds of overheads are neglected in the process of parallel speedup prediction.

Another kind of overhead occurs during the process of lock management in critical sections. Using OpenMP, this overhead may result from the directives CRITICAL and ATOMIC. Although this kind of overhead can be found frequently it is small compared to blocking [34] which is an effect in nearly every critical section.

Additionally, the process of context switching produces overhead. The change of a process on the current core by the operating system is called context switch. Similarly, a context switch occurs, when a parallel programming model, like OpenMP, changes the task being executed on the current thread. In this step the current context is saved and a new context is loaded. This procedure includes a lot of administration, like saving and loading registers and maps, updating tables and many more. Hence, this is quite time-consuming and thus this overhead has to be considered in the speedup prediction.

The context switch itself is executed by the dispatcher. However, the strategy for the context switches is created from the scheduler and leads to the scheduling overhead. This includes the calculation which task should be executed from which thread. In static scheduling, the distribution of the tasks on different threads needs to be determined only once. But scheduling can also be an ongoing process, which is the case in dynamic scheduling. When dynamic scheduling is applied, the scheduler has to decide which task to assign to a thread whenever a task has finished and a processor or thread has no more tasks to handle. Consequently, dynamic scheduling usually leads to a better scheduling, because it considers the length of single tasks and can react on unexpected interruptions. However, Figure 4.8 shows that dynamic scheduling has more overhead than static scheduling. Moreover, dynamic scheduling must be used in situations in which static scheduling is not possible. This is the case, when the number of tasks in not known at compile time, for example. In the figure static scheduling is compared to dynamic scheduling. For this purpose the overhead of a loop with 2048 iterations and a task length of 15 microseconds is measured using a different number of cores and increasing chunksize. Besides the overhead of loop scheduling, the overhead of task scheduling is considered in this thesis.

4.5.1 Overhead Determination

No matter what the source of the overhead is, the overhead depends on the executing computer. Among others, the memory system, the processor and the operating system have influence on the overhead. Consequently, making measurements on the target system is the most accurate approach to determine the overhead. Moreover, the overhead is also dependent on the OpenMP implementation [8]. To estimate the overhead of OpenMP programs the EPCC Microbenchmark Suite [8] was developed. The EPCC Microbenchmark Suite provides a framework to measure the overhead of each OpenMP directive. Thus, the thread start-up overhead (using the *omp parallel* directive), the task overhead [9] including nested parallelism [12], synchronization overhead and loop scheduling overhead [7] of an OpenMP implementation on a specific system can be evaluated.

The overhead of an OpenMP directive is the difference between the sequential execution time and the parallel execution time using the appropriate OpenMP directives. Let T_s represent the measured sequential execution time and pT_p represent the measured composite parallel execution time of the same program using p threads. Consequently, the overhead $T_{overhead}$ of a single thread is

$$T_{overhead} = T_p - \frac{T_s}{p} \tag{4.3}$$

To determine the overhead of an OpenMP-parallelized loop, the time to execute the code snippet of Code 4.9 is measured and compared to the execution time of a sequential reference code, which is presented in Code 4.10. In these code snippets, the function *delay* simulates the duration of a single iteration. It contains a dummy loop of length *delaylength*. Changing delaylength enables the user to simulate different task lengths. To determine the overhead of a single loop execution, the measured overhead of Code 4.9 must be divided by the number of iterations of the outer-loop (*reps*). This variable should be big enough, so that the overhead of the enclosing PARALLEL pragma can be neglected.

Code 4.9: *Code snippet including overhead of a parallelized loop.*

```
#pragma omp parallel
{
   for (j = 0; j < reps; j++) {
      #pragma omp for schedule([static|dynamic|guided],chunksize)
      for (i = 0; i < itersperthr * nthreads; i++) {
        delay( delaylength );
      }
   }
}
```

Code 4.10: *Sequential reference time to determine parallel overhead.*

```
for (j = 0; j < reps; j++){
   for (i = 0; i < itersperthr; i++) {
      delay( delaylength );
   }
}
```

Additional to the prediction of parallel loops, the tool also supports the speedup prediction of user defined tasks, which resemble OpenMP tasks. Due to more complex scheduling, the overhead of task scheduling is usually bigger than the overhead of loop scheduling. For this reason, it is necessary to estimate the task scheduling overhead in the prediction tool. In general the determination is similar to the determination of the loop-scheduling overhead. The parallel execution of several tasks (see Code 4.11) is compared to a reference method (see Code 4.12). Finally, the difference is divided by *reps* to determine the overhead of a single task. In contrast to Code 4.9, however, it is important that the tasks are created in a single thread. That means, the *omp single* pragma has to be used.

Code 4.11: *Code snippet including overhead of a parallelized loop.*

```
#pragma omp parallel private(j)
{
#pragma omp single
{
   for (int j = 0; j < reps * nthreads; j++) {
      #pragma omp task
      { delay(delaylength);}
   }
}
}
```

Code 4.12: *Sequential reference time to determine parallel overhead.*

```
for (int j = 0; j < reps; j++) {
   delay(delaylength);
}
```

To get more information about the behavior of the scheduling overhead of loops and tasks, several measurements were made using the appropriate methods of the EPCC Microbenchmark Suite. The benchmarks were executed on a server consisting of four Opteron processors with the detailed configurations presented in Table 4.2. In general, the results indicate, that in addition to the OpenMP implementation, the machine specification and the operating system, the overhead is mainly dependent on the number of threads and the length of the executed tasks.

Figure 4.11 presents the overhead of a loop with 128 iterations relative to the execution time of the loop. Similarly Figure 4.12 presents the overhead of different tasks relative to their task lengths. In these figures, it can be seen that an increasing number of threads leads to a bigger scheduling overhead. Moreover, in both examples the relative overhead is comparatively high for a small task length. Using a task length of one microsecond, the relative overhead in a loop is about 16% using two and four threads, about 39% using eight threads and 183% using 16 threads. In contrast to that the, the overhead for task scheduling is obviously higher. For a task length of one microsecond the relative overhead is about 171% using two threads, 936% using four threads, about 5307% using eight threads and even 24752% using 16 threads.

In Figure 4.10, the absolute loop-scheduling overhead for the same loop as in Figure 4.11 is presented. In general, an increasing task length leads to a higher overhead. Especially the overhead using 16 threads increases strongly. Using 16 threads, the overhead of a task with length 2048 microseconds is approximately 1600% of the overhead of a task having a length of 8 microseconds. The increase for the same task lengths is about 650% using eight threads and only 4% using two and four threads. Additionally, the overhead for very small tasks increases using eight and 16 cores. But all the measurements have in common, that the relative overhead becomes less important, the bigger the tasks get.

Furthermore, the results show that the loop scheduling overhead is also dependent on the

Server Configurations	
Server Name	Opteron
Sockets	4
Cores	12 per Socket
Processor	AMD Operon 6168 1.90 GHz
L1 Cache	64 + 64 KB
L2 Cache	512 KB
L3 Cache	12 MB shared
Compiler	GCC 4.8.0
Compiler flags	-fopenmp -O2

Table 4.2: *The server specification used for the measurements in this thesis.*

number of loop iterations and the chunksize. Figure 4.8 shows the static and dynamic scheduling overhead for an increasing chunksize. These experimental results confirm the fact, described in related work [7], that bigger chunksize leads to less overhead. Moreover, Figure 4.9 illustrates the absolute overhead of loops with a task length of 128 microseconds. Additionally the number of loop iterations is increased in each run. It can be figured out that the overhead also increases with a bigger number of loop iterations.

4.5.2 Overhead Prediction in the Tool

To support as many target machines as possible, measurements similar to them implemented in the EPCC Microbenchmark Suite are made in each run of the prediction tool. However, the overhead determination is expensive. Consequently the implementation of the original benchmark is adapted to reach a reasonable execution time of the prediction tool. Moreover, we developed a concept to predict the overhead accurately with as little measurements as possible.

Adaptions in the source code of the benchmark are made in the initialization of *delaylength* and in the number of repetitions of the benchmark. Apart from that, the basics of the overhead determination as introduced in Code 4.9 and Code 4.11, are not changed. As a result, the execution time to determine the overhead of a loop, for example, is less than ten percent of the original benchmark, without remarkable loss of accuracy.

Additional time can be saved minimizing the number of benchmark calls to determine the overhead. For this purpose, two different approaches are created, depending on whether the overhead of a loop, or the overhead of a task is determined.

Loop-Scheduling Overhead

If the overhead of a loop is determined, first the average length of all loop iterations within a parallel section is calculated. The average length of the loop iterations and the number of iterations are used as parameters to determine the overhead using the method of Code 4.9. Thus, the overhead is determined once in each parallel section. However, the determination of loop-scheduling overhead is very expensive. Especially calling the

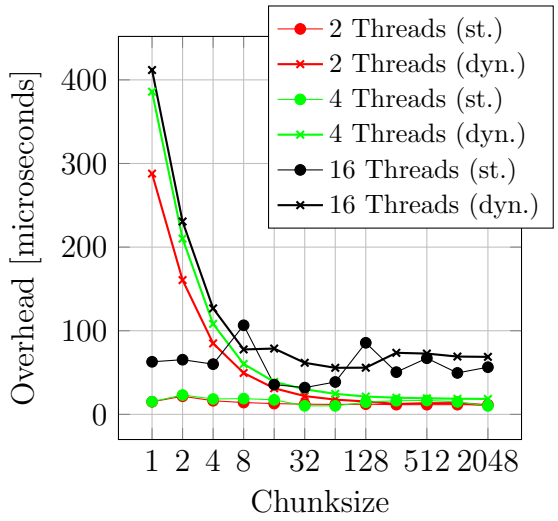


Figure 4.8: *The absolute overhead of a loop with 2048 loop iterations, each having a length of 15 microseconds. Different numbers of threads, as well as an increasing chunksize was chosen to compare the overhead. Moreover, static scheduling is compared to dynamic scheduling.*

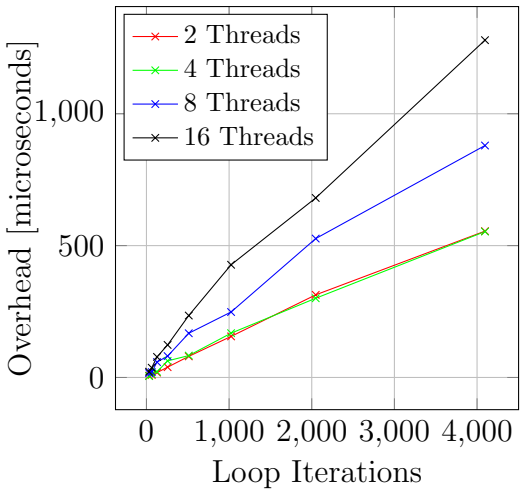


Figure 4.9: *The absolute overhead of a loop scheduled with (dynamic,1) and a task length of 128 microseconds. The overhead is compared using an increasing number of loop iterations.*

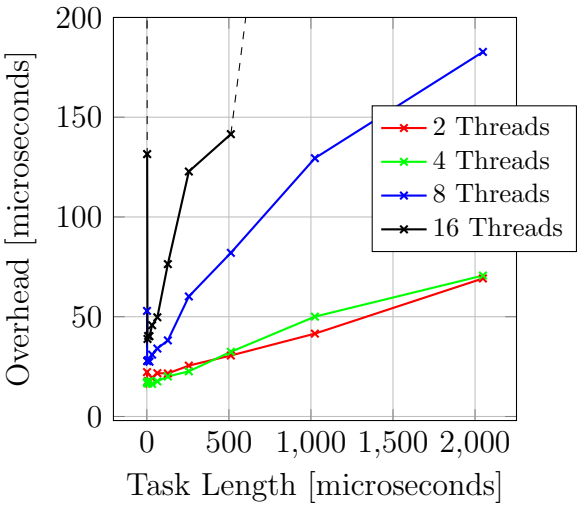


Figure 4.10: *The absolute scheduling overhead of a loop scheduled with (dynamic,1) having 128 iterations is compared using an increasing task length and a different number of threads.*

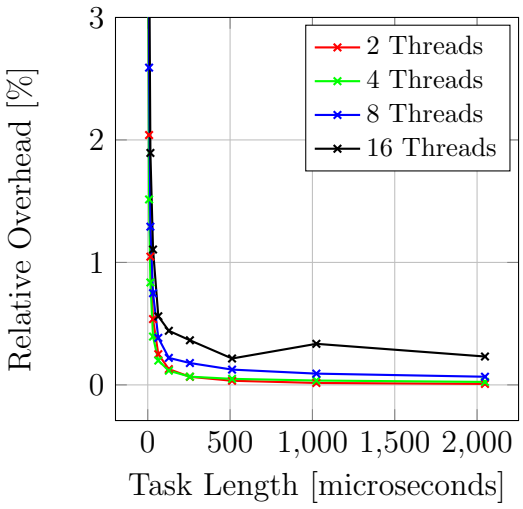


Figure 4.11: *The relative scheduling overhead of a loop scheduled with (dynamic,1) having 128 iterations is compared using an increasing task length. The relative overhead is only illustrated up to 3%. However, for task lengths less than 2 microseconds it is more than 35% using 8 threads and even more than 175% using 16 threads*

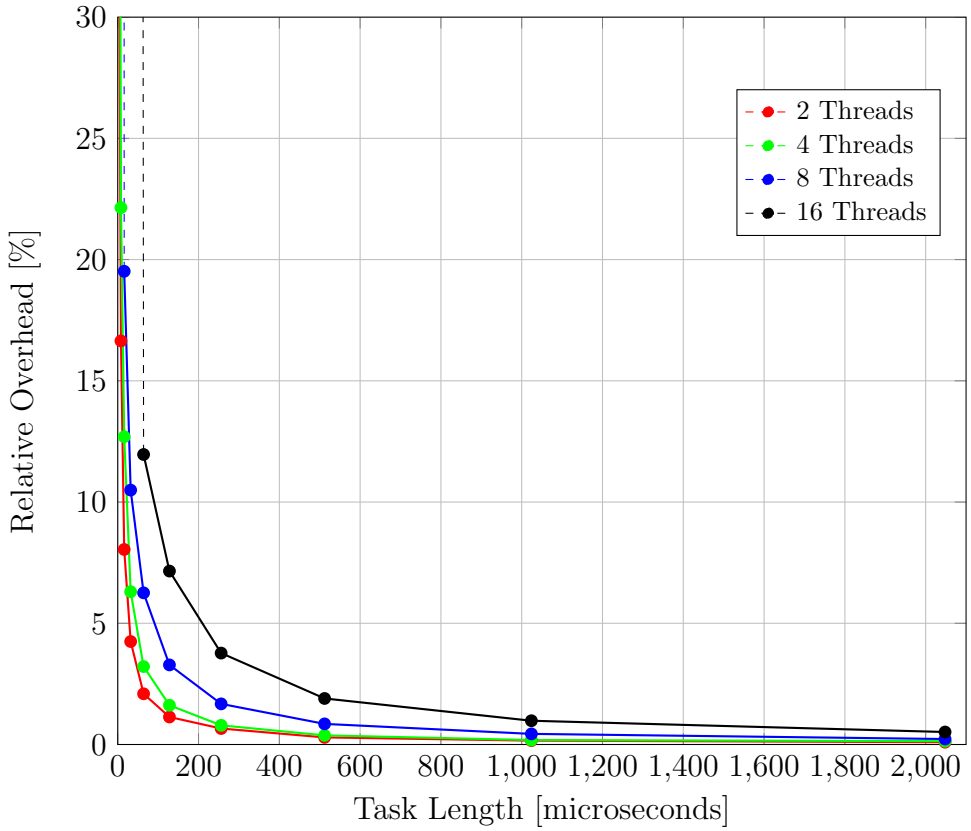


Figure 4.12: *Relative task overhead depending on the length of the tasks.*

benchmark with very big task lengths takes a lot of time, without getting much benefit of this call, because the bigger a task is, the less relevant its overhead is for the speedup prediction. For this purpose a one percent heuristic is introduced: if the overhead of a parallel section is less than one percent of its execution time, it can be neglected.

Task Overhead

Compared to the determination of the loop-scheduling overhead, the determination of the task-scheduling is less time-consuming. For this reason, the one percent heuristic is not used in parallel sections consisting of tasks. Nevertheless, it is avoided to determine the overhead of every single task, especially when a high number of tasks exist within a parallel section.

Consequently the average length of all tasks in a parallel section is calculated. Additionally the variance Var of the task lengths in a parallel sections is calculated using the following formula:

$$Var = \sum_{i \in tasks} (length_{avg} - lengthi)^2 \tag{4.4}$$

The variance Var is the sum of the quadratic difference of the average task length and all task lengths. It is supposed, that the length of most of the tasks in the parallel section is in the range of [avg-Var, avg+Var]. In this range the overhead of ten task

lengths is determined. The task lengths should be distributed equally among the range. Thus, ten supporting points were determined. Each overhead of a task having a length within the range [avg-Var, avg+Var] is interpolated. If the length of a task is outside the range, the overhead is determined using linear extrapolation. As there is only a little number of tasks outside this range, small inaccuracies in the overhead determination using extrapolation can be neglected. Experimental results in Chapter 5 show good prediction results. Moreover, the overhead of only ten tasks is determined per section. Consequently the execution time is not influenced, if a parallel section consists of a high number of tasks.

4.6 Scheduling

In the task graph all dependencies among the tasks are modeled. Consequently, we know which tasks have to wait for the execution of other tasks and which tasks can be executed concurrently. However, we don't know the optimal order of tasks on p threads to minimize the execution time. For this purpose a scheduling algorithm is integrated into the prediction tool. In contrast to sequential scheduling, multiprocessor scheduling is NP-complete [49]. On a single core, the Earliest-Deadline-First-Algorithm determines an optimal schedule for a given set of processes [51], for example. To get a nearly optimal scheduling for multi-core processors with little calculation cost, heuristics have to be used.

Figure 4.13: *A sample task graph including the execution time of each node.*

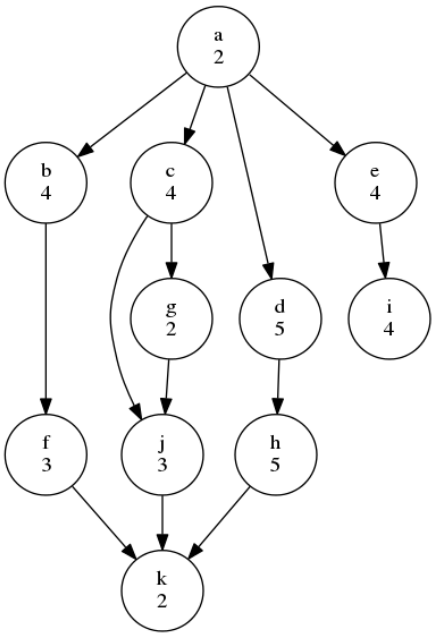


Table 4.3: *Bottom Levels and ALAPs of each node of the task graph shown in Figure 4.13.*

Nodes	bl	ALAP
a	14	0
b	9	5
c	11	3
d	12	2
e	8	6
f	5	9
g	7	7
h	7	7
i	4	10
j	5	9
k	2	12

The scheduling algorithm in the prediction tool has to meet some demands. As different parallel programming models implement different scheduling algorithms, no generic algorithm exists, which can simulate the scheduling realistically for every implementation. Consequently, we follow the goal to provide a scheduling algorithm, which provides a tight upper bound of parallel performance in moderate time. In real applications the number

of tasks, as well as the length of the tasks, is not known during runtime. Thus, usually a dynamic scheduling approach including work-stealing is chosen. In contrast to that, we have all necessary information to calculate an as optimal scheduling as possible due to the profiling and the task graph. Consequently, the Modified Critical-Path (MCP) algorithm [49] is implemented in the prediction tool, because it performs best among different alternatives (ISH (insertion scheduling heuristic) [28]), HLFET (highest level first with estimated times) [10], LAST (localized allocation of static tasks) [2], DLS (dynamic level scheduling) [45], and ETF (earliest time first) [23]) [29]. In addition to the standard MCP algorithm, some extensions considering lock scheduling are implemented in the tool. Moreover, the strategy pattern as shown in Figure 4.14, is implemented in the prediction tool. As a result, further scheduling algorithms can easily be added to the prediction tool to simulate different programming models.

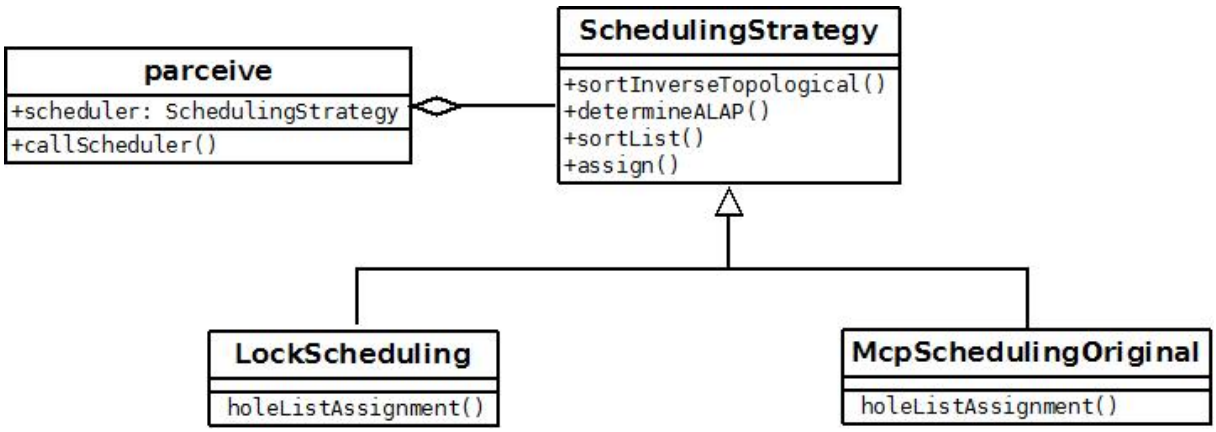


Figure 4.14: *The strategy pattern is implemented in the prediction tool to enable easy addition and exchange of future scheduling algorithms.*

Code 4.13 gives a rough overview of the MCP algorithm. In general the MCP algorithm uses the list scheduling heuristic [47]. Using this heuristic, the nodes of the task graph are sorted according to a priority scheme. Afterwards, each node of the list is assigned to a processor. Usually the processor is chosen which enables the earliest execution time of the task. This skeleton is also used in the MCP algorithm.

But before the MCP algorithm is described in detail, some terms are defined:

- **Bottom Level:** The bottom level $bl(n_i)$ of a node n_i is the length of the longest path to the end starting with this node. This path may include node weights (execution time of a task) and edge weights (e.g. communication costs). For example the bottom level of node c in the sample DAG in Figure 4.13 $bl(c) = 11$, because the longest path to the end is c, g, j, k. Moreover, the recursive definition of the bottom level of node $n_i \in V$ in G = (**V**, **E**, w, c) is

$$bl(n_i) = w(n_i) + \max_{n_j \in succ(n_i)} c(e_{ij}) + bl(n_j) \tag{4.5}$$

- **ALAP:** The as-late-as-possible (ALAP) value $T_L(n_i)$ of a node n_i is its latest possible starting time, so that the overall execution time does not exceed the critical

path length crit.

$$T_L(n_i) = crit - bl(n_i) \tag{4.6}$$

Code 4.13: *Pseudocode of the Modified Path Scheduling [49] including an extension to handle locks.*

```
step 1) create sorted node list
   a) sort graph inverse topological
   b) compute Bottom Levels
   c) compute ALAP
   d) sort list with increasing ALAP
step 2) hole list assignment
```

The bl and ALAP of each node of the graph in Figure 4.13 is shown in Table 4.3.

After having defined these terms, the MCP algorithm can be described as Wu and Gajski did in [50]. According to Code [49], a sorted node list is created in the first step of the MCP algorithm.

For this purpose, the graph is sorted initially in inverse topological order (step 1a). The order is determined using the algorithm in Code 4.14. This algorithm starts with the root node, which has no incoming edges. The root node is added to S, a set of all nodes with no incoming edges. Afterwards, the following steps are repeated, until S is empty: The current element is deleted from S and is added *at the beginning* of the inverse topological sorted list L. For all successors j of the current node i, the edge (i,j) is removed from the graph. If j has no incoming edges anymore, j is inserted into S.

Code 4.14: *Algorithm to sort graph in inverse topological order.*

```
L ← List that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove node n from S
    add n to head of L
    for each node m with edge e from n to m:
        remove edge e from the graph
        if m has no other incoming edges:
            insert m into S
if graph has edges:
    return error (graph has at least one cycle)
else
    return L (a topologically sorted order)
```

In step 1b the bottom level of each node is calculated using the Algorithm in Code 4.15. Starting with a list L in inverse topological order, every element of L is considered one after the other. For each node n_i, the value of $bl(n_i)$ is updated. The new value of $bl(n_i)$

is the node weight $w(n_i)$ added to the greatest bottom level value $bl(n_j)$ of n_i's successors. Additionally, the edge weight between node i and j is added to $bl_(n_i)$. With this step it is possible to extend the prediction model with communication costs in the future.

Using the bottom level, the ALAP of each node is calculated using Equation 4.6 (step 1c). Finally, the list is sorted in step 1d. As sorting heuristic, the ALAP of each node is used and nodes are sorted in ascending order. In case of two or more equal ALAP values, the successors of each nodes are considered and so on.

Code 4.15: *Algorithm to determine the bottom level of each node.*

```
L ← list with inverse topological sorted nodes
for each n_i in L:
   max ← 0
   n_bl_succ(n_i) ← NULL
   for each n_j ∈ succ(n_i):
      if c(e_ij) + bl(n_j) > max:
         max ← c(e_ij) + bl(n_j)
         n_bl_succ(n_i) ← n_j
      bl(n_i) ← w(n_i) + max
```

Now the list is sorted and the nodes have to be assigned to different threads. For this purpose, the MCP algorithm uses a so called hole list assignment in the second step of the scheduling algorithm. In general, tasks are assigned to threads, which allow the earliest execution. The earliest possible execution time of node i depends on:

- The predecessors of i. The latest end time of the predecessors determine the earliest start time of node i.
- The lock release. If i requests a lock l, i can only be executed after l is released.
- The starting time and lengths of free time slots (holes) in the thread schedule. The task can only be assigned to the first fitting hole of a thread.

To illustrate the concept of the hole list, Figure 4.15 represents two threads including already assigned tasks. On CPU1, a task is assigned to time slot 1-2 and another task is assigned to time slot 5-8. Accordingly, the holes of CPU1 - time slots in the schedule of a thread, to which no tasks were assigned yet - are 3-4 and 9-∞. Similarly, the hole of CPU2 is 4-∞. In the program, the holes of each thread are saved in lists. Assume, that another task t with an execution time of three time units should be assigned to a thread and that the earliest possible execution time of t is 2. T does not request any lock. The thread with the earliest execution time greater (or equal) than two is CPU1. However, the hole is not big enough. Consequently task t is assigned to CPU2 with starting time 4. A brief description of an efficient implementation of the hole list assignment can be found in [49].

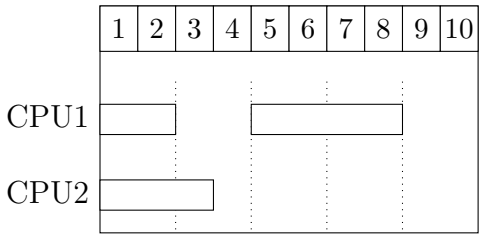


Figure 4.15: *Possible scheduling of three tasks to two threads. Each slot without any assigned task is called hole.*

The theoretical runtime of step 1a and 1b of Code 4.13 is dependent on the number of edges in the graph, thus O(E). Step 1c) is a simple subtraction for each node, which is done in O(N), with N being the number of nodes in the graph. According to [49] the complete sorting in step 1d) has complexity of O(E+Nlog N), if ties are broken by the child that has the smallest ALAP value. A brief description about the complexity of the hole list assignment can also be found in [49]. As the hole list assignment can be done in O(N^2), the complexity of the complete algorithm is O(E+Nlog N+N^2) $\in$ O(N^2).

4.7 Prediction

After the tasks are scheduled, the speedup can be evaluated. The predicted speedup S_p of p cores is

$$S_p = \frac{sequential\ \ execution\ \ time}{predicted\ \ execution\ \ time\ \ of\ \ p\ \ cores} \tag{4.7}$$

The time elapsed between $PAR\_BEGIN()$ and $PAR\_END()$ represents the sequential execution time. In the prediction tool the sequential execution time is determined summing up all intervals of the profiling. This is equivalent to the sum of the execution times of all nodes in the task graph without considering the overhead. The predicted parallel execution time is determined after the scheduling has finished. Each core has the information when to execute which node. Thus, the time of the node which finishes latest on any core represents the predicted parallel execution time. In other words, the slowest thread limits the speedup.

To analyze the scalability of a program, the speedup prediction can be repeated several times using more cores each time. Beginning with two cores, the number of cores is usually doubled each time, until the maximum number of available cores on the target machine is reached.

5 Experimental Results

After the prediction tool is described in Chapter 4, we evaluate its accuracy based on the gained prediction results of four sequential examples. Following, the prediction results are compared to the actual speedup of parallelized versions of the examples. Moreover, the execution time of the prediction tool is analyzed. However, before the experimental results of the prediction tool are presented, the examples are described in detail. The examples are executed on a server containing four AMD Opteron 6168 processors. Each socket has 12 cores and can use a clock frequency of 1.90 GHz. The detailed server configurations can be found in Table 4.2. As the GCC compiler version 4.8.0 is used, the OpenMP specification 3.1 is supported.

5.1 Description of the Examples

To evaluate the prediction tool, different examples are chosen. Each of them has distinct characteristics to highlight different features of the tool.

In total, four examples were chosen: the calculation of the Mandelbrot set, the LU reduction, the determination of a histogram representing the number of different characters in a text file and the recursive quicksort algorithm. Mandelbrot and LU reduction both contain parallelizable loops. In contrast to the mandelbrot example, the loop iterations of LU reduction can get very small. Due to workload imbalance LU reduction represents an example in which inner loop parallelization is most common. This means many forks and joins are executed. Histogram and quicksort do not contain parallelizable loops. In the parallel version, both are parallelized using OpenMP tasks. Histogram additionally contains locks and quicksort contains a recursive function, leading to nested parallelism. An Overview of the examples and their characteristics is given in Table 5.1.

Example	**Characteristics**
Mandelbrot	for-loop
LU Reduction	for loop inner loop parallelization small tasks
Histogram	tasks critical sections
Quicksort	tasks workload imbalance nested parallelism

Table 5.1: *The characteristics of the examples being predicted in this thesis.*

5.1.1 Mandelbrot

The code snippet used in this example draws the visualization of the mandelbrot set [37]. The mandelbrot set is the set of complex values c, for which the recursively defined sequence z_0, z_1, z_2, ... stays bounded. The sequence is defined as

$$z_{n+1} = z_n^2 + c, \quad with \quad z_0 = 0. \tag{5.1}$$

The set is visualized in the complex plane. Points in the set are illustrated with black points in the plane, the other points are colored. The visualization of the mandelbrot set is illustarted in Figure 5.1. This figure is the output of the code snippet of this example shown in Code 5.1.

Code 5.1: *Method to draw the mandelbrot set including annotations.*

```
PAR_SEC_BEGIN("section", true);
   for (int i = 0; i < y_resolution; i++)
   {
      PAR_TASK_BEGIN("task");
      for (int j = 0; j < x_resolution; j++)
      {
         double y = view_y1 - i * y_stepsize;
         double x = view_x0 + j * x_stepsize;
         std::complex<double> Z = 0 + 0 * I;
         std::complex<double> C = x + y * I;
         int k = 0;

         do
         {
            Z = Z * Z + C;
            k++;
         } while (abs(Z) < 2 && k < max_iter);
            if (k == max_iter) {
               //color pixel [i][j] black
            }
            else {
               //color pixel [i][j] with different colors
            }
      }
   PAR_TASK_END();
   }
PAR_SEC_END();
```

The drawing of the mandelbrot set evaluates the most basic features of the prediction tool. A single parallel section is created and a for loop with balanced load is parallelized. A resolution of 480×380 pixel is chosen and the number of different colors used (max_iter) is initially set to 1000. This means, the parallelized loop has 380 iterations (y_resolution).

In Code 5.1 additionally the annotations are inserted. The mandelbrot example consists

of two parallelizable loops. Consequently, the region with both loops is marked as parallel section using PAR_SEC_BEGIN() and PAR_SEC_END(). The content of the outer loop is annotated as tasks. Hence, each iteration of the outer loop can be executed in parallel within the parallel section. As no race conditions occur within this code snippet, no further synchronization using locks has to be annotated.

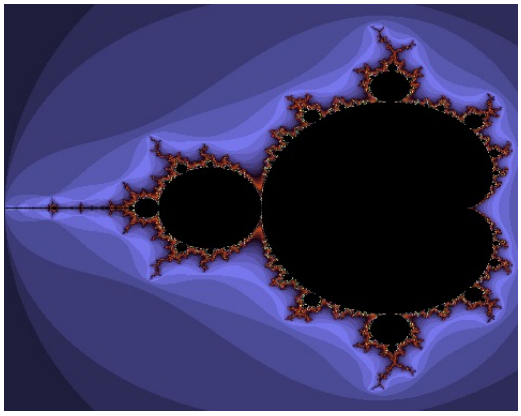


Figure 5.1: *Visualization of the mandelbrot set.*

5.1.2 LU Reduction

LU reduction is a parallel algorithm performing a LU decomposition. The input of the LU decomposition is the matrix M. In the algorithm, the matrix is factored as the product of a lower triangular matrix L and an upper triangular matrix U:

$$M = LU \tag{5.2}$$

The characteristic of a lower triangular matrix is that every element above the diagonal is zero. In an upper triangular matrix every element below the diagonal is zero. For example, the LU decomposition of a 3×3 matrix looks like this:

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

The implementation of the parallel LU reduction benchmark is taken from the OmpSCR benchmark suite [39]. Code 5.2 shows the annotated version of the LU reduction. In this example, the annotations are set similarly to the directives of the OpenMP parallelization. This means, the parallel-directive in the second most outer loop is replaced with PAR_SEC_BEGIN() and each loop iteration is annotated with PAR_TASK_BEGIN() and PAR_TASK_END(). The parallel sections are handled as loops, because the *isLoop*-flag is "*true*" in PAR_SEC_BEGIN(). Consequently, we emulate the loop scheduling strategy dynamic with chunksize of 1 iteration. Due to the parallelization strategy many parallel sections are created. In each section the loop iterations are very small compared to the loop iterations of drawing the Mandelbrot set. The average task length in the Mandelbrot example is about 18 milliseconds and no task is bigger than 5 microseconds. The

smallest task has the length of only a few nanoseconds. For the speedup evaluation, an input matrix with the size 1500×1500 is chosen. Consequently, 1499 parallel sections are created. As the number of loop iterations are getting smaller in the inner most, as well as in the second inner most loop, each created parallel section consists of less tasks and the tasks are getting smaller in every parallel section.

Code 5.2: *LUreduction algorithm including annotations.*

```
// memory allocation of two 2D matrices
double **M = ...;
double **L = ...;

//matrix initialization
...

//LUreduction
PAR_BEGIN();
for(k=0; k<size-1; k++) {
  PAR_SEC_BEGIN("section", true);
  for (i=k+1; i<size; i++) {
    PAR_TASK_BEGIN("looptask");
     L[i][k] = M[i][k] / M[k][k];
       for (j=k+1; j<size; j++)
        M[i][j] = M[i][j] - L[i][k]*M[k][j];
   PAR_TASK_END();
  }
PAR_SEC_END();
}
PAR_END();
```

5.1.3 Histogram

The histogram example determines how often each character occurs in a text file and creates a global histogram out if this information. For this purpose, p different threads are started. It is the intention of each thread to analyze different chunks of the text file until the end is reached. For this purpose, each thread executes an infinite loop. As soon as one thread reaches the end of the text file, the loops terminate at the end of their iteration. To know which chunk of the text each thread has to analyze, two pointers are implemented - a local and a global one. Each thread sets the local pointer to the starting point of the part it is currently analyzing. Additionally the global pointer has to be updated. The global pointer, however, can be accessed from each thread and must be thread-safe. Consequently, the update of the global pointer is synchronized with a lock. Chunks of 16 KB can be analyzed by a single thread at each iteration. Each character is counted and added to a local histogram. At the end of these 16 KB, the global histogram has to be updated, which is synchronized with locks, again. The annotated source code used for the speedup prediction is shown in Code 5.3.

In the speedup prediction, Leo Tolstoy's "War and Peace" [48] is analyzed 100 times in a row to gain reasonable execution times for the evaluation. This results in an text input of 3,291,648,000 Bytes in total. In contrast to the examples in Section 5.1.1 and Section 5.1.2 tasks are parallelized on this example instead of loops. Moreover, we can show the correct functionality of the lock scheduling with this example.

Code 5.3: *Histogram algorithm including annotations for parallel speedup prediction.*

```
int cont = 1;
int lock = 1;
int lock2 = 2;
PAR_SEC_BEGIN("sec1", false);
  for (;;) {
  PAR_TASK_BEGIN("t1");
    char* chunk = 0;

    PAR_LOCK_BEGIN(&lock);
    if (*gbuffer == TERMINATOR) {
      cont = 0;
    } else {
      chunk = gbuffer;
      gbuffer += chunk_size;
    }
   PAR_LOCK_END(&lock);
    if (cont) {
      unsigned int* local_histogram = calloc ...;

      for (unsigned int i=0; i < arg->chunk_size; i++) {
        if (chunk[i] == TERMINATOR) {
          cont = 0;
        break;
        }
        if (chunk[i] >= 'a' && chunk[i] <= 'z')
          local_histogram[chunk[i]-'a']++;
        else if(chunk[i] >= 'A' && chunk[i] <= 'Z')
          local_histogram[chunk[i]-'A']++;
      }
      PAR_LOCK_BEGIN(&lock2);
        for (int i=0; i<NALPHABET; ++i)
          global_histogram[i] += local_histogram[i];
      PAR_LOCK_END(&lock2);
    }
   PAR_TASK_END();
   if (!cont)
    break;
   }
PAR_SEC_END();
```

5.1.4 Quicksort

The fourth example is the quicksort algorithm. Given an array of integer numbers, the algorithm returns a sorted array. The algorithm chose a pivot element in the array randomly. Then all elements smaller than the pivot element are saved before the pivot element in the array and all elements bigger than the pivot element are saved after the pivot element. The pivot element separates the array into two new arrays. The new arrays are the input of a recursive function call, to sort the two new arrays. These steps are repeated until all arrays are in the correct order - or in other words, until all new arrays have a size of maximum one.

The recursive function calls can be executed in parallel. Each recursive step, however, needs the sorted array of calling function. Executing quicksort in parallel can lead to a slowdown due to a high scheduling overhead relative to the task lengths, if the unsorted arrays are very small. For this reason, the quicksort algorithm presented in Code 5.4 sorts all arrays with more than 10,000 elements in parallel. If arrays have less than 10,000 elements, however, no new task is created and the array is sorted in the currently executing thread using a sequential quicksort implementation. A task graph of the quicksort algorithm is presented in Figure 5.2. The task graph illustrates the dependency between each recursion level. Additionally, the length of the tasks is illustrated by the area of the nodes. Moreover, it can be seen, that the length of the tasks decreases with each recursion step. In the last (parallel) recursion level, however, the task length is bigger than before, because the sequential sorting of the small arrays is executed in the same task from which the sequential recursion was called.

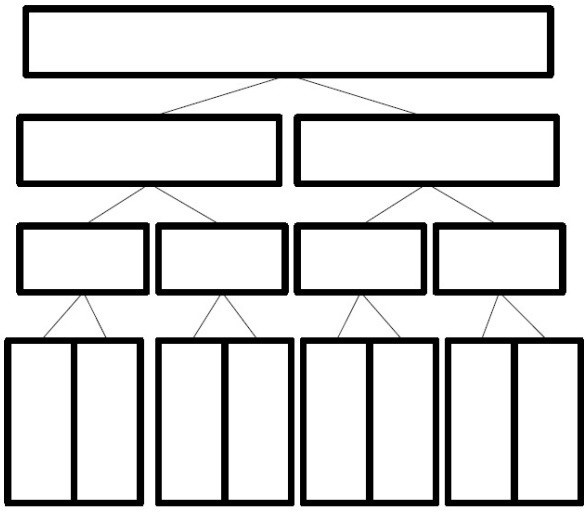


Figure 5.2: *This figure shows the task graph of the quicksort algorithm. The area of the nodes represent the task lengths. Each level represents a recursion step. At the end of each recursion step, the new created tasks have to wait until the calculations of the last recursion step are finished.*

In the main method of Code 5.4, the first call of the recursive quicksort function is surrounded with PAR_SEC_BEGIN() and PAR_SEC_END() to annotate the parallel region.

Consequently, each recursive call of the algorithm is executed within the parallel section. In the quicksort_par() function, the sorting of the element to the appropriate place relative to the pivot element is not annotated to ensure that the function calls which need the sorted array as input are definitively called afterwards. Additional to this synchronization, one of the two recursive functions in the quicksort_par() function has to be annotated as task to allow the parallel execution to the other recursive function call.

This example is difficult to predict, because all the tasks have different lengths. Moreover, nested parallelism occurs. Consequently, this example shows the correct scheduling of nested tasks and evaluates how accurate the overhead for nested tasks is chosen in the prediction model.

Code 5.4: *Annotations in the Quicksort algorithm.*

```
main(){
   ...//initialization
   PAR_BEGIN();
   PAR_SEC_BEGIN();
      quicksort_par(array, 0, a.length-1);
   PAR_SEC_END();
   PAR_END();
}

void quicksort_par(int *a, int left, int right){
   if(right-left < 10000){
      quicksort_seq(a, left, right);
   }
   else{
      if (left < right){
         //swap elements, so that all elements smaller than the pivot have
            smaller array index and all elements bigger than the pivot
            have greater array index
         int index = ...; //current array index of the pivot element
         PAR_TASK_BEGIN();
         quicksort_par(a, left, index - 1);
         PAR_TASK_END();
         quicksort_par(a, index + 1, right);
      }
   }
}

void quicksort_seq(int *a, int left, int right){
      if(left < right){
         // ...
         quicksort_seq(a, left, index - 1);
         quicksort_seq(a, index + 1, right);
      }
}
```

5.2 Prediction Results

The speedup prediction results are presented in this section. For this purpose, the speedup of the examples presented in the previous sections is determined executing them in parallel using 2, 4, 8 and 16 threads. In the parallel runs, the OpenMP threads are pinned to cores in such a way, that a minimum number of different sockets has to be used. This means, that parallel programs executed with less than 12 cores are executed on a single socket. The speedup is compared to the speedup predicted by the tool for the same number of cores.

Measurements of the execution time were also made for parallel programs on 32 threads. However, the execution time measured for 32 cores has a high deviation. This may be the result of NUMA effects of the underlying hardware, which are higher compared to the measurements using 16 threads. As the 16 threads are distributed on two processors neighbored to each other, NUMA effects are less. The prediction tool does not simulate NUMA effects. Consequently, examples including strong NUMA effects cannot be predicted accurately.

5.2.1 Mandelbrot

In Figure 5.3 the experimental results of the mandelbrot example are illustrated. The applications scales well up to 16 cores and the real speedup, as well as the predicted speedup, is near the optimum.

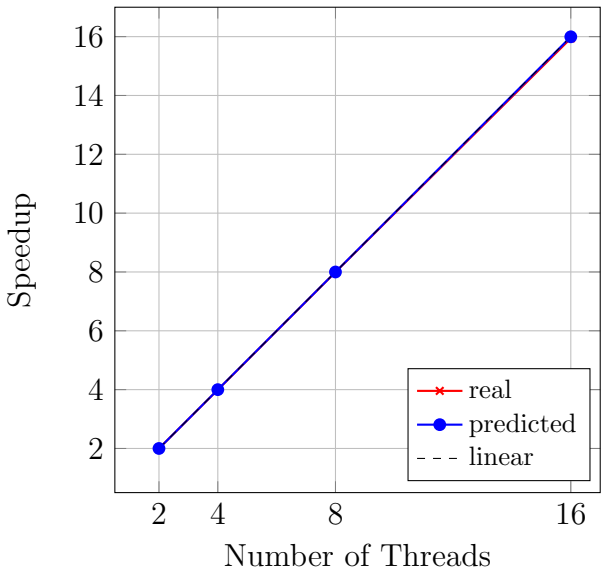


Figure 5.3: *Comparison of the real speedup and the predicted speedup of the mandelbrot example. Both are near the linear speedup.*

Due to the characteristics of the example, this behavior was expected. The loop, which is parallelized has no workload imbalance and the loop iterations are big enough, so that the parallel overhead is insignificant. As a result OpenMP can schedule the iterations optimal

to the threads with little overhead. This behavior is predicted and simulated precisely in the prediction tool. The MCP-scheduling assigns the tasks equally to different threads. Moreover, the prediction tool calculates an upper bound for the task length. If the tasks are bigger than this bound, the scheduling overhead is less than 1% and consequently the overhead is neglected in the prediction tool. In the mandelbrot example this bound is 200 microseconds for two threads, 400 microseconds for four threads, 800 microseconds for eight threads and 1600 microseconds for 16 threads. In the speedup prediction of the mandelbrot example no overhead is considered, because the average task length of 19 milliseconds is smaller than the bound. As a result, the prediction is very close to the actual speedup. The predicted speedup exceeds the real speedup for at most 0.3% on 16 threads. Most likely, the difference is the overhead, which occurs on real machines, but which is neglected in the prediction due to execution time optimizations. These results nicely show the ability of the prediction tool to estimate the speedup of parallel applications with uniform and coarse-grained tasks.

5.2.2 LU Reduction

The LU reduction example has a high number of parallel sections and different task lengths. More concretely, 1499 parallel sections are created, each having a decreasing number of tasks with descending length. The main challenge in this example is the correct prediction of very small tasks in a parallel inner loop. A task length of about 4.7 microseconds for the loop iterations in the first parallel section was measured. With each iteration of the outer loop, the tasks in the inner loop decrease. The smallest length of a task was measured with only 12 nanoseconds.

Figure 4.11 shows that the relative overhead of very small tasks is very high. Tasks smaller than one microsecond can have am overhead larger than 100% relative to the task length. Using 16 threads the overhead is more than 100 times higher than using 2 threads. As a result, the red graph in Figure 5.4 confirms the assumption, that this application has a limited parallel performance and doesn't scale well.

The prediction tool determines the overhead as it is shown in Figure 5.5. In each parallel section, the average task length is calculated and the appropriate overhead is determined. The task lengths and the number of loop iterations differ in the parallel sections. With increasing k in the outer loop, the task lengths and the number of inner loop iterations decrease. In the figure it can be seen that especially for small tasks a high overhead is estimated. Moreover, the overhead using 16 threads is multiple times higher than the overhead using 2, 4 or 8 threads.

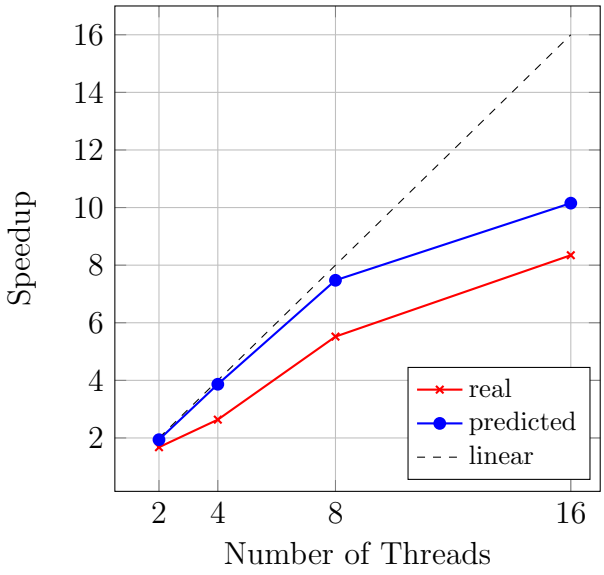


Figure 5.4: *Comparison of real speedup and predicted speedup of the LU reduction example.*

The parallel behavior of the of the LU reduction is predicted with an inaccuracy of at least 15.6% (using two threads) and at most 46.6% (using four threads). Using eight threads, the inaccuracy is 35.4% and using 16 threads it is 21.7%. Compared to the prediction results of the other examples the deviation from the real speedup is higher. A reason for this is the inaccuracy in determining the overhead of the smallest tasks that occur during execution. Even very small measurement errors have a higher relative effect on tasks of small length, than on tasks of big length. For example, an overhead measurement error of 200 nanoseconds can be neglected at tasks having a length of one millisecond - the error is only 0.02%. The error, however, is 20% at a task having a length of one microsecond. The result of the measurement error is some variance in the speedup prediction. In this example single runs exist with predictions varying up to 30% from the average value.

Another reason for the comparatively high deviation from the real speedup may be the effect of an inner-loop parallelization. In Section 4.5 it is explained that there is no permanent new thread creation, as OpenMP holds a thread pool. However, at every joining point an implicit barrier leads to synchronization. As a result of very small and a very high number of parallel sections, this synchronization overhead has some influence on the parallel performance.

As it is a lot of additional effort to determine the synchronization overhead correctly, we do not consider this overhead in the model of the prediction tool to keep the balance between execution time and accuracy.

5.2.3 Histogram

The example determining the histogram of characters occurring in a text file is parallelized using OpenMP tasks and consists of two critical sections in each iteration, which are be synchronized using locks. In contrast to the overhead of loop scheduling, which occurs in the Mandelbrot and LU Reduction example, the task scheduling overhead is

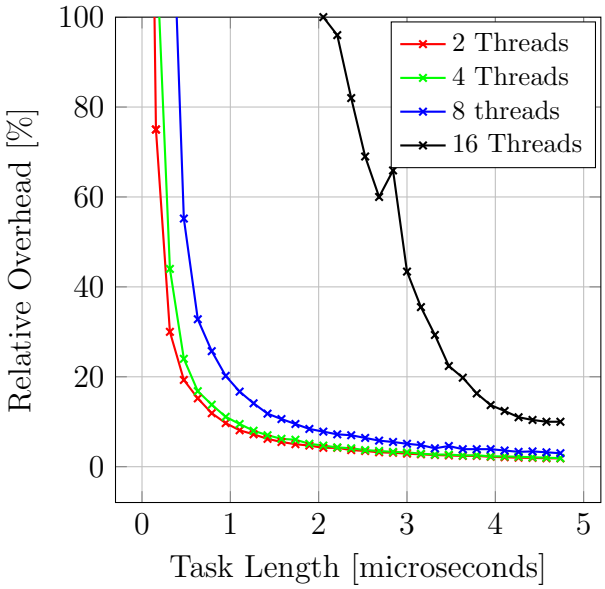


Figure 5.5: *The overhead determined in the prediction of the LU Reduction depending on different number of threads.*

determined in this example. Moreover, according to reasons explained in Section 4.5, lock overhead is neglected. In this context, the lock overhead describes the time to request and release the lock. The time elapsed while waiting for another lock being released is of course considered in the prediction. One of the main purposes of this example is the demonstration of correct speedup prediction in the presence of critical sections.

Figure 5.6 compares the predicted speedup with the real speedup on different number of threads. The application scales well and the speedup is only a bit lower than the optimal speedup. Neglecting overhead and locks, an optimal speedup is possible, due to theoretically equal sized tasks. Usually locks lead to a decrease of speedup, because concurrent tasks have to wait until critical sections can be executed. In this example, however, the execution time of the critical sections is less than 0.1% of the execution time of a single task. As a result, these locks have only little influence on the parallel performance.

The real speedup of this application is 1.896 on two threads, 3.774 on four threads, 7.439 on eight threads and 14.052 on 16 threads. The deviation from linear speedup is mainly caused by the task scheduling overhead.Comparing the predicted speedup with the real speedup, a maximum deviation of 6.1% is determined. Thus, this result shows that the tool can predict applications parallelized with OpenMP tasks including critical sections with high accuracy. Moreover, the increase of task overhead on different number of threads is also simulated correctly. Figure 5.7 illustrates the determined task overhead on different number of threads. The parallel run of the application consists of approximately 60,000 tasks, each having a length of roughly about 80 microseconds (which is about 160,000 clock cycles). For a task length of 80 microseconds, the prediction tool determines a task overhead of 1.44 microseconds executed with two threads. This is 1.8% of the task length. Executed on four threads, the overhead is 2.26%. The overhead increases to 3.75% on 8

threads and up to 5.91% on 16 threads.

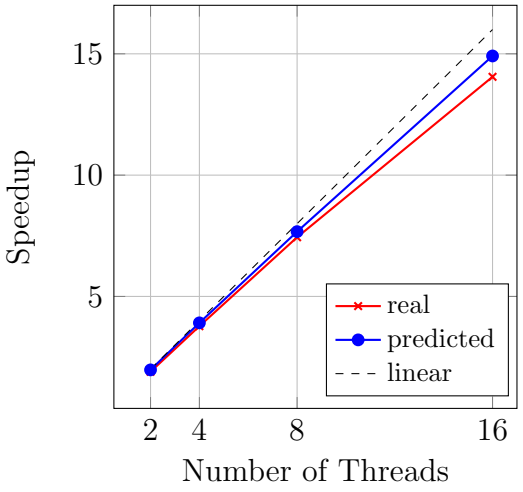


Figure 5.6: *Comparison of the real speedup and the predicted speedup of the histogram example.*

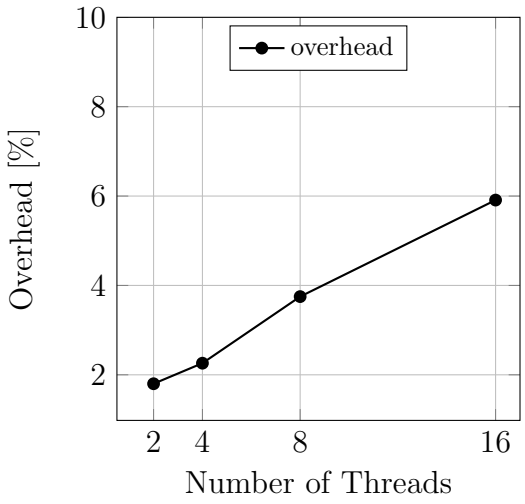


Figure 5.7: *The estimated task overhead of the histogram example in percent, depending on the number of threads.*

5.2.4 Quicksort

The quicksort algorithm represents an example for nested parallelism with many tasks of different lengths. It is predicted by the tool with an inaccuracy of less than 3%, which can be seen in Figure 5.8. To achieve this accuracy, the scheduling must consider the dependencies correctly. This means, it must be considered that the elements of the array in the function quicksort_par() of Code 5.4 are sorted correctly relative to the pivot element before the recursive function calls are executed. Due to these sequential parts of the algorithm, it does not scale linear. This is shown with the black graph in Figure 5.8, which illustrates the speedup determined by the prediction tool considering no overhead.

The overhead has to be considered, however, to achieve higher accuracy. In the quicksort example many tasks exist with different task lengths, which is illustrated in the histogram in Figure 5.9. This figure gives an overview of the distribution of the task lengths. In the parallel execution of the quicksort usually a few big and a lot of small tasks exist. In this example, most of the tasks have a length of about 70 to 9100 microseconds. The biggest occurring task has a length of nearly 1 millisecond. In total, 1751 tasks are created. The overhead of all these tasks is determined using the inter- and extrapolation approach presented in Section 4.5. For this purpose, the average task length of 2600 microseconds and the variance of 6500 microseconds is determined. Consequently, ten overhead values are determined in the range between 0 and 9100 microseconds, which act as supporting points. In Figure 5.10, the supporting points are illustrated with gray lines. This figure shows the determined overhead for the quicksort example regarding different task lengths. Continuous lines represent interpolated values and dashed lines represent extrapolated values. Interpolated overhead of tasks usually has high accuracy, because the lengths of the tasks are usually close to the supporting points. As linear extrapolation

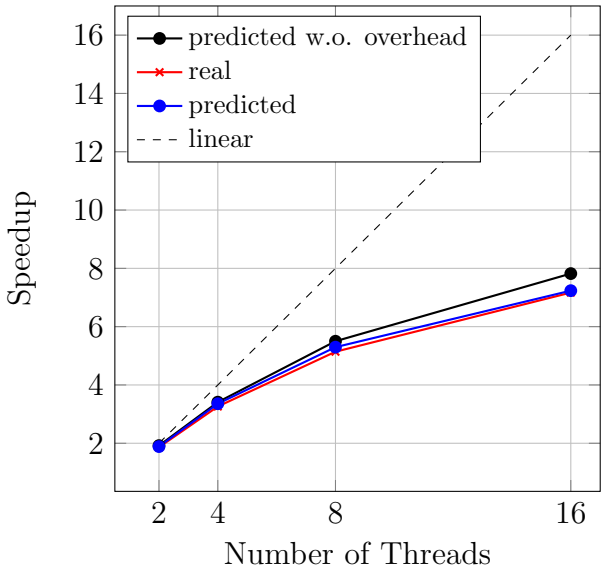


Figure 5.8: *Comparison of the real speedup and the predicted speedup of the quicksort example with and without overhead.*

is used, extrapolated overhead values can somewhat deviate from the real overhead. However, this inaccuracy has little influence on the accuracy of the overall speedup prediction. As it can be seen in Figure 5.8, most of the tasks have a length of 70 up to 9100 microseconds. Less than 5% of the tasks are bigger than 9100 microseconds. Moreover, Figure 5.11 shows, that the overhead of tasks being bigger than 9100 microseconds is smaller than 0.7% on 16 threads and even smaller than 0.2% on less than 16 threads. Consequently the speedup prediction is influenced mostly by the interpolated overhead values. Considering this overhead leads to the high accuracy in speedup prediction, as it is shown in Figure 5.6.

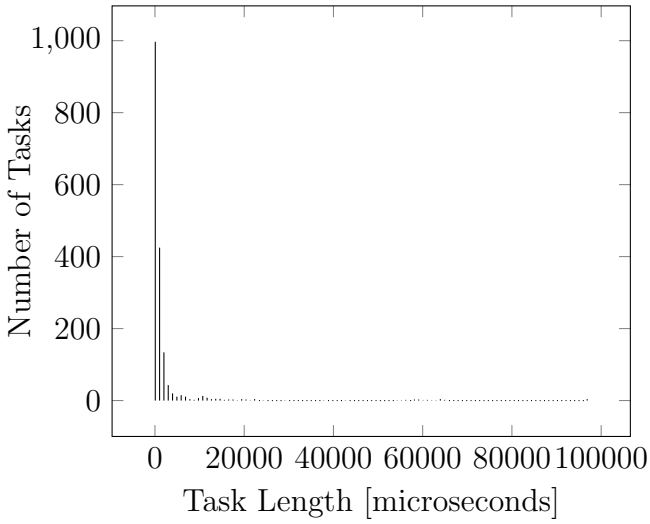


Figure 5.9: *Distribution of the task lengths in the quicksort algorithm. Most of the tasks have a length between 70 microseconds and 1000 microseconds. The average task length is approximately 2600 microseconds.*

The high accuracy of the prediction of the quicksort example confirms several assumptions. First of all, the prediction model simulates nested parallelism correctly. Moreover, the model to predict the task overhead can also be used to predict the overhead of nested tasks, as it delivers sufficient accuracy. Even for a large number of tasks with different lengths, the overhead can be determined accurately. This is one of the main reasons to predict the speedup with an inaccuracy of less than 3%. Finally, the quicksort example confirms that the approach to determine the task overhead using inter- and extrapolation is chosen well.

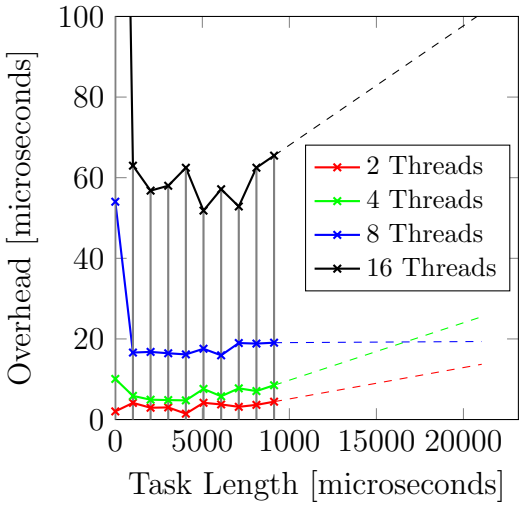


Figure 5.10: *Estimated overhead of the quicksort example. The ten supporting points are marked with gray lines. Continuous lines represent the interpolated overhead and dashed lines represent the extrapolated overhead.*

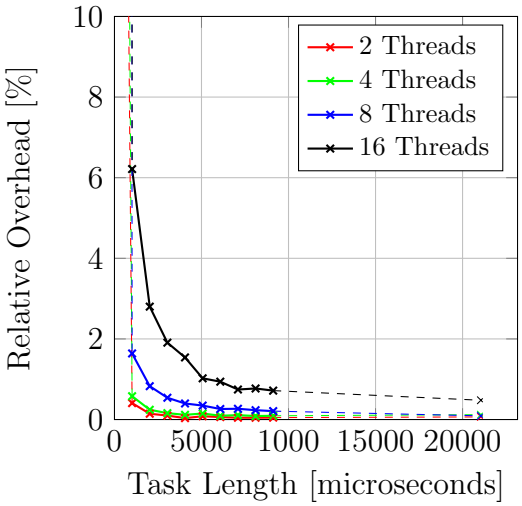


Figure 5.11: *The relative overhead of the quicksort example regarding the task length.*

5.2.5 Comparison of the Results

Finally, all the prediction results are summarized. For this purpose Table 5.2 gives an overview of the real speedup, as well as the predicted speedup of each example.

Threads		2	4	8	16
Mandelbrot	real	1.999	3.994	7.994	15.952
	prediction	2	3.99	7.999	15.995
LU Reduction	real	1.678	2.6353	5.5197	8.345
	prediction	1.942	3.865	7.475	10.153
Histogram	real	1.896	3.774	7.439	14.052
	prediction	1.973	3.911	7.671	14.912
Quicksort	real	1.864	3.261	5.140	7.169
	prediction	1.886	3.336	5.296	7.234

Table 5.2: *An overview of the measured speedup results of all examples.*

All of the examples used for the evaluation of the program are predicted with a tight upper

bound. The maximum prediction error occurs in the estimation of the speedup of the LU reduction. This example is predicted with an inaccuracy of 15.6% to 46.6%. Due to an inner loop parallelization a high number of parallel sections are created in this example, each having tasks with a very small length. Measuring errors in the estimation of the overhead for the prediction of very small tasks, as well as the neglect of synchronization overhead leads to the prediction inaccuracy of this result. We accept this, because the inaccuracy is the result of a tradeoff between execution time and accuracy. Moreover, the prediction tool is created for the prediction of a higher abstraction level, which usually doesn't has such small tasks. Three out of four examples, however, are predicted with a very high accuracy. None of the speedup predictions of the mandelbrot calculations, the determination of the character histogram and the quicksort algorithm differ more than 6.1% from the real speedup. We can predict these examples with an average prediction inaccuracy of 2.1%.

On the whole, the experimental results demonstrate that we have chosen the scheduling algorithm appropriately in the prediction tool and that we determine the overhead with high accuracy using different heuristics. Consequently the tool predicts the speedup of different kind of applications including workload imbalance, critical sections and nested parallelism accurately.

5.3 Execution Time of the Prediction Tool

The main focus of this thesis is the optimization of the speedup accuracy of the prediction tool. However, the slowdown caused by the tool compared to the sequential execution time of the user applications should be as small as possible. As the input code must be executed once for profiling, the prediction process has at least the same execution time as the input code. But additionally creating a task graph, determining the overhead and scheduling all the tasks lead to a longer execution time of the process compared to the input application. The factor, which the tool including the execution of the instrumented program runs slower than the sequential version of the input program, is called slowdown.

In this section the slowdown of the prediction tool is determined and the execution time is analyzed in detail. The overall execution time of the prediction tool can be divided into four parts:

- profiling
- task graph creation
- overhead determination
- scheduling

Table 5.3 gives an overview of the execution times of the single parts in the prediction tool determined utilizing the example applications. Each row shows the execution time for the prediction of an example with a specific number of threads. The columns indicate which part of the prediction tool is regarded. Thus the execution times for of the profiling, the

task graph creation, the overhead determination and the scheduling can be compared for different number of threads. Additionally, the slowdown is shown in the last column.

		Profiling	Create Task Graph	Determine Overhead	Scheduling	slowdown
Histogram	2 Threads	5722 ms	336 ms	885 ms	14151 ms	4.15
	4 Threads		352 ms	902 ms	12945 ms	3.92
	8 Threads		346 ms	885 ms	16225 ms	4.56
	16 Threads		333 ms	890 ms	14490 ms	4.22
LU Reduction	2 Threads	7838 ms	2303 ms	89361 ms	60013 ms	43.76
	4 Threads		2281 ms	97741 ms	178342 ms	79.58
	8 Threads		2222 ms	121650 ms	419978 ms	170.31
	16 Threads		2242 ms	271405 ms	527842 ms	210.30
Mandelbrot	2 Threads	7333 ms	0.7 ms	159 ms	0.9 ms	1.02
	4 Threads		0.7 ms	223 ms	1.2 ms	1.03
	8 Threads		0.7 ms	223 ms	1.5 ms	1.04
	16 Threads		0.7 ms	338 ms	1.7 ms	1.05
Quicksort	2 Threads	1633 ms	3.6 ms	1265 ms	5.5 ms	1.78
	4 Threads		3.6 ms	2382 ms	6.4 ms	2.47
	8 Threads		3.8 ms	5820 ms	12.8 ms	4.59
	16 Threads		3.7 ms	28260 ms	29.9 ms	18.39

Table 5.3: *This table gives an overview of the execution times of the profiling, the task graph creation, the determination of the overhead and the scheduling of different examples in the prediction tool in microseconds. Additionally the slowdown of each example is shown.*

It cannot be made a general statement about the percentage of the execution time in each of this parts, because the execution time strongly depends on the the input application. Moreover, the slowdown of the prediction tools depends on the input application, too. Whereas the Mandelbrot example can be predicted with an slowdown of 1.05 in the worst case, the prediction of the LU reduction can take more than 210 times the execution time of the sequential input application. For all the examples, however, the time spent to create the task graph is less than 2% and therefore has only little influence on the overall execution time.

In the mandelbrot example, usually more than 96% of the execution time is spent in the profiling (compare Figure 5.12). Consequently, the further prediction steps have little influence on the slowdown. As the profiling needs the same time as the sequential execution of the input application, the prediction process of the mandelbrot example only has a slowdown of 2-5%.

In Figure 5.13 it is illustrated that in the LU reduction, the influence of the overhead determination on the overall execution time is between 22 and 54%. The influence of the scheduling is between 40 and more than 70% using more than 4 cores. In general, the

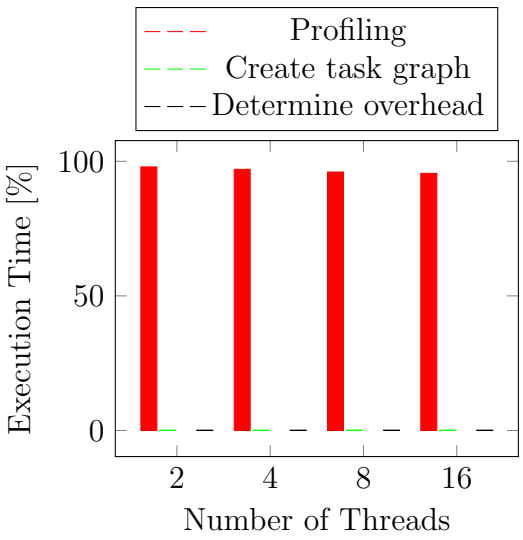


Figure 5.12: *Distribution of the execution time of the mandelbrot example in the single parts of the program in %.*

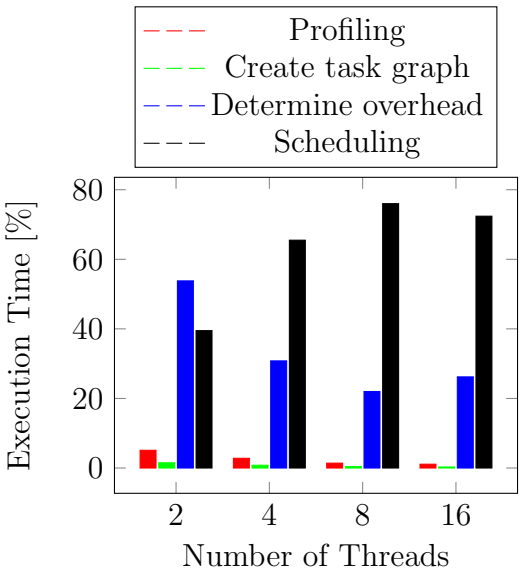


Figure 5.13: *Distribution of the execution time of the LU reduction in the single parts of the program in %.*

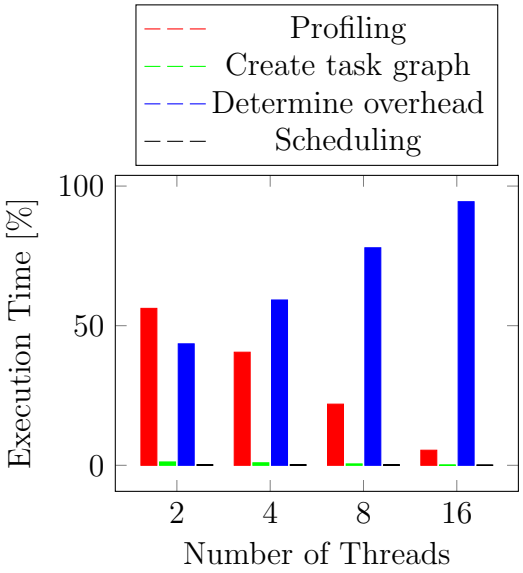


Figure 5.14: *Distribution of the execution time of the quicksort algorithm in the single parts of the program in %.*

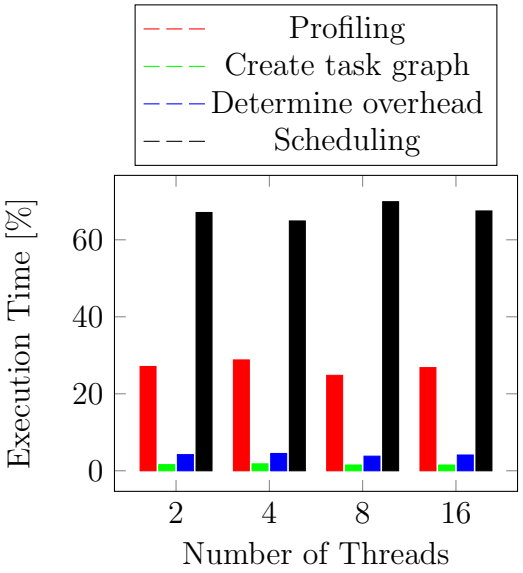


Figure 5.15: *Distribution of the execution time of the histogram example in the single parts of the program in %.*

time to determine the overhead and the scheduling is comparatively high in this example. This leads to a slowdown of approximately 44 using 2 threads. But the time to determine the overhead and the scheduling increases up to 880% of the initial value using more number of threads. Consequently, the slowdown increases, too. It is factor 79.6 using four threads, 170.3 using eight threads and 210.3 using 16 threads.

The slowdown factor in quicksort is at most 18.4 using 16 threads. However, using less threads, it is between 1.78 and 4.59. The reason for the high slowdown using 16 cores is the increase of overhead determination time of more than 220%. In Figure 5.14 the influence of the overhead determination of 43.5 up to 95.4% can be seen.

In contrast to that, Figure 5.15 shows that in the histogram example, the scheduling has the most influence. The influence is between 65 and 70%. As the scheduling time stays constant using more than 2 threads, the slowdown is also constant. For all prediction runs on different number of cores, the slowdown factor is between 3.9 and 4.56.

With the exception of the histogram example an increasing scheduling time is observed. The reason therefore is the increasing scheduling effort of the algorithm. A bigger number of holes have to be traversed in the lists of the MCP algorithm to find the optimum scheduling for a bigger number of threads. In contrast to that, due to the heuristic used for scheduling critical sections, the execution time of the scheduling does not increase in the histogram example.

Especially in the LU reduction and the quicksort algorithm, the overhead determination increases quite fast. This is the result of estimating very small tasks, which have an overhead bigger than 100%. Additionally, the overhead determination approach repeats the measurements several time to gain accuracy. Consequently, the execution time of the overhead determination exceeds the length of the tasks multiple times.

6 Conclusion

Program parallelization is an error-prone, time consuming and expensive task. To ensure that it is worth the effort, there is a high demand for developer tools. These tools should assist the parallelization process and estimate the parallel performance potential of a sequential program. To assist the developer, we propose a speedup prediction tool in this thesis. It requires an annotated serial source code as input and predicts a tight upper bound of the parallel speedup on an arbitrary number of cores. Additionally, the scalability of the input programs can be estimated. The tool supports the prediction of programs consisting of parallel loops, as well as parallel tasks and can handle workload imbalance, critical sections and nested parallelism.

For the realization of the prediction tool we propose a concept in this thesis. Hence, the proposed annotations are defined. Annotations are used to mark parallel sections, concurrent tasks and critical sections and moreover synchronization can be expressed using implicit barriers. Using different annotation parameters, we also introduce the possibility to differ between tasks and loops in the source code to adapt the prediction approach accordingly. The annotated source code is profiled and a task graph is created using the information from the profiling. Consequently, many different program workflows can be represented. In contrast to Parallel Prophet, we can predict more complex programs using this task graph. Following, the parallel behavior is simulated. For this purpose, the tasks of the task graph are scheduled using the modified critical path scheduling algorithm. The prediction tool is designed, so that the scheduling algorithm can be changed during runtime easily. In the scheduling, the parallel overhead is considered. We implemented a time saving approach to determine the overhead in the prediction tool according to different parameters, like number of threads, task length, number of tasks and type of tasks.

The prediction tool is evaluated by several examples, which have different characteristics including nested parallelism, workload imbalance and critical sections. Only the LU reduction example has an inaccuracy of 15.6% to 46.6%. Frequent inner loop parallelization and very small tasks lead to this inaccuracy. Three of four examples, however, are estimated with an inaccuracy of less than 6%. The average prediction inaccuracy is only 2.1%. Consequently, we present a tool with accurate prediction. Especially the speedup of the quicksort algorithm, an example for nested parallelism, which is usually difficult to predict, is estimated with a deviation of less than 3%. Consequently, we improved problems in predicting nested parallelism in related work with the prediction tool of this thesis.

Finally, the execution time of the prediction tool is analyzed. The execution time is dependent on the input program and the number of predicted cores. 69% of the prediction runs have a slowdown of less than 4.60. Most of the time of the prediction is spent in

the overhead determination and in the scheduling. The prediction tool was designed to keep the balance between high prediction accuracy and little slowdown. However, single methods of the tool are not optimized. In future development steps the slowdown can be minimized by optimizing the overhead determination routine. Especially the overhead estimation of very small tasks can be improved.

Apart from that, programs influenced by memory effects could be predicted accurately implementing a memory simulation. As there is a high prediction accuracy in this thesis, however, the most benefit can be gained improving the usability of the tool and therefore support the developer to use the program faster and more easily. For this purpose, the determination of parallel sections and critical sections can be supported by other tools. Semi-automatic insertion of annotations would further assist the developer during parallelization and would decrease the time investment to use this prediction tool. Moreover, the prediction of further programming models, like Intel TBB, can be supported in future development steps.

Bibliography

[1] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. ACM, 1967.

[2] Jeff Baxter and Janakh Patel. The LAST Algorithm- A Heuristic-Based Static Task Allocation Algorithm. In *International Conference on Parallel Processing, University Park, PA*, 1989.

[3] William Blume, Ramon Doallo, Lawrence Rauchwerger, Peng Tu, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, et al. Parallel Programming with Polaris. *Computer*, (12), 1996.

[4] Richard P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *Journal of the ACM*, 21(2), 1974.

[5] Greg Bronevetsky, John Gyllenhaal, and Bronis R. De Supinski. CLOMP: Accurately Characterizing OpenMP Application Overheads. In *OpenMP in a New Era of Parallelism*. Springer, 2008.

[6] Edouard Bugnion, Shih-Wei Liao, Brian R. Murphy, Saman P. Amarasinghe, Jennifer M. Anderson, Mary W. Hall, and Monica S Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *Computer*, (12), 1996.

[7] J Mark Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *Proceedings of First European Workshop on OpenMP*, volume 8, 1999.

[8] J. Mark Bull and Darragh O'Neill. A microbenchmark suite for OpenMP 2.0. *ACM SIGARCH Computer Architecture News*, 29(5), 2001.

[9] J. Mark Bull, Fiona Reid, and Nicola McDonnell. A Microbenchmark Suite for OpenMP Tasks. In *OpenMP in a Heterogeneous World*. Springer, 2012.

[10] K. Mani Chandy, Thomas L. Adam, and J. Dickson. A Comparison of List Scheduling for Parallel Processing Systems. *Communication of ACM*, 17, 1974.

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization*. IEEE, 2009.

[12] Vassilios V. Dimakopoulos, Panagiotis E. Hadjidoukas, and Giorgos Ch. Philos. A Microbenchmark Study of OpenMP Overheads under Nested Parallelism. In *OpenMP in a New Era of Parallelism*. Springer, 2008.

[13] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing*, 10(4), 1990.

[14] Michael J. Flynn. Some Computer Organizations and their Effectiveness. *IEEE Transactions on Computers*, 100(9), 1972.

[15] Ian Foster. Designing and Building Parallel Programs, 1995.

[16] Saturnino Garcia. *A practical Oracle for Sequential Code Parallelization*. PhD thesis, University of California, San Diego, 2012.

[17] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. Kremlin: Rethinking and Rebooting Gprof for the Multicore Age. In *ACM SIGPLAN Notices*, volume 46(6). ACM, 2011.

[18] Urs Gleim and Tobias Schüle. *Multicore-Software: Grundlagen, Architektur und Implementierung in C/C++, Java und C*. dpunkt.verlag, 2012.

[19] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5), 1988.

[20] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The Cilkview Scalability Analyzer. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2010.

[21] Alexander Herz and Chris Pinkau. Real-World Clustering for Task Graphs on Shared Memory Systems. In *Job Scheduling Strategies for Parallel Processing*. Springer, 2014.

[22] Mihai Horoi and Richard J. Enbody. Using Amdahls Law as a Metric to Drive Code Parallelization: Two Case Studies. *International Journal of High Performance Computing Applications*, 15(1), 2001.

[23] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling Precedence Graphs in Systems With Interprocessor Communication Times. *SIAM Journal on Computing*, 18(2), 1989.

[24] Intel Corporation. Intel parallel advisor. http://software.intel.com/ en-us/articles/intel-parallel-advisor/.

[25] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: Parallel Speedup Estimates for Serial Programs. In *ACM SIGPLAN Notices*, volume 46(10). ACM, 2011.

[26] Minjang Kim. *Dynamic Program Analysis Algorithms to Assist Parallelization*. PhD thesis, School of Computer Science Georgia Institute of Technology, 2012.

[27] Minjang Kim, Pranith Kumar, Hyesoon Kim, and Bevin Brett. Predicting Potential Speedup of Serial Code via Lightweight Profiling and Emulations with Memory Performance Model. In *IEEE 26th International Parallel & Distributed Processing Symposium*. IEEE, 2012.

[28] Boontee Kruatrachue and T.G. Lewis. Duplication Scheduling Heuristics (dsh): A New Precedence Task Scheduler for Parallel Processor Systems. *Oregon State University, Corvallis, OR*, 1987.

[29] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and Comparison of the Task Graph Scheduling Algorithms. *Journal of Parallel and Distributed Computing*, 59(3), 1999.

[30] James LaGrone, Ayodunni Aribuki, and Barbara Chapman. A Set of Microbenchmarks for Measuring OpenMP Task Overheads. In *Proceedingis of International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 2. Citeseer, 2011.

[31] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization*. IEEE, 2004.

[32] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *ACM SIGPLAN Notices*, volume 33(11). ACM, 1998.

[33] Charles E. Leiserson. The Cilk++ Concurrency Platform. *The Journal of Supercomputing*, 51(3), 2010.

[34] Paul Lindberg. Performance Obstacles for Threading: How Do they Affect OpenMP Vode? https://software.intel.com/en-us/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code, 2009.

[35] Chris Lomont. Introduction to Intel Advanced Vector Extensions. *Intel White Paper*, 2011.

[36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM Sigplan Notices*, volume 40(6). ACM, 2005.

[37] Benoit B. Mandelbrot. Fractal aspects of the iteration of z λz (1-z) for complex λ and z. *Annals of the New York Academy of Sciences*, 357(1), 1980.

[38] Michael McCool, Arch D. Robison, and James Reinders. *Structured Parallel Programming*. Morgan Kaufmann, 2012.

[39] OmpSCR: OpenMP Source Code Repository. http://sourceforge.net/projects/ompscr/.

[40] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0, May 2008.

[41] Chuck Pheatt. Intel® Threading Building Blocks. *Journal of Computing Sciences in Colleges*, 23(4), 2008.

[42] Fiona Reid and J. Mark Bull. OpenMP Microbenchmarks Version 2.0. In *Proc. EWOMP*. Citeseer, 2004.

[43] Rizos Sakellariou. Estimating the Parallel Start-up Overhead for Parallelizing Compilers. In *Parallel Computing Technologies*. Springer, 1997.

[44] Tao B. Schardl, Bradley C. Kuszmaul, I. Lee, William M. Leiserson, Charles E. Leiserson, et al. The Cilkprof Scalability Profiler. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*. ACM, 2015.

[45] Gilbert Cc Sih, Edward Lee, et al. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2), 1993.

[46] Luis Moura Silva and Rajkumar Buyya. Parallel Programming Models and Paradigms. *High Performance Cluster Computing: Architectures and Systems*, 2, 1999.

[47] Oliver Sinnen. *Task Scheduling for Parallel Systems*, volume 60. John Wiley & Sons, 2007.

[48] Lev Tolstoy. *War and peace*. Gutenberg EBook, 2013.

[49] Min-You Wu. MCP Revisited. *Department of Electrical and Computer Engineering, University of New Mexico*, 2000.

[50] Min-You Wu and Daniel D. Gajski. Hypertool: A Programming Aid for Message-Passing Systems. *IEEE Transactions on Parallel & Distributed Systems*, (3), 1990.

[51] Jia Xu and David Lorge Parnas. Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations. *IEEE Transactions on Software Engineering*, 16(3), 1990.