Theory and Engineering of Scheduling Parallel Jobs

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Jochen Matthias Speck

aus Heidelberg

Tag der mündlichen Prüfung:	14.12.2017
Erster Gutachter:	Prof. Dr. Peter Sanders
Zweiter Gutachter:	Prof. Dr. Klaus Jansen

Hiermit versichere ich, die Arbeit selbstständig angefertigt, alle Hilfsmittel und alles, was aus Arbeiten anderer entnommen wurde, angegeben oder kenntlich gemacht zu haben.

Jochen Speck

Abstract

Scheduling in the context of computer science is finding an efficient execution plan called schedule for a single job (scheduling internal tasks) or a set of jobs on a computing system. This includes assigning the used resources, for example cores or memory/cache space as well as setting other parameters like operating frequencies. Of course the found schedule should be feasible and realizable. Good schedules are important for an efficient usage of computing systems regarding different goals like high throughput, small power consumption or small makespans. Also it is important that the computation of the schedule itself does not require too much time or resources.

Nowadays computers are usually parallel machines with several concurrently working cores. Also the used applications (especially the ones with a high computation demand) are more and more parallelized today. Hence the scheduling of parallel applications is becoming an important factor of the efficiency and performance of today's computing systems.

Scheduling is a large field investigated in many different contexts and with many different approaches. For example the "Handbook of Scheduling" (edited by Joseph Y-T. Leung) has 1224 pages and references about 2000 other scientific works in the field. Even scheduling parallel jobs on parallel machines is a large field. Despite the invested effort there is a large gap between research in scheduling theory and the engineering solutions for the scheduling of parallel jobs used for experimental research or production implementations.

Our approach is to study the developments in theory as well as in practice (engineering) and to use the knowledge from both sides to produce results which might narrow the gap. One reason for the gap between theory and practice is that the scheduling models used in theory assume more knowledge about the jobs than the knowledge which is available in practice. Furthermore, some models from theory assume even a possibility for the system scheduler to change the behavior of the applications which does not exist in practice yet. But there are approaches

to make the applications more adaptive to changes of the system status and to provide more information for the operating system. One example of developments that aim to improve the information exchange between operating system and applications and also the adaptivity of applications is the InvasIC project. The author of this work took part in this project for four years which led to a collaboration with experts for many different parts of modern computer systems. The improved information exchange and the adaptivity of applications are engineering steps which change the system such that results from scheduling theory are better applicable. Especially the model of malleable jobs, which can adapt their resource usage during runtime, looks like a fitting counterpart to the InvasIC developments on the theoretical side.

During our work within InvasIC we were able to identify four main research directions for the scheduling of parallel computing systems (each described in its own chapter):

- *Distribution of decisions and information among the different decision makers within the system.* In modern systems different decision makers are present, for example the OS scheduler or application-internal schedulers. These schedulers need some kind of coordination to produce good results.
- *Fast and efficient finding of good decisions (the classical problem researched in scheduling theory).* The efficient solution of the often complex scheduling problems remains the core of every scheduling system. A large basis for this can be found in scheduling theory.
- *The efficient usage of memory and caches.* The efficient usage of the memory system is a central requirement to reach high performance and efficiency.
- *The reduction of power and energy usage.* Energy consumption and heat issues are a main barrier against easy performance gains through increasing clock speeds. Moreover the rise of battery-powered devices increases the importance of this area.

This work presents results in all of these four areas. The main result of this work is a fast scheduling algorithm for malleable jobs that finds optimal schedules given that the problem fulfills some conditions. The objective functions that are minimized by our main algorithm can be either maxima of job properties or sums of job properties. We also show that our main algorithm can be parallelized and that the parallel version finds an optimal schedule in a polylogarithmic time when the number of participating cores is at least as large as the number of jobs. This makes this algorithm the first parallel scheduling algorithm for problems with parallel jobs (to our knowledge). Some applications of the main algorithm are presented including the minimization of the energy consumption of a set of malleable jobs. To our knowledge, the application to energy minimization is the first algorithm to optimally solve the energy minimization for a problem with malleable jobs. Besides the main result we also describe a heuristic for the fast and efficient scheduling which has proved successful in a competition.

Regarding the coordination and the information exchange between different schedulers, this work contains some general considerations from other fields and an example of a malleable application and a fitting interface.

The memory and cache behavior of modern computers is investigated with extensive experiments in this work. Two case studies of memory-optimizations for real-world problems were conducted with our contributions. Their results are described following the basic experiments regarding cache and memory behavior. We also look into the power consumption of memory operations.

Acknowledgements

There are many people I have to thank for their help and support during my way into research and during writing this thesis. As nobody will read 20 pages of acknowledgements, I have to focus on the most important ones.

First of all I wish to thank my doctoral advisor Peter Sanders for his numerous ways in supporting me. He not only introduced me into many interesting topics (especially scheduling) and helped me with his wide knowledge about computer science and research methodology but also left me a great amount of freedom. During my time in his group I had the opportunity to learn a lot from him. I also wish to thank Klaus Jansen for taking time to review my thesis and for his helpful remarks.

For their help during the writing of my thesis I would like to thank Veit Batz for his help with the thesis layout, Timo Bingmann for the discussions about properties of memory hierarchies, Marius Hillenbrand for discussions about the Linux kernel and some helpful literature suggestions in that area and Lorenz Hübschle-Schneider for proofreading parts of my thesis. I also would like to thank all my paper co-authors Felix Brandt, Patrick Flick, Tobias Maier, Peter Sanders, Raoul Steffen, Jesper Larsson Träff and Markus Völker. The collaboration with you was a pleasure. Further thanks go to my current and former colleagues at the chair of Peter Sanders especially to Veit Batz, Christian Schulz and Timo Bingmann for the interesting and instructional work-related and non-work-related discussions. The time in the research group was enjoyable and I learned a lot. In addition I would like to thank the InvasIC team for introducing me into many different parts of modern computer systems and for all discussions that helped my understanding of scheduling problems.

Special thanks go to my parents Elisabeth Speck and Wolfram Speck for their countless ways of supporting me. I would also like to thank my uncle Bernhard Westermann for introducing me to computers and programming.

Contents

1

Introduction and Formal Basics

1.1 Introduction

What is Scheduling? Given a set of jobs **scheduling** is making the decision **when**, with **which worker** and **how** to do each of these jobs. For example if a large numerical computation consisting of many small sequential jobs has to be done on a parallel computer, it has to be decided when to compute each job, which core does the computation and maybe the operating frequency of the selected core has to be determined. Of course, there might be a lot of constraints for this decision, for example job A has to be completed before job B starts, two jobs should not run on the same core at the same time or there might be an upper power limit for the computer. There are also some possible goals for a good scheduling like "completing the whole computation as fast as possible" or "using as little energy as possible". Finding a schedule has obviously also some practical constraints: finding the schedule should be fast compared to the whole computation and the schedule should not consume too much memory.

The scheduling of a large numerical computation is an example for the field of scheduling problems. Such problems can be categorized by their four main properties:

- Decision Space: What can be decided, for example which job will run on which core.
- Properties and Constraints: What are the main properties of the jobs and cores, for example the amount of work in each job. What restricts the possible decisions, for example two jobs cannot run on the same core at the same time.
- Goal: What is the goal of these decisions, for example to finish the whole computation as fast as possible.
- Information: Which properties are known by the scheduler, for example the

running time of the small jobs.

In order not to get lost in details the actual setting of a scheduling problem is usually reduced to a **model** to get rid of the unnecessary details. This also helps to find similarities between different scheduling problems.

Scheduling is a quite common part of many activities. If you want to cook a meal consisting of meat, potatoes and vegetables you have to schedule when to prepare each of the three parts and when to cook them (Baker and Trietsch [9] also use a cooking example in their introduction on page 2). Maybe each of the three parts needs a different time on the hotplate. If your goal is to get the meal ready as fast as possible a typical scheduling decision might be to prepare the part which needs the longest time on the hotplate first, so you can prepare the other parts while the first is already cooking. This description already implies a simplified model. There might be possibilities to reduce the cooking time by using a higher temperature for example, but this is not part of the described decision space.

Nobody will think about a formal schedule for a job like cooking, but scheduling (maybe without an explicit name) has a long history as some kind of time planning was probably done (even hundreds of years ago) for example while building large bridges or cathedrals. One of the first methods of planning jobs formally were Gantt charts introduced at the beginning of the 20th century (see Wilson [142]). These charts made job durations and dependencies explicit, but it remained in the responsibility of the scheduler (a real person in the first half of the 20th century) and his intuition to create a good schedule from this information.

Scheduling algorithms appeared much later. One of the first scheduling algorithms was Johnson's algorithm for finding an optimal sequence in a job shop problem, which was published in 1954 [82]. Hu's algorithm (published in 1961 [68]) for scheduling a task-DAG to parallel workers is one of the first scheduling algorithms for parallel workers. These algorithms and the introduction of computers into manufacturing planning built the basis for modern scheduling.

Scheduling is now a very big field for example the "Handbook of Scheduling" [93] one of many books about scheduling has 1224 pages and references about 2000 other scientific works in the field. Scheduling also plays an important role in many different application fields from computer science to manufacturing planning over logistics to civil engineering (a book about scheduling especially aimed at civil engineers is [69]). Scheduling is an important tool in these fields to reach several important goals or combinations of them:

- Meet deadlines
- Speed up projects
- Increase the efficiency of workers
- Maximize throughput

- Reduce costs
- Decide the most efficient application of resources

Hence scheduling plays an important role in the worldwide economy to provide economic wealth, to reduce resource usage and to provide on time service. More general optimization (like scheduling) is the often underestimated power that improves efficiency in all parts of society (from plant schedules to faster journey times) and hence facilitates a high standard of living.

In the first years after the invention of useful programmable computers in the 1940s **scheduling in computer science** was no important topic. Until the early 1960s most computers where specialized machines for either numerical computations or business management computations (see Tanenbaum and Bos [129, page 9] and Tanenbaum and Austin [128, page 38,39]). The programs were either run by the programmers (who reserved the machine for a certain time) or by batch processing. Especially there was at most one (sequential) program ready to run at the same time. Hence there was no decision space for a scheduler inside the computer. Thus the first schedulers inside computers appeared with the invention of multi-programming. Multi-programming was introduced in the early 1960s to use the waiting time for input in one program to run another one (see Tanenbaum and Bos [129, page 11]). Having now possibly more than one program ready to run at the same time there was a real decision to make. Since then schedulers are a typical part of operating systems.

Scheduling in computer science has a lot in common with scheduling in other fields, but there are also some specialties in computer science that are important to find good schedules and scheduling algorithms. First of all scheduling in computer science is usually fully automated. There are typically a lot of decisions to make and many of them have to be fast. For example, if a job runs for an average of 5 milliseconds, then each 5 milliseconds a decision has to be made which job to run next. Hence in computer science schedules are typically computed by scheduling algorithms (and not by humans) and it is often not distinguished between scheduling and scheduling algorithms. Early computers had one sequential core which ran sequential programs, hence the main decision space was which ready program should run next. On parallel machines the decision space grows because now it is not only necessary to decide which job is next but also on which core. Also common resources like common caches and memory connections play an important role now. Parallel programs might add some more constraints or at least things to incorporate into a scheduling approach like which threads of a parallel application should run in parallel or on the same NUMA socket. Large parallel programs also often come with their own application-internal scheduler which can use application-internal knowledge to schedule internal jobs. This leads to the more complicated situation of a scheduling hierarchy with one scheduler on the operating system level responsible for the whole machine and possible application-internal schedulers for each parallel application which have the advantage of a possible better knowledge about their application. Thus changes in hard- and software have major implications on the scheduling in computer science. Also the goal of saving energy during computations becomes more and more important.

The constant changes in hard- and software lead to a constant adaption of schedulers in practice. Scheduling theory instead often focuses on more general models that might not fit all properties of modern computers. The common theoretical base of scheduling in computer science with scheduling in other fields also slows down the adaption of new hard- and software developments in scheduling theory. Also theory often assumes much knowledge about the jobs to schedule. A major example of differences between scheduling theory and computers used in practice are the memory system and caches which have a relevant influence on the performance, but current scheduling theory pays little attention to them. Scheduling in practice often has nearly no knowledge about the jobs because interfaces designed long ago prevent these schedulers from accessing relevant information. Hence these practical schedulers have no need for complex theory because without this knowledge complex scheduling algorithms are hardly useful. Thus there is a widening gap between research in scheduling theory and the scheduling that is used in practice on real computers.

The goal of this work is to research scheduling solutions for future parallel computer systems. Another goal is reducing the gap between scheduling theory and scheduling practice in computer science. We work towards this goal by applying the methods of algorithm engineering on scheduling and by further development of the theory. On the practical side this is done by including ideas from theory and by the proposal of new interfaces which can improve efficiency. On the theory side the focus lies on the development of fast scheduling algorithms with optimal or nearly optimal results. The work also lays more stress on the efficiency gains through scheduling than on scheduling to meet deadlines. Thus it has more a systems efficiency approach than a real-time systems approach. Our published results related to this work are listed in Section 3.4.4.

Overview of the work In the rest of this Chapter the formal basis for the decision process of scheduling is specified in Section 1.2. Chapter 2 contains a description of the developments in computer science that are important for scheduling. The main directions of work in theoretical and practical scheduling in computer science are depicted in Chapter 3 which also contains a detailed description of our approach. Our view of the interplay of different schedulers (system-wide, application-internal) is described in Chapter 4. Our work concerning the computation of schedules (which is usually meant with the term scheduling) is described in Chapter 5. We also look into scheduling solutions for the memory hierarchy (see Chapter 6) and energy-efficiency (see Chapter 7). This work is concluded by a summary and an outlook in Chapter 8.

1.2 Formal Descriptions and Complexity

This section is devoted to the formal basis of scheduling. As already noted the four main properties of a scheduling problem are the decision space, the constraints, the goal and the information. For a general scheduling problem consisting of a set of jobs $\mathscr{J}$ and a set of workers/machines $\mathscr{M}$ we will now define these four properties:

Decision Space The decision space is everything that can and must be decided by the scheduler. The fixed decisions form the schedule.

- Which workers work on which jobs at which time intervals.
- What are the execution parameters for the workers and the jobs (if the problem contains such additional parameters).
- How are the additional resources distributed among the jobs (if the problem contains such resources).

If all decisions are fixed for all jobs, the schedule is complete as there are no further decisions to make. The decisions within the decision space can often be described by decision variables.

Properties and Constraints The properties contain the relevant information about the jobs and the workers. Not all possible combinations of decisions from the decision space lead to feasible schedules. The constraints restrict the possible combinations of decisions. The constraints also contain all other restrictions for the problem.

- The availability of the workers, possibly time dependent.
- Which workers can work on which jobs and how fast is the progress for these combinations.
- How do the additional parameters from the decision space influence progress or resource consumption.
- The circumstances under which a job becomes ready to be worked on (release time, dependencies).
- Deadlines and restrictions on additional resources.

It is difficult to distinguish between properties and constraints. A deadline of a job might be more a constraint than a property, but the amount of work of the job which has to be completed before the deadline is a job property but clearly restricts the possible decisions in the schedule.

Goal/Objective For some scheduling problems it is sufficient to find a schedule that fulfills all constraints, but more often the problem is to find a schedule that additionally optimizes a goal. In the classical scheduling theory the goal is usually a non-decreasing function of the finishing times of all jobs of the problem (for example the finishing time of the latest job C_{max} or the sum of finishing times of all jobs $\sum_j \omega_j C_j$). Baker and Trietsch [9, page 13] call an objective function $Z = f(C_1, \ldots, C_n)$ of the finishing times "regular performance measure" if $Z' > Z$ implies that $C_j{}' > C_j$ for at least one job j. Additionally/alternatively the objective function can include parts dependent on the amount of the used resources (for example used energy).

Information The information is the knowledge the scheduler has about the properties and constraints of the problem while making a decision. If parts of the relevant information of the scheduling problem become known to the scheduler only during the runtime of the schedule, scheduling becomes an online problem. There are two reasons for missing information: the information is generated during the scheduled process or the information is not accessible by the scheduler. Information that is not accessible by the scheduler at all is usually not part of the scheduling model.

Of course these four properties always heavily depend on the used model. The modelling of parallel computers is described in Section 2.6.2.

For the rest of the Section we look into the details of the 3-field problem classification which is widely used in scheduling theory. Finally we introduce the theory of NP-completeness and some complexity theory in order to have a basis to decide how difficult it is for certain scheduling problems to find a good schedule.

1.2.1 3-Field Problem Classification

As there is such a big number of different scheduling problems there is a need to describe them in a fast and standardized way. Graham et al. [55] introduced the 3-field problem classification $\alpha|\beta|\gamma$ to cope with this problem. The classification always describes the things which characterize the problem class. Individual problems in that class contain additional input.

The first field specifies the machine (or the workers), the second field describes the jobs and the third field the goal (or optimality criteria). To get an idea how this works we give some important descriptors for each field and some problem examples afterwards.

Machine Field α $\alpha = 1$ stands for a sequential machine, $\alpha = P$ for a parallel machine with identical workers and $\alpha = Pm$ means that the number of available parallel workers is fixed to m (otherwise the number of workers is part of the input). Other possible machine types are $\alpha = Q$ which means uniform workers which only differ in their general speed independent of the job. $\alpha = R$ means unrelated workers such that the speed of each worker might be dependent on the job. The necessary speeds for R and Q are part of the input. Hence the machine field specifies the machine properties of the scheduling problem but also a part of the decision space (number of possible workers for each job).

Job Field β If β contains *pmtn*, the jobs can be stopped and the rest of the job can be completed later (preemption), otherwise all started jobs must run without interruption. $prec \in \beta$ denotes the existence of precedence relations between the jobs, these relations form a DAG with the jobs as nodes where a directed edge from a to b means that a has to be completed before b can start. $d_i \in \beta$ means that the jobs have individual deadlines, $r_i \in \beta$ analogously means that the jobs have individual release times. If *prec*, d_i or r_i are not present in β, this means respectively that there are no precedence relations, deadlines or all jobs are available from the start. In the opposite case when *prec*, r_i or d_i are given, concrete precedence relations, release times and deadlines for each job are part of the input. Another important part of β is p_i where $p_i = 1$ means that all jobs have the same running time and $\bar{p} \leq p_i \leq \hat{p}$ means that all running times lie in a specified interval.

Although the 3-field notation was only intended for sequential jobs in the beginning, it was later expanded for parallel jobs (the notations for parallel jobs are taken from the "Handbook of Scheduling" [93, page 25-5]). If no indication of job parallelism is given in the β field, all jobs are sequential. $size_i \in \beta$ means that there are parallel jobs with a fixed degree of parallelism which means that there is a fixed number of workers for each job which must work in parallel on this job. *any* is similar to *size* but the scheduler can decide the degree of parallelism for each job when the job starts, these jobs are called moldable. *var* gives the scheduler even more freedom as the degree of parallelism can be changed anytime during the execution of a job, this includes using 0 workers and thus preemption, these jobs are called malleable. For some problems a maximal degree of parallelism for each job is given through $\delta_i \in \beta$. When the scheduler can decide the degree of parallelism of a job, it is of course important how the parallelism affects the job's behavior, especially the running time. If nothing is given in the β field about the working speed of a job with different numbers of workers, the speeds can be defined completely arbitrary in the input, for example running with 4 workers can even be slower than running with 3. If the speedup is a simple function of the number of workers, it is often given in the form $p_i(q) = \frac{p_i}{q}$ (for a linear speedup, q is the number of workers for the job) within β. More complex speedup functions or function classes are usually given through an additional text outside the 3-field classification.

There are several different notations for additional resource needs depending on the nature of resources and other special conditions. A notation for discrete resources is given by Błażewicz et al. [25].

The job field can also contain some description about the information the scheduler has about the jobs. For example the Handbook of Scheduling [93, page 15-2] defines $online-time$ and $online-time-nclv$. In $online-time$ the scheduler learns about the existence of a job when it becomes ready but also gets the knowledge about the running time of the job immediately. $online-time-nclv$ gives less information as the scheduler does not know the running time of a job until the job is finished.

Altogether the job field specifies the properties of the jobs, the constraints of the problem, the available information and parts of the decision space (degree of parallelism). In some cases the job properties are too complex to be specified in such a formal way.

Goal Field γ This field contains the definition of the goal of the scheduling process. It is the measurement which makes one schedule better than another. The goal of a scheduler is to minimize the function given in γ. The most common component of such functions are the completion times of jobs, usually denoted as C_i where i denotes the respective job. Typical goals are to minimize $\gamma = C_{max}$, which means the completion time of the last job, or $\gamma = \sum \omega_i C_i$ which means the weighted sum of completion times with an individual weight for each job. Resource usage dependent goals (energy minimization) do not have a widely used standard in the 3-field notation yet.

Examples

- $1||C_{max}$ describes problems on a sequential machine with unrelated sequential jobs which are all available from the beginning and have no deadlines. The running times of the jobs are part of the input. The goal is to minimize the finishing time of the last job. This is one of the simplest problem classes.

- $P|pmtn|C_{max}$ describes problems on a parallel machine with unrelated sequential jobs which are all available from the beginning and have no deadlines. The jobs can be preempted. The running times of the jobs are part of the input. The goal is to minimize the finishing time of the last job.
- $P|pmtn, size_i|C_{max}$ is the same class as above but with parallel jobs instead of sequential ones. The number of workers each job needs is part of the input.

The 3-field notation is a compact and useful way to define classes of scheduling problems. Unfortunately for some problems there is no 3-field notation yet, for example for scheduling problems from the field of dynamic voltage and frequency scaling (DVFS). As the 3-field notation is the typical language of scheduling theory, it will be used as often as possible in this work.

1.2.2 Computational Complexity

In order to find a solution for a scheduling problem, it is not only important to find a good schedule, it is also important that finding a good schedule is not too expensive and does not take too long. Hence we will give some basics from the big $\mathcal{O}$ notation, NP-completeness and related subjects here in order to have the tools to describe the computational complexity of scheduling algorithms.

Big $\mathcal{O}$ Notation A typical way to describe the computational effort of an algorithm is the big $\mathcal{O}$ notation, which is a basic tool in many parts of computer science. As this notation is used frequently throughout this work we will recapitulate its basics here:

A function $f : \mathbb{N} \to \mathbb{N}$ is in $\mathcal{O}(g(n))$ if and only if there exist $c \in \mathbb{N}$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

A function $f : \mathbb{N} \to \mathbb{N}$ is in $\Omega(g(n))$ if and only if there exist $c \in \mathbb{N}$ and $n_0 \in \mathbb{N}$ such that $c \cdot f(n) \geq g(n)$ for all $n \geq n_0$.

A function $f : \mathbb{N} \to \mathbb{N}$ is in $\Theta(g(n))$ if and only if $f(n) \in \mathcal{O}(g(n)) \cap \Omega(g(n))$.

These notations are commonly used to give a clue about running times of an algorithm, where n denotes the size of the input in a useful way. The following definition specifies the typical way of coarsely classifying algorithms in algorithm theory.

Definition 1.2.1. *An algorithm with running time $f(n)$ for the worst input of size n and $f(n) \in \mathcal{O}(g(n))$ is called*

- *polylogarithmic time algorithm if and only if there exists a fixed $k \in \mathbb{N}$ such that $g(n) = \log(n)^k$ for all n.*

- *polynomial time algorithm if and only if there exists a fixed $k \in \mathbb{N}$ such that $g(n) = n^k$ for all n, and it is no polylogarithmic time algorithm.*
- *exponential time algorithm if and only if there exists a fixed $r \in \mathbb{R}$ with $r > 1$ such that $g(n) = r^n$ for all n, and it is no polylogarithmic or polynomial time algorithm.*

NP-Completeness A very important question for a scheduling problem is whether or not an optimal solution is computable within a reasonable amount of time. In complexity theory the common rough rule of thumb is that a polynomial time algorithm can be computed within reasonable time limits, but an exponential time algorithm can usually not be computed within a reasonable time.

There is an important class of problems called NP-complete problems which allow a solution with a polynomial time algorithm on a nondeterministic Turing machine, but it is still unknown if these problems can be solved with a polynomial time algorithm on a deterministic Turing machine (or an ordinary computer). The best-known problem in this class is called the 3-SAT problem, which is the question "Given a Boolean logic formula in conjunctive normal form with at most 3 variables per clause is it possible to find a value in $\{\text{true}, \text{false}\}$ for each variable such that the outcome of the whole formula is true?". A problem in the class of NP-complete problems is called NP-complete. Traditionally the theory of NP-completeness considers only problems where for each instance the solution is just *yes* or *no*. Here we consider problems where a useful solution is computed by the algorithms.

Between all NP-complete problems there exist polynomial time algorithms (on deterministic Turing machines) such that an algorithm A that solves one NP-complete problem can solve all other NP-complete problems by adding such polynomial time algorithms before and after A. Hence a polynomial time algorithm on a deterministic Turing machine that could solve one NP-complete problem would lead to polynomial time algorithms on deterministic Turing machines for all NP-complete problems. As many important problems are in the class of NP-complete problems and no algorithm was found yet to solve any of these problems with a polynomial time algorithm on a deterministic machine, the conjecture is that it is not possible to solve NP-complete problems with polynomial time algorithms on deterministic machines.

A good introduction and reference for the theory of NP-complete problems is given by Garey and Johnson [51]. For the reason of self-containedness we also give a definition of NP-hardness and NP-completeness.

Definition 1.2.2. *Let A_{SAT} be an algorithm to solve the 3-SAT problem. Then a problem P is called NP-hard if and only if for all algorithms A_P solving P there exist polynomial time algorithms A_{SP} and A_{PS} on deterministic machines such* *that $A_{PS}(A_P(A_{SP}(I)))$ is a solution for each 3-SAT input I. If additionally there are polynomial time algorithms $\bar{A}_{SP}$ and $\bar{A}_{PS}$ on deterministic machines such that $\bar{A}_{SP}(A_{SAT}(\bar{A}_{PS}(I)))$ is a solution for each P input I, then the problem is called NP-complete.*

In practice it is usually only important that a problem is NP-hard as this already yields that finding a polynomial time algorithm on a deterministic machine is highly unlikely as no NP-hard problem could be solved with a polynomial time algorithm yet.

When finding the size n of an input I of a problem P, it is usually assumed that more than one symbol is used to encode I especially the numbers in I. Hence the value of the largest number contained in I does not have to be bounded by a polynomial function of n. Let m be the maximum of n and the largest number encoded in I. Then some NP-hard problems can be solved by polynomial time algorithms with respect to m instead of n, for example PARTITION (see Garey and Johnson [51, page 90] and [51, page 60] for the NP-Completeness of this problem). If the problem restricted to the case that all numbers contained in I are bounded by a polynomial function of n is still NP-hard, then the problem is called NP-hard in the strong sense [51, page 95] or short **sNP-hard** or **sNPh** (a notation frequently used in the Handbook of Scheduling [93]).

Example: $P2||C_{max}$ is NP-hard but not sNP-hard as it is equivalent to the PARTITION problem (as pointed out by Lenstra et al. [92]). The more general (the number of workers is part of the input) $P|pmtn|C_{max}$ with preemption is not NP-hard as there exists a simple polynomial time algorithm called McNaughton's wrap-around rule [104] which finds the optimal solution. $P||C_{max}$ on the other hand is sNP-hard as pointed out by Garey and Johnson [50] (reduction from 3-PARTITION).

P-Completeness Given big parallel machines, it is often interesting whether an input can be computed on a large machine in polylogarithmic time regarding the size of the input (and thus make the running time less size-dependent). The book by Greenlaw, Hoover and Ruzzo [57] introduces into the theory of the limits of parallel computation. For a class of problems called Nick's Class there exist algorithms which can compute solutions in polylogarithmic time with a number of processors polynomial in n. For some other problems such algorithms seem impossible (similar to the NP-completeness there is no real proof other than many have tried and all failed). These problems are called P-complete. Especially for scheduling of big machines it is interesting to use all processors to compute the schedule in order to complete the scheduling in polylogarithmic time. Hence it is important to investigate if a scheduling problem is P-complete. Some special scheduling problems have already been proven to be P-complete.

1.2.3 Approximations and Heuristics

Algorithms which solve NP-hard problems optimally usually have an infeasible running time (exponential time). Hence one often has to use algorithms that may compute suboptimal solutions in order to get feasible (polynomial) running times. There are two different ways to get a faster algorithm: approximation and heuristic. Both ways are well-known in computer science but often differently defined. Hence we give our own definition for the reason of self-containedness.

Let I be an instance of a problem class P and A_{opt} an algorithm (not necessarily with feasible running time) which computes an optimal solution. For any algorithm A let further $\gamma(A(I))$ be the resulting value of the objective function after A has been used on instance I. An approximation algorithm A_{app} for a problem P is an algorithm such that there exists a (useful) function f with $\gamma(A_{app}(I)) \leq f(\gamma(A_{opt}(I)))$ for all $I \in P$ (assuming the goal is minimization). Common functions are $f : x \mapsto (1+\eta) \cdot x$ for an algorithm-specific $\eta > 0$, the respective approximation algorithms are usually called $(1+\eta)$-approximation and $(1+\eta)$ is called the approximation ratio. If η is not too big, such an approximation might be very helpful in practice. Depending on the problem approximation algorithms can be much faster than algorithms which compute an optimal solution even for small η.

If it is not possible (for example because the algorithm development would be too costly) to find a reasonably fast exact or approximation algorithm, one often uses a heuristic algorithm. Heuristic algorithms are often based on general optimization ideas. There are no guarantees for the solution quality compared to an optimal solution, but there are good heuristic algorithms in practice which are fast and yield quite good results for many of the relevant instances. Hromkovič [67, chapter 6] gives a good introduction into heuristics and also reviews different possibilities to define a heuristic.

2

Scheduling Fundamentals

The goal of this chapter is to introduce the environment of scheduling within computer science. Section 2.1 gives some examples of areas with high computational demand today and in the future and takes a look at the cost of these computations and how scheduling might help to further reduce these costs. In order to make decisions about the use of computer systems, it is of course important to know something about their build and structure. This topic is handled in Section 2.2 in which we first go into detail why we need parallel computers in order to fulfill high resource demands (Section 2.2.1). Then the parts of modern computer systems that are relevant for scheduling are described (Section 2.2.2) and their probable future development is depicted (Section 2.2.3). An idea from economics why such development processes might lead to suboptimal results is presented in Section 2.3. Some basics about the usage of parallel computer systems are presented in Section 2.4. It is also important in which software hierarchy level (library, application, operating system, etc.) the scheduling takes place and which information is available on this level. The hierarchy and the information flow through interfaces are depicted in Section 2.5. To develop a scheduling algorithm one needs to abstract from many details of the machine and the programs, hence one needs to use a model. The modelling of parallel computers and their applications is described in Section 2.6.2. The general algorithm engineering process is introduced in Section 2.6.1, we use this methodology on scheduling (more details in Section 3.4). We also introduce the DFG-project Invasive Computing, which was the main work area of the author of this work, in Section 2.7. Our findings during Invasive Computing are described in a later Section 4.4.

2.1 Importance and Cost of Large Computations

To get an impression of the order of magnitude of expenses for today's computing we can just take a look at Intel's annual report [70] which reports revenues of 14 billion dollars in 2014 for the Data Center Group alone (56 billions for Intel altogether) which is only a part of the sum spent on data center hardware (computers) in 2014. Following a report of van Heddeghem et al. [138] data centers worldwide used 268 TWh of electric energy in 2012 which is nearly one half of the annual consumption of electrical energy in Germany. This spending shows the huge importance of computations in today's world.

So why do people spend that much on computation, what are the typical applications? From the early days of computation when Konrad Zuse invented the computer, computers were used for numerical calculations in engineering. As computers became more widespread, the users generated a lot more different applications of computing. Today computing intensive applications are not only used in data centers but also on PCs and even smartphones. PCs and even smartphones today are parallel computers as they normally have more than one computing core.

Hence we have a lot of different application types which require a lot of computation effort. We list some of them:

- Numerical applications are still a big consumer of computing power especially in data centers of engineering companies or research laboratories. An important property of linear algebra algorithms is whether they are for dense linear algebra or sparse linear algebra, because the former is especially dependent on the performance of the functional units of the computer system whereas the latter is more dependent on the used data structures and the memory system. In both cases the floating point units and their memory connection play an important role. These applications play an important role in the design of cars, planes and a lot of other things needed in everyday life to improve their efficiency, safety and to reduce production costs. Research in many areas like material sciences or fluid dynamics is heavily dependent on these applications.
- Optimization applications play a role across all levels of computer systems, from route and tour planning algorithms on smartphones to business optimizations on servers. The resource demand of these applications can be different as a lot of different algorithms are used. Optimization (like scheduling itself) is an important tool throughout the economy and thus an important application of computers.
- Computer knowledge generation is the kind of application where the com-

puter is used to gain knowledge out of unprepared, unstructured and somehow natural data. This ranges from speech recognition in smartphones and PCs to search engines and artificial intelligence.

- Special applications like in-memory databases or applications from the computational biology also require high computational effort but are also an important service for their users.

Although there is no common special resource (floating point units, integer/logic units, cache, memory bandwidth, or others), all of these applications have in common that they have significant computation costs but also provide a high value for their users.

How can scheduling help to improve the efficiency of these applications and thus reduce costs? Let us first take a look at the three main cost factors of computation besides administration:

1. Hardware. The purchase cost of the hardware or the depreciation during the running time of the application.
2. Energy. The energy usage of the hardware due to the execution of the application. Also cooling effort, which is an often underestimated part of the energy usage.
3. Programming or software licences.

If programs run faster or more efficiently the applications use the hardware for a shorter time or more applications can be run on the same hardware in the same time. Scheduling can improve running time and efficiency for example through useful resource distribution (allocate resources to the program which uses them in the most efficient manner), better ordering of small tasks within programs to reduce waiting times and through balancing temporarily changing resource demands between different applications. Hence scheduling can reduce hardware cost. Scheduling can also reduce the number of cache misses, memory swaps to hard disk and the needed average core frequency. Thus scheduling can also improve the energy consumption of a computing application. The cost for software licences can be reduced in the case where licences are needed during the time a special library runs or if the licence cost is proportional to the number of computers. In these cases the software cost has the same behavior like the hardware cost (software licences are then like an additional piece of hardware). The programming effort can not be reduced directly by scheduling, but maybe in an indirect way. For applications with a high performance demand programmers often avoid the use of libraries (for example parallelization libraries) to get the most out of the machine through optimized code. If these libraries would include an efficient scheduling such that their performance comes closer to the specialized code for the situation then the programming effort for these applications could be reduced.

So scheduling can be used to improve efficiency and reduce cost. In order to reach that goal, the designer of scheduling algorithms has to know the performance relevant parts of modern parallel computers which are introduced in Section 2.2.

2.2 Parallel Computing Systems

2.2.1 Reasons for Parallelization

The applications described in the last Section have a high demand of computational power. The demand usually increases with the affordability of computing power. In the beginning even supercomputers were sequential computing machines. Between 2005 and 2010 personal computers became parallel devices and today even handheld devices have more than one computing core and thus are parallel computers. But what are the main reasons to move from sequential computers to the more complex parallel computers? The main reason is that the performance improvement of sequential computers was slowed by three performance walls: The Power-Wall, the ILP-Wall and the Memory-Wall (Asanovic et al. [8]).

Power-Wall Following the work of Borkar and Chien [15] in the 20 years before 2011 we had a typical scaling factor $s = 1.4$ for every technology generation. This means that for each generation the transistor dimensions are scaled by $1/s$ (also the capacitance), hence the area of the same logic is scaled by $1/s^2$. Due to the scaling the frequency could be scaled by s and the supply voltage by $1/s$. Using a formula from Kogge et al. [86] we get for the power density P_{dens}, the transistor capacitance C, the transistor density d (transistors per mm^2), the operating frequency f and the supply voltage V_{dd}:

$$P_{dens} = C \cdot d \cdot f \cdot V_{dd}^2$$

Thus the power density stayed the same when scaling by s without regarding the leakage power. As observed by Kogge et al. [86] in the last years the scaling of the supply voltage slowed down, this is partly due to efforts to reduce the leakage power [15]. Hence the power density is increasing if we continue scaling in the same manner. Thus the gains in operating frequency will slow down or stop totally when we have an upper limit for power density, which is indeed the case in order to have an efficient affordable cooling. This reduces further gains in sequential computing due to increased operating frequency substantially.

ILP-Wall More and more transistors per core were available over the last 20 years. Before the rise of parallel processors the designers used these transistors to build more complex cores in order to dynamically execute instructions from a sequential program in parallel. This is called **i**nstruction **l**evel **p**arallelism (ILP), which includes techniques like pipelining, branch prediction, register renaming, alias analysis and multiple parallel functional units. According to Borkar et al. [15] there was a 1000-fold microprocessor performance increase in the two decades before 2011 of which at least one order of magnitude is due to microarchitecture improvements. The other nearly two orders of magnitude are due to operating-frequency increases. In 1991 Wall [139] studied the limits of instruction level parallelism for real programs. Even though he assumed extreme capabilities for the processors, he found the average parallelism is around 7 and the median around 5. Hence the reachable degree of instruction level parallelism is bounded. Following Hennessy and Patterson [64, page 213] the returns of even more complex architectures for increased instruction level parallelism diminished around 2005. So the industry will no longer deliver gains in instruction level parallelism because it would be too costly and inefficient.

With 10.6 billion dollars Intel had the third largest research and development spending of all companies in the world in 2013, according to Fortune [28], thus one should not expect dramatically bigger spending for processor development in the future. Hence further large gains by dramatically improved processor designs are also unlikely from this perspective.

Memory-Wall Hennessy and Patterson [64, pages 18-21] picked 7 performance milestones from the years 1982 to 2010. For the first milestone (Intel 80286 processor) the memory latency was about two thirds of the processor latency (according to Hennessy and Patterson [64, page 20], probably memory access latency and the latency of a simple operation are meant). For the last milestone (Intel Core i7 processor) the situation changed dramatically, the memory latency is now 9 times the processor latency. Figure 2.1 illustrates the general trend: Both memory and processors gain speed through higher bandwidth and lower latency where the bandwidth gains are in both cases much bigger than the latency improvements. But especially in case of latency improvements the gain of the processor was much bigger than the gain of the memory. Hence waiting for a memory access is much more costly (in terms of possible operations during the waiting period) today than it was 30 years ago. In 1994 Wulf and McKee [143] realized that, if nothing changes, a processor would spend most of its time waiting for the memory instead of working. So these high memory access times (relative to the duration of an instruction) slow down sequential computations. Cache improvements can help to reduce the number of memory accesses, but not all these accesses can be saved. So the problem cannot be solved through caches alone. In case of a multi-threaded application the processor/core can work on

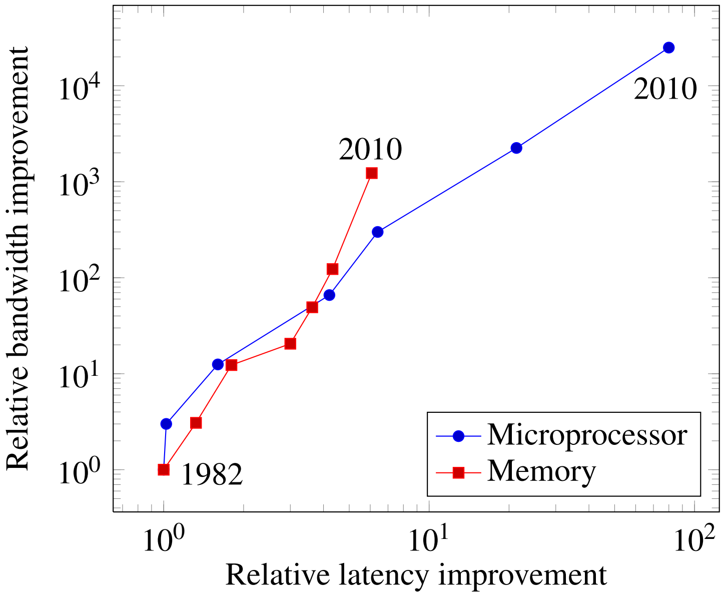


Figure 2.1. Log-log plot of the memory and bandwidth improvements of the performance milestones from 1982-2010 taken from Hennessy and Patterson [64], who show a similar picture.

one thread while another thread is waiting for the memory. We will present our own measurements on current architectures and our work on dealing with these latencies in Chapter 6.

The Power-Wall, the ILP-Wall and the Memory-Wall hinder bigger performance gains in sequential computing but can be circumvented by parallelization. So multi-threaded applications on systems with parallel computing cores are the best way of getting substantial performance gains in the future.

Also the cost of energy plays a more important role in the future. Parallelization can also help with that. Kogge et al. [86] point out that supply voltage scaling can be used for more efficient computations. The idea is that if one runs a chip with lower operating frequency, it is usually also possible to use a lower supply voltage. As the power consumption of a chip is proportional to $f \cdot V_{dd}^2$, but its performance is proportional to f, we can get half of the performance with less than half of the power. So if we can parallelize an application without significant overheads, running it on two processors with half frequency leads to the same running time but with a lower power consumption.

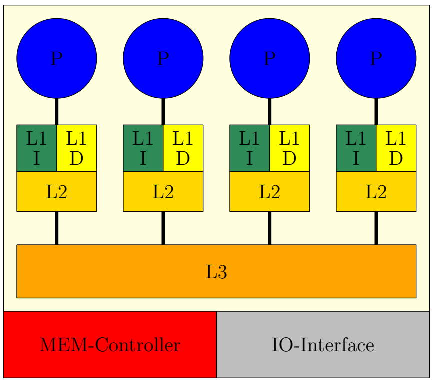


Figure 2.2. A typical multicore uniprocessor with two levels of cache exclusive for every core and one shared last level cache.

2.2.2 Build and Structure

Section 2.2.1 shows that parallel computing systems are needed for performance and future performance gains. We now look into how they are built and how their structure affects performance.

The by far most important producers for desktop and server processors are Intel and AMD. So how are the current processors composed? Since the introduction of the K8-architecture in 2003 all AMD processors have their memory controller on chip. Intel introduced the memory controller on chip with the Nehalem architecture in 2008 and has also built all desktop and server processors with this feature since then. Typically we have several independent cores on each processor which have the computational units to compute their own thread. In some cases such cores can even work on two independent threads which then share parts or all of the computational units (Intel's Hyper-Threading or the modules of AMD's Bulldozer architecture). These cores often share the memory controller and some sort of last level cache. In nearly all current processors the last level of cache is the third level where level 1 and 2 are not shared between cores. The level one cache is usually divided between data and instructions which is not the case for the rest of the cache hierarchy. A typical 4 core processor is depicted in Figure 2.2.

In case of a multiprocessor system several of these processors work together in one system. A typical 4-socket system is depicted in Figure 2.3. As every processor has its own memory controller, for a core on one processor it is faster to read or write data from the memory directly connected to that chip instead of accessing memory connected to a neighboring processor. This is called **n**on **u**nified **m**emory **a**rchitecture (NUMA) as access speeds to different parts of the memory can differ. Processors designed for multi-socket usage have special components to share their memory with processors on other sockets.

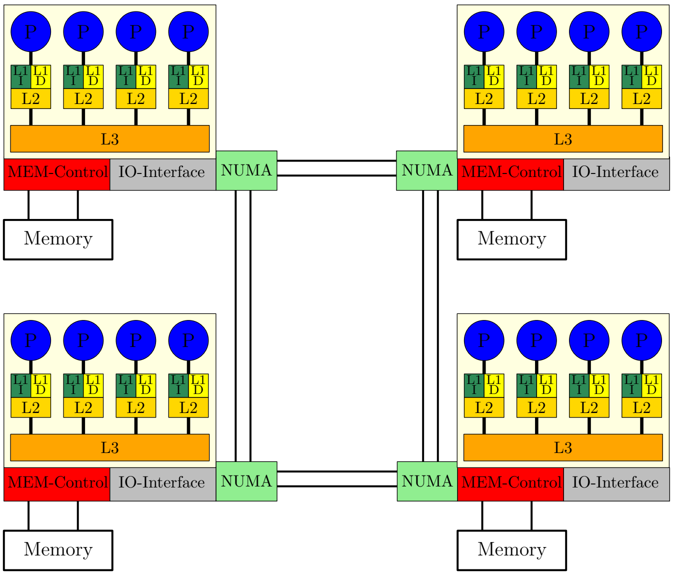


Figure 2.3. A typical four-socket multiprocessor multicore with NUMA-memory architecture. Each processor has a NUMA-controller for the connection to other processors.

Many other parallel computing devices have a similar build like parallel uni-/multiprocessor systems with several computing cores and a cache hierarchy between them and the memory.

The most important features of a parallel computer system hardware regarding computing power are the number of cores, the performance of a single core and the structure and the speed of the memory hierarchy. Of course the operating system and the used compiler also have an influence on the performance of a given program.

A careful design regarding the memory hierarchy (NUMA/non-NUMA, cache sizes, shared caches) is important for performance and also affects the energy consumption as a cache hit is much cheaper in terms of energy than a cache miss.

The producers of parallel computers have of course also found some ways to deal with the walls (see Section 2.2.1) that block further performance gains of sequential computers:

- ILP-Wall → multi-core processors, multi-socket machines
- Power-Wall → DVFS and lower frequencies in general
- Memory-Wall → cache-hierarchies and a large number of other techniques like prefetchers

Also the applications are more and more prepared for parallel computers by using multiple threads.

2.2.3 Further Development

The further development of parallel computers will lead to more devices integrated in even more parts of our everyday life. But also the computing power of "ordinary" computers and servers will increase as it is demanded by the market. After the end of frequency scaling computing power increase mainly stems from the increasing integration density of integrated circuits which allow to use the additional available transistors for example to build multicore processors or larger caches.

In 1965 Moore [105] expected the cost-optimal number of components on a single chip to double every year. This has become the basis of the term Moore's Law which is today used for many slightly different phenomena of exponential growth regarding transistor density or transistor count of chips used in computers. As already described in the discussion about the power wall in Section 2.2.1 voltage and frequency are not likely to scale anymore. Also semiconductor revenue as an important driver of development has slowed down its increase as observed by Mack [99]. It is likely that the increase in transistor density will continue for some time thus further reducing the cost per transistor [99]. Hence future developments will probably lead to processors with more cores and larger caches in order to use the more available transistors. One major problem will be to avoid to hit the power wall, as the voltage scaling has stopped and a significantly increased power density is economically unsustainable.

Altogether there seem to be some design trends for computing systems, that are without an alternative:

- Parallelization and multi-core processors
- Complex memory systems like several cache levels and multi-socket NUMA systems
- Power/energy-efficiency

2.3 Possible Suboptimal Development Results

This section is based on ideas found in the book *Increasing Returns and Path Dependence in the Economy* by Arthur [7]. The term system here is meant as the interfaces between hardware, operating system and applications and their usual properties, that ensure compatibility and the possibility to exchange all components independently of each other.

New hardware for a system is usually developed in order to speed up existing programs on that system. Also improvements in compilers and efficiency-related parts of operating systems (like schedulers) are designed to fit into the existing systems. The same is true for applications which are also designed to fit into existing systems. Hence optimizations that are not useful within an existing system are usually not developed as they would bring no benefits.

Let us now look at systems in an abstract way. Imagine systems A and B which are incompatible which means that there are substantial changes needed to transfer hardware, operating systems or applications from one system to the other. Let us further assume that the systems have similar benefits and initially the same market share. If by chance the market share of A gets ahead of the share of B, we can expect several results: as components of A generate more revenue than components of B, probably more money is spent on the further development of components for A; more software is designed to have a good performance on A than on B (or even designed for A only) because A is the more important platform. Hence customers choosing between system A and system B can expect better performance of A because of the higher development effort and the better fitting software. Thus the market share of A will further increase compared to B. Altogether we have a positive feedback cycle for the market share of A and B which will lead to the domination of the market by one system even if the fundamental ideas of both systems are equally good. Hence the competition between two systems is heavily influenced by past decisions/situations and is a typical example of **path dependence** as described by Arthur [7].

Following Arthur, it is even possible that the system which is worse in a long-term perspective wins the market under these circumstances because the initial benefits were greater or because the system was first in the market (for example because it was developed from an old one). So it is possible that the development path of modern computer systems is on a suboptimal path. The problem that current solutions might be only a local optimum is amplified in the computer industry by the extreme cost of replacing old interfaces with new ones. A replacement of a substantial part of the application $\leftrightarrow$ operating system interface would mean that all software programmed for that system would have to be changed. Even for testing new interfaces many components have to be re-designed and re-implemented. This makes testing alternative approaches for system interfaces very costly.

But given that path dependency in computer system development, it is possible that there are unrealized design options that would lead to a much higher efficiency. For example InvasIC is an approach to investigate such options.

2.4 Parallel Computer Systems in Use

Performance and efficiency of modern parallel computers does not only depend on their hardware but also on the used software and proper management and scheduling. Performance-demanding software for parallel computers typically offers the possibility that several workers work on different instruction flows (multithreaded software). An important property of such software for parallel computers is how well it can use the available parallel workers of the machine. Typically this property is measured by the **speedup** and the **parallel efficiency**. Given running times $T(1)$ on one core and $T(p)$ on p cores, the speedup is defined as $s(p) = \frac{T(1)}{T(p)}$ and the parallel efficiency as $e(p) = \frac{s(p)}{p}$. Given $s(p)$ for a certain job, it is called its speedup function. By definition $s(1) = 1$ and $s(0) = 0$. In reality the speedup functions depend on the used algorithm and other program characteristics but also on properties of the used machine like latency on the connections between cores or memory bandwidth of processors. Two rules of thumb are often used to guess the possible speedup: Amdahl's Law [5] and Gustafson's Law [58].

Amdahl's Law The general idea is that in all programs there exists a certain fraction s of the work that can only be done sequential. The rest of the work (the fraction $r = 1 - s$) can be done in parallel. When p cores are used the running time changes from $T = (s + r)T$ to $(s + r/p)T$. Hence we get a speedup of $s(p) = \frac{s+r}{s+r/p} = \frac{p}{sp+r}$ which is smaller than $1/s$ for all p. Thus the speedup is limited especially as Amdahl guessed that the sequential part would be 20% for most programs which limits the speedup to a maximum of 5.

Gustafson's Law Gustafson's Law comes from an article where Gustafson criticizes the speedup boundaries of Amdahl's law as too pessimistic. Gustafson assumes that the sequential part of a program is independent of the problem size the program is working on. He also assumes that computers with a larger degree of parallelism will rather be used for larger problem sizes instead of computing the same problem size faster. He assumed the problem size (computation effort) is proportional to the degree of parallelism of the machine. If the fraction of non-parallelizable work of a run on a sequential machine is s and $r = 1 - s$, then the running time of the problem for the parallel machine on the sequential machine is $T(1) = s + p \cdot r$ and $T(p) = s + r$ on the parallel machine. This leads to a speedup of $s(p) = \frac{s+pr}{s+r} = s + pr = p - (p - 1)s$.

Then Encyclopedia of Parallel Computing [109] identifies the often used term weak scaling with Gustafson's Law and the term strong scaling with Amdahl's Law. Also an often used term is the **efficiency** of a certain parallel program given as $e(p) = s(p)/p$. Inefficiencies might stem from the algorithm (sequential parts, P-completeness, see Section 1.2.2), the implementation or the machine.

The parallelization overhead of machine and implementation often increases with the degree of parallelism and stems from different sources:

- When more cores are used, data access over NUMA-connections or even network is needed when not all used cores can be part of the same processor.
- The expected average waiting time at barriers and other synchronization points increases. With more threads it is more likely that one thread needs more than for example 110% of the average running time between two synchronization points. Variance in running time between different threads are common because of machine disturbances like different cache hit rates or because of data-dependent computation effort.
- A higher degree of parallelism often also leads to a higher fraction of waiting time per core while waiting for sequential resources like locks.
- When resources like disks or network are the main bottlenecks of an application, a higher degree of parallelism just decreases efficiency.
- Shared resources like the memory controller or L3-caches are often also a reason for slowly increasing speedup functions.

In order to increase efficiency, it might be beneficial to run several programs in parallel on one parallel machine. If one program cannot use all the available cores of the machine (maybe only for some time interval) or runs its sequential part, other programs can use the free cores. The same is true if one application waits for input, disk or network. Also otherwise unused resources like memory bandwidth can be utilized better if a compute-intensive program is combined with a memory-intensive one. Of course it is important to take care that different programs do not slow down each other and thus decrease efficiency. This can happen if only one thread is slowed down between two synchronization points or if common resources are used in a way that slows down all users (for example cache thrashing). Thus there has to be an instance (scheduler) which distributes the available resources between different programs. Hence scheduling can play an important role to gain efficiency when using parallel computers.

2.5 Abstraction Hierarchies and Interfaces

One of the core concepts which made computer-science so successful is abstraction. Complex parts of a system that has to be used but not understood are encapsulated and the only thing to be known to the user is the interface. These encapsulations and interfaces are also relevant for scheduling.

Tanenbaum and Bos [129, page 4] give a nice example: The hardware interface of SATA hard disks had a 450 page description in 2007. No sane application programmer wants to deal with the hard disk at that level, because it would just take too long. Hence the operating system (or more precisely the disk driver) provides the programmer with a much simpler interface which provides reading and writing of blocks and encapsulates the more complex parts. This leaves more time for the programmer to work on the relevant parts of the application.

From this simple example we can already see the three main benefits of abstraction through encapsulation and interfacing:

- Complexity reduction. One only has to understand a relatively simple interface instead of the whole complex thing.
- Development work must be done only once. The interface can be used by many different clients, but the work within the encapsulation has to be done only once.
- Hiding of change. If the encapsulated function changes without changing the interface, the client does not need to be adapted. The client can even use different systems as long as the interface stays the same.

Today's modern computers even encapsulate the access to the basic resources every computation needs, to the computing cores and the memory. Modern computer architects have developed a whole hierarchy of abstractions from the basis of digital circuits up to modern programming languages. According to Tanenbaum and Austin [128, page 24] we have the following six layers of abstraction (five interfaces) in our modern computers:

1. Digital logic, consisting of transistors and gates.
2. Micro architecture, execution units and register files. Usually controlled by micro code.
3. Instruction set architecture, the language of interaction between software and hardware.
4. Operating system.
5. Assembler language.
6. Problem oriented language.

For this work the different layers of hardware abstraction are not important as there is no room for independent decisions in layers 1 and 2. We thus will look at the hardware (layers 1-3) as one layer that can make some decisions, for example cache replacement. The step from layer 6 to layer 5 is done via compilers which are not part of this work so we will look at the application software as one layer regardless of the used language.

An important layer missing from our perspective is the layer of libraries. Modern computers with parallel cores and cache hierarchies have made programming more complex, especially if an important goal is performance or efficiency. In order to make the task of programmers easier, there are a lot of libraries for the performance-consuming algorithms that sometimes also include their own parallelization. It is even possible that an otherwise sequential program only uses parallelism within its used libraries for example by using the MCSTL as developed by Singler et al. [121]. Also libraries are often provided by computer systems vendors which gives another reason to separate them from the application code. Unfortunately the border between library and application is less clear than the border between hardware and operating system or operating system and library/application. Also many libraries might use other libraries or there might be programs without complex libraries. In some domain-specific cases (numerical linear algebra for example) it is clearer where the border between application and library layer is. Hence we omit the library layer for the general considerations although we think that it is an important layer to reduce complexity for the application programmers.

Hence for the rest of this work we will look at 3 different abstraction layers of computer systems:

1. Hardware
2. Operating system
3. Application code (including potential libraries)

Each layer has its own decision space relevant for the performance and efficiency of the machine:

On the hardware layer cache replacement, memory prefetching, processor internal branch prediction and superscalar execution are decided. Also memory controllers or network devices may reorder data transfers within their own decision space.

The operating system level has the biggest decision space, hence it is the layer most people think about when they hear scheduling. The operating system decides the resource allocation between different jobs and the placement on the hardware. More specific: Which thread runs on which core and for how long (libraries or programs might request special cores). The thread-based decisions might also be based on the job that created the thread. It also decides about the memory allocation requests and on which NUMA-node they are fulfilled (except the library or program requesting the memory give special orders about that). Also the access to other types of hardware is usually managed by the operating system.

The application can decide the degree of parallelism and the order and the specific thread on which small sub-tasks are computed. Also the used algorithm is part of the decision space of the application. If the application is programmed such that it uses the computing resources of many computers connected over the network, it may also move work and data between computers. In some cases the amount of computation needed might be a decision of the application, for example it may reduce the quality of a video in order to keep the frame rate when there is not enough computing power available for high quality.

The advantages of this abstraction hierarchy obviously include the advantages of encapsulation as stated above. Especially the independent development and optimization of hardware, operating system and application (with restrictions by the stable interfaces) are important because each of these development processes is huge on its own and would be impossible without separation. Another very important reason for this abstraction hierarchy is that an operating system and applications can run on different kinds of computers, and applications are usually portable within the same operating systems family. Also applications do not need to know if other applications are running and which ones (this can also be seen as disadvantage, see below). Even a crash in one application often does not affect the others.

The operating system virtualizes computing cores which means that a job can be stopped and later restarted without preparations in the program. Hence preemptability (*pmtn* in the 3-field notation) can be usually assumed, but there might be performance losses through these preemptions, especially for parallel jobs or by losing the cache context. With the virtualization of memory and the operating system control over the other resources a program virtually has the whole computer for itself (from its own perspective). Hence the application programmer has no burden to think about other programs on the same machine. Also for the decision which algorithm to use for a task the application is independent of the state of the system or the hardware.

Unfortunately the abstraction hierarchy has not only advantages but also some disadvantages. The main disadvantage is the loss of information through the interfaces. Neither operating system nor hardware know much about the programs' internal data structures or computation flow. Also the programs usually have no knowledge about the system status, for example if other programs are present and how much resources they use, and the state of the hardware.

Also the optimization goals of the program (frame rate is more important than quality in the example above) are generally unknown to the lower layers of the system. This can lead to performance and efficiency penalties because decisions made within the lower levels have unwanted effects. DVFS of different cores may speed up the core computing the result needed last, NUMA-placement of threads may lead to many unnecessary memory accesses over the NUMA-connections, and an important thread may be stopped instead of a less important thread of the same application to make room for a new application. In Chapter 4 we take a deeper look into the interplay of different decision makers and its interference with the abstraction hierarchy.

A further disadvantage is that interfaces are difficult to change once they are established (see Section 2.3).

2.6 Algorithm Engineering and Modelling

2.6.1 Algorithm Engineering

An important scientific methodology which is used in this work is algorithm engineering as defined by Sanders [115]. We will summarize the most important aspects of algorithm engineering here and describe the adaptions for this special work in Section 3.4.

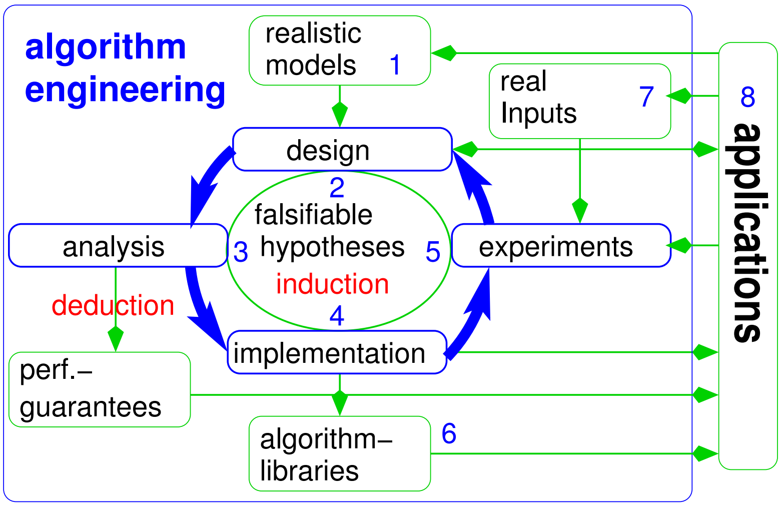


Figure 2.4. Algorithm engineering as a cycle of design, analysis, implementation and experimental evaluation of algorithms. Graphic received from Peter Sanders.

Algorithm engineering (AE) is the approach to design algorithms that are not only good in theory but also in practical applications. The core process of AE is a feedback loop or cycle: Design an algorithm → Analyze it in theory → Implement it → Perform experiments which might lead to new insights for a better algorithm design. This is different from the algorithm theory methodology which has a shorter loop which only consists of design and analysis.

The main parts of the AE process are:

1. Models: Good realistic models are the base of AE. Good models abstract from reality by keeping the important properties and are simple. Such models are often hard to find.

2. Algorithm design: In AE the goal of algorithm design is not only the best asymptotic worst case efficiency but also takes care of constant factors and the performance for real world inputs. Also the needed implementation effort plays a role in the design of algorithms.
3. Analysis: The analysis of algorithms can lead to performance guarantees but is often hard for practical algorithms under real circumstances.
4. Implementation: The implementation is an important part of AE not only to have code for the experiments but also the crossing of the semantic gap between theory and programming languages can lead to new insights.
5. Experiments: Well planned experiments with realistic input data are the best way to prove the quality of an algorithm for some kind of application.
6. Libraries: A good implementation of a good algorithm is often not easy. Implementations with good software quality can be used as library and reduce the complexity for the application programmers.
7. Real inputs: Meaningful tests require realistic inputs to get meaningful results.
8. Applications: A clear definition of the application scenario for an algorithm to develop is important.

In practice the AE cycle is followed for several iterations to get good results. The methodology is used in many different domains of efficient algorithm research.

2.6.2 Important Models

The methodology of algorithm engineering relies on useful models. This is also the case for scheduling theory which is dependent on models as basis for the investigation of decision problems. Hence we will describe some widely used models for parallel systems here.

An often used model for sequential machines in algorithm analysis is the random access machine model (RAM model). In this model a central computing unit (e. g. a processor core) with limited own storage can access a large main memory by directly addressing the memory cells. In order to describe parallel machines, this model is often expanded to the parallel random access machine model (PRAM model) with $m > 1$ central computing units. These m computing units access a common main memory. Usually the PRAM model is further specified by the kind of operations different computing units can execute on the same memory cell at the same time. Exclusive read exclusive write PRAM (EREW PRAM) means that only one computing unit can read or write a memory cell at the same time, concurrent read exclusive write PRAM (CREW PRAM) and concurrent read concurrent write PRAM (CRCW PRAM) are further often used PRAM models. JáJá [83, page 9] gives a more detailed introduction to the PRAM model.

The other commonly used model for parallel computers is the network model (JáJá [83, page 16]). These two models are also the two machine models described by JáJá [83]. The basis for the network model is a graph (the network) in which each node is a computation unit with local storage. These compute nodes can send each other messages over the edges. Nodes not directly connected can reach each other by routing over other nodes such that a path between the two communicating nodes is formed. Usually the length of this path plays a role for the duration of a data movement between the nodes.

Both parallel models are widely used for the development and analysis of parallel algorithms even as they ignore a lot of (more or less) important properties of modern computing systems for example memory hierarchies, hierarchical connections (same socket, same machine, same local network) and others. On the other hand ignoring less important properties and concentrating on the most important aspects of reality is the basic of modelling. Models must reduce the complexity of reality to make them useful. This helps the users of models to focus on the central properties of a problem/solution without getting distracted by too much information about less relevant aspects of the problem/solution. This leads to the central problem of creating models: which are the relevant properties of reality and what can be ignored? The answer to this question is unfortunately situation-dependent. For basic parallel algorithms the PRAM model might be the right choice, but if we want to develop especially cache-efficient parallel algorithms, the PRAM model is probably the wrong choice. Hence the right choice of the used model depends on the question to be investigated.

We use different models in this work. The job models from scheduling theory are introduced in Section 3.1.1. The Roofline model for the relation of memory bandwidth and computation is introduced in Section 6.2.1 and the energy model for energy-efficient schedules used by us in Section 7.1.

2.7 Introduction to Invasive Computing

Invasive Computing (Acronym: InvasIC) is a DFG funded project (transregional collaborative research centre) in which the author of this work took part as a PhD-student for 4 years. For a short definition of the principle idea we cite an overview article by Teich et al. [131]:

> **Definition:** Invasive Programming denotes the capability of a program running on a parallel computer to request and temporarily claim processor, communication and memory resources in the neighborhood of its actual computing environment, to then execute in parallel

the given program using these claimed resources, and to be capable to subsequently free these resources again.

The intended development of a new paradigm of parallel computing includes the redevelopment of all parts of a computing system: New language (at least in parts) and compiler, new operating system and new hardware. The central idea is *resource aware programming* which means that programs are self-organizing such that they can adapt to the available resources and negotiate with other programs about the usage of scarce resources. As these adaptions naturally can only take place during runtime the programs must be able to change their degree of parallelism and other resource needs *after* being programmed. This is facilitated by special programming techniques implemented in the language and the compiler and an agent system for the negotiation between the programs. The operating system and the hardware support collect the information needed for the decisions about the resources and support the resource allocation and deallocation. Some optimizations might also be in the responsibility of operating system and hardware. The negotiations and decisions are done by an agent system. We take a more detailed look at the scheduling decision making in the InvasIC system in Section 4.4

During the time the author of this work worked on InvasIC the project consisted of 12 research sub-projects (the list of projects can also be found in the annual reports [132] or [133]):

- A1 Basics of Invasive Computing: Modelling and definition of the programming language and the resource adaption techniques.
- A3 Scheduling and Load Balancing: Scheduling, the sub-project in which the author of this work took part.
- B1 Adaptive Application-Specific Invasive Microarchitecture: Reconfigurable processors for InvasIC.
- B2 Invasive Tightly-Coupled Processor Arrays: Special hardware accelerators for nested loops.
- B3 Invasive Loosely-Coupled MPSoCs: Development of special hardware controllers for the collection of scheduling relevant data and for making local decisions.
- B4 Hardware Monitoring Systems and Design Optimization for Invasive Architectures: Hardware monitors (temperature, degradation, power consumption, ...) for InvasIC.
- B5 Invasive NoCs – Autonomous, Self-Optimizing Communication Infrastructures for MPSoCs: The network on chip controllers for InvasIC.
- C1 Invasive Run-Time Support System (iRTSS): The operating system of

InvasIC.

- C2 Simulation of Invasive Applications and Invasive Architectures: Development of simulators.
- C3 Compilation and Code Generation for Invasive Programs: The compiler for InvasIC and its new language.
- D1 Invasive Software-Hardware Architectures for Robotics: Application 1.
- D3 Multilevel Approaches and Adaptivity in Scientific Computing: Application 3.

During the author's work on InvasIC a further sub-project Z2 was added in order to build a hardware simulator.

As one can see from the setup of the project the idea was to build a completely new architecture in order to escape the problems of the current architectures. This fits into the considerations about path dependence from Section 2.2.3. It is also clear that a core idea was to distribute the decision making process of resource allocation. The NoC-controllers and the controller developed by B3 make scheduling decisions as well as the operating systems and the applications (or their agent part). Especially the idea of the applications adapting to the available resources and taking part in the resource allocation process had a big influence on this work. InvasIC was also helpful to gain understanding of all principal parts of a computer system as specialists for these parts were involved. The results and its differences to this work are reviewed in Section 4.4.

Another observation from the list of sub-projects is the large share of hardware development projects.

3

Scheduling of Computer Systems

This chapter describes the starting point of our research: the existing theoretical solutions in Section 3.1, the existing solutions in scheduling practice in Section 3.2. We then take a look at the major differences between the scheduling solutions in theory and practice in Section 3.3. In Section 3.4 we present our own approach and our research directions.

3.1 Developments in Theory

3.1.1 Models

Theoretical work in the area of scheduling usually starts from a model in order to have a defined basis for mathematically proven results. Due to the influential 3-field-notation (see Section 1.2.1) the models used in theory usually consist of a machine model, a job model and an objective function. The most common objective function is to minimize the finishing time of the last job (C_{max}), but also other objective functions are relevant. As the objective functions are closely problem-related, we will look at them together with the problems.

There are also many machine models. This work is dedicated to the scheduling on uniform parallel machines which means that all cores are similar and differ at most by their general speed. Hence we restrict our theory survey to parallel machines with uniform Q or even identical P workers (P, Q here mean the same as in the 3-field problem classification, see Section 1.2.1). Today's modern multicore machines or clusters typically fit very well into the model of identical or uniform workers if the application is compute-intensive. If the memory or other devices are more important, there are influences between the operation of different cores which are usually not part of models in theory. In theory P (parallel identical workers) is the most common model for parallel machines.

As this work is dedicated to the scheduling of parallel jobs we will describe the most important models for parallel jobs. Of course sequential jobs are always part of these models as extreme cases. One of the main differences between the models is how much the scheduler knows about the internals of the parallel job and how much it interacts with internal structures. The first case in which the scheduler only assigns resources to a possibly parallel job without managing the distribution between job-internal tasks is called parallel job scheduling. The second case in which the scheduler (also) manages the internal tasks is mainly DAG-scheduling.

Parallel Jobs

As already noted in Section 1.2.1 there are three typical models of parallel jobs:

- Fixed-size jobs: the degree of parallelism is fixed.
- Moldable jobs: the degree of parallelism is decided by the scheduler when the respective job starts but cannot be changed later.
- Malleable jobs: the degree of parallelism can be changed anytime by the scheduler. This includes preemption by setting the degree of parallelism to zero for some time.

Feitelson and Rudolph [43] introduce the terms malleable and moldable in the way we use them here. Their definition is the same as the one used in the Handbook of Scheduling [93, pages 25-5 to 25-7] which classifies parallel jobs in a similar way as it is done here.

The Handbook of Scheduling also defines a special sub-category of fixed-size jobs where the degree of parallelism is a power of 2 which is less relevant for this work. We can regard the degree of parallelism as some kind of interface. The scheduler decides (if possible) about the number of cores that are given to a job but does not care about their internal use. In the case of fixed-size jobs there is even only one amount of cores which can be used by these jobs. In case of malleable or moldable jobs the scheduler has to know how long a job will run (or other objective-related properties) for different degrees of parallelism. The jobs provide the scheduler with the information how fast they work with different degrees of parallelism. This information is also part of the application $\leftrightarrow$ scheduler interface at least in the used scheduling models. Unfortunately on current standard systems there is no common interface to pass speedup functions (or other objective-related properties) from the application to the system scheduler.

Sequential job scheduling is always an extreme case for these models, in case of fixed-size jobs all jobs can have one as degree of parallelism, in case of moldable or malleable jobs the running time (or other objectives) with more than one core can be at most as good as the running time with one core which makes parallel execution useless.

Due to virtualization it is possible on most modern machines to stop a thread and to continue the execution later at the stopping point (see Section 2.5). If only one thread of a multi-threaded application is stopped, this might lead to heavy performance losses if the other threads are dependent on the results of the stopped thread. Hence stopping only some (but not all) threads of a job is usually not useful at least for fixed-size or moldable jobs. Stopping all threads at the same time can also lead to some performance losses as important cache contents might be evicted while the job is not running. Hence preemption of jobs is always possible on modern computer systems, but it might come along with some performance penalties.

Now we take a close look at the three used models of parallel jobs and the basic problems resulting from their execution on parallel identical cores (P in the 3-field problem classification) with the goal of minimizing the makespan (C_{max} in the 3-field problem classification). We also describe how the job models can be justified from practice.

Fixed-size jobs have a degree of parallelism that is an individual property of each job. This degree of parallelism cannot be changed during runtime (especially not by the scheduler). The basic problem for this kind of jobs is $P|size_j|C_{max}$ (and with preemption $P|size_j, pmtn|C_{max}$) which is the scheduling of the given fixed-size jobs on a given number of cores in order to minimize the finishing time of the job finishing at last (see results in Section 3.1.3).

A practical example of this kind of jobs would be an application with a degree of parallelism fixed at programming time. The different threads are interrelated in a way such that stopping some (but not all) of them (or not running some threads from the start) causes huge performance penalties or even a deadlock. For example the program might contain a barrier at which all threads wait until a certain number of threads arrive. Hence one is forced to run all threads of the job at the same time in order to run the job efficiently. Most applications that allow parallelism can be programmed to fit into the fixed-size job model.

Moldable jobs are similar to fixed-size jobs, but their degree of parallelism can be decided by the scheduler at the start of the job. This degree of parallelism cannot be changed afterwards. Of course the scheduler needs information about the effects of the different degrees of parallelism in order to make a useful decision. The basic problem for this kind of jobs is $P|any|C_{max}$ (and with preemption $P|any, pmtn|C_{max}$) which is the scheduling of the given moldable jobs on a given number of cores in order to minimize the finishing time of the job finishing at last (see results in Section 3.1.3). In case of the basic problem the scheduler needs the information on how long a job runs with each possible degree of parallelism.

This information can either be given as running time for each possible degree of parallelism or as speedup function $s(p)$ which maps a degree of parallelism p to the quotient of sequential running time of the job divided by the running time on p cores.

A practical example of this kind of job would be a numerical simulation. In the beginning the simulation space is partitioned between the participating threads (for example 3D points for a fluid simulation). Then each thread works on the properties (for example pressure and flow) of its points and their change over time. This includes an intensive exchange of data between threads which work on neighboring parts of the simulation space. Stopping one thread also hinders other threads in their progress severely.

Malleable jobs on the other hand are able to change their degree of parallelism (according to orders from the scheduler) without efficiency losses, or the efficiency losses are so small that they can be ignored for the model. For malleable jobs the decision space of the scheduler is much larger than for moldable jobs. As the degree of parallelism can change during the execution of a job, it is necessary to know the progress a job makes when running on p cores during a time of t. This is usually given through speedup functions for each job. Let T be the sequential running time of a job and s its speedup function, then the job is finished after the k-th time interval I_k when $T = \sum_{i=1}^{k} t_i \cdot s(p_i)$ and a degree of parallelism p_i is used during intervals I_i of length t_i. The basic problem for this kind of jobs is $P|var|C_{max}$ (preemption is included as the scheduler can set the degree of parallelism to 0), which is the scheduling of the given malleable jobs on a given number of cores in order to minimize the finishing time of the job finishing at last (see results in Section 3.1.3).

One typical example for such jobs are applications whose threads work on independent (small) workpackages and have nearly no interaction. If a thread is stopped after finishing such a workpackage, it can be stopped without negative effects on the other threads. The number of cores is of course important for the performance of these applications, but typically they gain efficiency with fewer cores rather than losing efficiency (or even deadlock) like fixed-size or moldable jobs. In the future jobs might also be developed to be malleable because they need mechanisms designed to cope with hardware failures.

One must be careful when reading literature that is concerned with malleable jobs because at least two different definitions exist and are widely used. According to Jansen and Zhang [80] the term *malleable* was introduced by Turek et al. [135]. Both articles use the term *malleable* for jobs that would be called moldable in this work.

Task-DAGs

Task-DAGs are directed acyclic graphs in which the nodes are sequential tasks and the directed edges are dependencies between the tasks which are represented by their incident nodes. Parallel jobs can consist of (small) sequential pieces of work and dependencies between these tasks. If one task b has to wait for input of another task a before it can start, b is dependent on a. We write $a \rightarrow b$, so the direction of the edges in the task-DAG indicates the operation flow. Of course if one task is (indirectly) dependent on itself, it can never be done. If there are no dependencies, we get back to the extreme case of sequential job scheduling. The structure of the task-DAG is important for scheduling. Schedules for simple DAG structures like trees are often easier to find than schedules for general DAGs. An important overview paper for DAG-scheduling is written by Kwok and Ahmad [90]. Often the tasks of a task-DAG are called jobs and the dependencies precedence constraints.

DAG-scheduling is typically the work of application-internal schedulers. For example in numerical applications the tasks are the calls of the sequential libraries. The system scheduler usually does not know about the tasks contained in a job or their dependencies as it just schedules threads that execute these tasks.

Further Properties of Job Models

Both model groups can be enhanced by release times, deadlines and further restrictions. A release time can be given for each job and is the earliest time at which we can start working on a job. A deadline can also be given for each job and is the time at which a job has to be finished. Schedules in which a job is not finished until its deadline are infeasible. Another property that can occur in both models is preemption. A job that is preemptable can be stopped at some time and later restarted without performance penalties. Together all parts of the preempted job have the same running time as if the job is run in one piece, but it is not possible to run different parts of the same preempted job in parallel.

More complex models like for example DAGs of malleable tasks are not in the focus of this work. The work of Marchal et al. [101] is an example for the work on this topic.

3.1.2 Basic Solutions and Properties

Basic Solutions

The first basic solution is **McNaughton's Wrap-Around-Rule** [104] for $P|pmtn|C_{max}$. Although the solution is only for sequential, independent jobs, it is often part of solutions for more complex problems. Also the scheduling problem for sequential, independent jobs is an extreme case of both scheduling of parallel jobs and DAG-scheduling. The description is taken from [93, page 3-6] as it is shorter and easier to understand than the original article.

Given a set of k sequential independent jobs with running times p_j and a machine with m parallel identical cores (k, the p_j and m are part of the input) we compute $D = \max\{\max\{p_j | j \in \{1 \ldots k\}\}, \sum_{j=1}^{k} p_j/m\}$. Starting with job 1 we put jobs on the first core until with job i the running time of the jobs on core 1 exceeds D. The remaining part ($\sum_{j=1}^{i} -D$) of job i which would run after D is then "wrapped around" and placed as the job first to run on core 2. As $p_i \leq D$, the two parts of job i never run in parallel. Continuing with the same method all k jobs can be placed on the m cores within a time limit of D which is also the optimal solution for $P|pmtn|C_{max}$.

An important basic algorithm for many heuristics and approximation algorithms is **list scheduling** (as described in many textbooks about scheduling like [93, page 9-2], [9, page 202 ff], [122, page 108 ff] and [23, page 138 ff]). For list scheduling the unfinished jobs are kept in an ordered list. Every time a core becomes idle the first ready job is assigned to that core (often all jobs are assumed to be sequential). Many easy heuristics can be implemented on the basis of list scheduling. For example *largest processing time first* (LPT) can be implemented by sorting the list of jobs beginning with the largest. Approximation algorithms based on list scheduling were investigated as early as 1966 by Graham [54]. It is important to note that the first ready job is assigned to the free core and not necessarily the next job in line regarding the list order (some jobs might be blocked due to precedence constraints). This is even more important for the scheduling of parallel jobs. If for example only k cores are idle at the same time, the first job with a degree of parallelism of at most k is selected. Hence jobs which use many cores in parallel might be overtaken by jobs which are behind them regarding the list order.

Dominant Set of Schedules

The idea of a dominant set of schedules is taken from the book of Baker and Trietsch [9, page 14]. For most scheduling problems there exists a huge and in most cases even infinite number of schedules which fulfill all constraints. Take for example an arbitrary number $r \in \mathbb{R}_+$ as starting time for the first job instead of 0. In most cases it is possible to find some properties of schedules such that for each feasible schedule S there exists a feasible schedule S' that has these properties and that S' is at least as good as S for the given objective. In such a case the set of all schedules which fulfill these properties is called a dominant set of schedules. The restriction to the dominant set of schedules can be useful as it can be much smaller and the reachable quality of schedules is the same.

3.1.3 Parallel Job Scheduling

The scheduling of parallel jobs is still a young and small topic within scheduling although we have already seen (Section 3.1) that the model is justified by and useful in practice. Probably one of the first papers concerned with the scheduling of parallel jobs was published in 1984 by Błażewicz, Weglarz and Drabowski [27]. Even some recent textbooks like *Principles of Sequencing and Scheduling* [9] from 2009 do not cover the topic at all. Although being such a new subject, there is quite some theory in the field of parallel job scheduling, the *Handbook of Scheduling* [93] from 2004 devotes two chapters to the topic and gives more than 150 references in these chapters. In order to keep the overview small, we stick to the parallel job scheduling problems with independent jobs. We will first look at the computational complexity (NP-hardness) of some relevant problems and then take a look at some fast algorithms that may be suitable for practice.

NP-Hardness

The most basic problem of parallel job scheduling is $P|size_j|C_{max}$ which is scheduling fixed-size non-preemptable jobs in order to minimize the total schedule length. Du and Leung [40] prove that this problem is sNP-hard even for a fixed number of cores m if $m \geq 5$ (problem $Pm|size_j|C_{max}$). The proof is done by reduction from 3-PARTITION. The authors also show that the problem is not sNP-hard and only NP-hard for $m = 2, 3$. Only very recently it was shown by Henning et al. [65] that the problem is sNP-hard for $m = 4$. The article of Henning et al. [65] also contains a lower approximation bound of $\frac{5}{4}$ for pseudo-polynomial strip packing which is matched by Jansen and Rau [78], who present an algorithm with an approximation ratio of $\frac{5}{4} + \varepsilon$.

The problem $P|size_j, pmtn|C_{max}$ with additional preemptability of jobs is easier. For all fixed m it is possible to compute an optimal schedule in time polynomial in the number of jobs n as shown by Błażewicz et al. [22]. The idea is to compute all possible combinations of jobs that can run at the same time (their number is in $\mathcal{O}(n^m)$) and assign to each a variable x_i which contains the running time of this combination. With some linear constraints which ensure that all jobs are finished by the end of the schedule the sum of these x_i ($\sum x_i = C_{max}$) is minimized through a linear program. This linear program and thus the optimal schedule can be computed in time polynomial in n. If m is not fixed, $P|size_j, pmtn|C_{max}$ remains NP-hard which is shown by Drozdowski [39]. This is proven by reduction from PARTITION (hence the proof needs an m which can be exponential in the input length). If for each element with size a_j there exists a job for which the number of cores denoted by $size_j$ is equal a_j and a running time of 1, then a schedule with $C_{max} = 2$ exists if and only if the PARTITION instance allows an equal split. Jansen and Porkolab [77] show that the optimal schedule for $P|size_j, pmtn|C_{max}$ can be computed in $\mathcal{O}(n)+$ a term polynomial in m. The result does not conflict with the NP-hardness of $P|size_j, pmtn|C_{max}$ as m as part of the input can be exponential in n, it just shows that $P|size_j, pmtn|C_{max}$ cannot be sNP-hard. For realistic settings the number m of cores is always polynomial in the number of jobs n.

Du and Leung [40] also describe the idea why the more "variable" problems ($P|any|C_{max}$, $P|any, pmtn|C_{max}$ and $P|var|C_{max}$) with moldable or malleable jobs are at least as hard as the fixed-size problems ($P|size_j|C_{max}$ and $P|size_j, pmtn|C_{max}$) if the possible speedup functions are not restricted. However we are not sure if the authors themselves recognized the implication for $P|var|C_{max}$ as malleable jobs do not occur in their paper. If we set the speed for a job to 0 for all numbers of assigned cores q less than $size_j$ and to the same speed for all $q \geq size_j$, then the only useful number of cores to use is $size_j$ in all cases. Hence $P|size_j|C_{max}$ and $P|size_j, pmtn|C_{max}$ are special cases of $P|any|C_{max}$, $P|any, pmtn|C_{max}$ and $P|var|C_{max}$ respectively (remember var contains $pmtn$ by default). The increased decision space can even lead to a harder problem. Du and Leung [40] show that $P|any, pmtn|C_{max}$ is sNP-hard (even for an m polynomial in the input length) where for $P|size_j, pmtn|C_{max}$ only NP-hard was shown (for m polynomial in the input length the problem is even in P). For fixed $m \geq 2$ the problem $Pm|any, pmtn|C_{max}$ with a fixed number of cores m is shown to be NP-hard [40] instead of polynomial time like $Pm|size_j, pmtn|C_{max}$.

Jansen and Porkolab [77] show that an optimal solution for $Pm|var|C_{max}$ can be computed in polynomial time with a linear programming approach. The idea is similar to the one used for the solution of $Pm|size_j, pmtn|C_{max}$, compute all possible configurations of the machine and minimize the sum of running times of the configurations with the restriction that each job is finished by the end of the schedule. Here one just has to include the speedup function into the computation when a job is finished and the number of possible configurations is higher due to the variability of the job's parallelism.

An interesting observation in case of fixed m and $pmtn$ is that the growth of the decision space from fixed-size to moldable leads to a higher complexity, but the further growth from moldable to malleable reduces the complexity.

Instead of such special speedup functions which create highly complicated scheduling problems the speedup functions in reality are often less steep and easier to handle. Hence there is an important part of theory about handling scheduling problems with restricted classes of speedup functions. Often also a maximal degree of parallelism δ_j for each job is given, which is nearly the same as if there are no further speed gains if the core amount grows larger than δ_j.

speedup funct.	*pmtn*/ *m* in input	fixed-size	moldable	malleable
general/none	no / yes	sNP-hard	sNP-hard	–
	yes / yes	NP-hard	sNP-hard	NP-hard
	yes /$m \in poly(n)$	P	sNP-hard	?
	no / no	sNP-hard	sNP-hard	–
	yes / no	P	NP-hard	P
linear	yes / yes	–	?	P
concave	yes / yes	–	?	P

Figure 3.1. Computational complexity of different parallel scheduling problems on parallel identical cores with the objective of minimizing C_{max}.

The two typical restricted classes of speedup functions are linear speedup functions and concave speedup functions. Linear speedup functions are realistic for easily parallelizable applications where the speedup against a computation on one core is just proportional to the number of used cores. Linear speedup functions are also a special case of concave speedup functions.

A function is concave when the straight connection between two points of the function graph is never above the function graph between these two points. This means that given a larger u and a smaller ℓ core amount the speedup of the average core amount can never be below the average speedup with ℓ and u cores or in formula: $s(\beta u+(1-\beta)\ell) \geq \beta s(u)+(1-\beta)s(\ell)$ for $\beta \in [0,1]$. With this restriction it is possible to avoid local minima that can occur with more general speedup functions and compute the optimal result quite fast. Błażewicz et al. [26] showed that the optimal result when *non-integer* core amounts are allowed can be computed in time $\mathcal{O}(n\max\{m, n\log^2 m\})$ for scheduling problems with only moldable or malleable jobs (moldable or malleable is no difference in this case) with concave speedup functions. We describe their approach in more detail, because we speed up and parallelize their result in this work. Given is a set of jobs $j_1,\dots,j_n$, each with a concave, piecewise linear speedup function s_i and an amount of work to do w_i. Their goal is to find a possibly non-integer distribution $(x_1,\dots,x_n) \in \mathbb{R}^n_{\geq 0}$ of the available m cores such that C_{max} is minimized. They define $R = \{(x_1,\dots,x_n) \in \mathbb{R}^n_{\geq 0} | \sum_{i=1}^n x_i \leq m\}$ as the set of possible resource distributions and $U = \{(s_1(x_1),\dots,s_n(x_n)) | (x_1,\dots,x_n) \in R\}$ as the set of resulting possible speeds. They note that U is a convex set due to the concavity of the speedup functions. They use a result of Weglarz [144] that there is an optimal solution in the intersection of U with the straight line l through the origin and $(w_1,\dots,w_n)$. This is intuitively clear as w_i/C_{max} is the least needed speedup of job j_i to reach C_{max}. By interval halving the amount of needed cores for a given speedup $x_i = s_i^{-1}(w_i/C)$ can be computed in time $\mathcal{O}(\log m)$. For each job j_k the authors compute a point $u^{(k)}$ on the line I with an integer component $u_k^{(k)}$. This point is chosen such that $\sum_{i=1}^{n} s_i^{-1}(u_i^{(k)}) \geq m$ and the point on I for the next smaller integer component $u_k^{(k)} - 1$ does not fulfill this condition (I crosses the surface of U between those points). This is done by interval halving for x_k which needs at most $\mathcal{O}(\log m)$ steps, and in each step there are $n-1$ inversions to compute. Thus finding $u^{(k)}$ takes time $\mathcal{O}(n \log^2 m)$. Computing $u^{(k)}$ for each job j_k takes time $\mathcal{O}(n^2 \log^2 m)$. Among the n points $u^{(k)}$ the one with the smallest u_1 is taken (this point also has the smallest u_i as I is a line through the origin). With this point the final solution is computed by using the linear interpolations of the speedup functions. The authors write that the computation of all coefficients of the linear interpolations (between all neighboring points of all speedup functions) can be done in time $\mathcal{O}(nm)$ which leads to the remaining part of the running time. It looks as if for large m it would be more efficient to compute only the needed coefficients which would lead to a total running time in $\mathcal{O}(n^2 \log^2 m)$. For situations where m might be exponential in n this running time would still be polynomial.

As non-integer core amounts are not realistic, Błażewicz et al. [24] also created an algorithm which converts the non-integer schedule to an integer schedule for malleable jobs in time $\mathcal{O}(n)$ (in Section 5.2.2 we describe a version of the algorithm that is adapted to our approach, both versions of the algorithm do not work for moldable jobs). Together with Błażewicz et al. [26] this solves the problem $P|var|C_{max}$ with concave speedup functions (and also for the subset of linear speedup functions) altogether in time $\mathcal{O}(n \max\{m, n \log^2 m\})$. This algorithm is fast enough to be useful in many areas of computer scheduling and was further sped up and parallelized by us (see Section 5.2).

Fast Algorithms

As scheduling itself should not be the main task of computers there is a need for fast scheduling algorithms. For many problems there is no known algorithm which can compute the optimal solution in a reasonable time. Hence also algorithms which might compute a suboptimal solution, but are fast, are of interest (approximation algorithms and heuristics, see also Section 1.2.3). As there is not much that can be proven about a heuristic, heuristics usually are of lower interest in theory. In the case of parallel jobs there are also usually quite fast approximation algorithms with good approximation ratios which also leads to a lower interest in heuristics.

Let us first take a look at the scheduling problems with fixed-size jobs $P|size_j|C_{max}$ and $P|size_j, pmtn|C_{max}$. Although $P|size_j|C_{max}$ looks similar to the 2D strip packing problem by placing jobs in the core × time rectangle, there are some differences. In 2D strip packing it is not allowed to split the rectangles to be placed, but in scheduling there is no need to assign a job to a continuous set of cores. Modern architectures have no connection networks that are built like a line. Usually the networks look like a mesh, a cycle, a tree or an even more complex graph. Turek et al. [135] give an example in which the optimal solution for $P|size_j|C_{max}$ is better than the optimal solution for the analog strip packing problem. Garey and Graham [49] show that each algorithm based on list scheduling can take at most twice as long as the list scheduling algorithm with the optimal list order (they write about a continuous resource instead of cores, but the result applies to $P|size_j|C_{max}$). The proof by Garey and Graham uses the fact that no list scheduling schedule can take more than twice the time of the optimal schedule. If we update the list of times after each core is available for each job assignment, we can perform the list scheduling with a low polynomial running time. Hence list scheduling is a fast approximation algorithm. Johannes [81] shows that there can be no polynomial algorithm with an approximation ratio better than 1.5 for $P|size_j|C_{max}$ (unless $P = NP$) and also gives factor 2 list scheduling approximation algorithms for $P|r_j, size_j|C_{max}$ and $P|r_j, size_j, pmtn|C_{max}$ for the more complex problem with release times. As the release times are part of the input, these approximations also hold for instances where all jobs have the same release time. Jansen [74] presents an $1.5+\varepsilon$ approximation algorithm for $P|size_j|C_{max}$ which runs in time $\mathcal{O}(n\log n) + f(1/\varepsilon)$. This algorithm also works for moldable jobs and even with a mixture of moldable and nonmoldable jobs with the same approximation ratio but with a larger polynomial running time.

If m is not part of the input or if there are restrictions on the size of m, better and faster approximations are obtainable. For the case of m polynomially bounded in n Jansen and Thöle [79] give a factor $(1+\varepsilon)$ approximation algorithm (PTAS) for the problem $P|size_j|C_{max}$. Although too complicated for the application in practical computer scheduling, this kind of algorithms is of high theoretical interest. For the case when m is fixed Amoura et al. [6] give linear time algorithms (with respect to n) to compute the exact solution for $Pm|size_j, pmtn|C_{max}$ and a factor $(1+\varepsilon)$ approximation for $Pm|size_j|C_{max}$. Unfortunately the running time is exponential in m, and thus these algorithms are unlikely to be fast in practice.

There are also approximation algorithms for the scheduling problems for moldable jobs $P|any|C_{max}$ and $P|any, pmtn|C_{max}$. Some authors use the term malleable for the kind of jobs that are called moldable in this work. The speedup functions or the class of speedup functions that are allowed play an important role in this case. Depending on the assigned number of cores q, we use $t_j(q)$ as the job duration and $u_j(q) = q \cdot t_j(q)$ for the usage of core time (often called work $w_j(q)$ in other publications) for job j. Jobs for which $t_j(q)$ is a decreasing function and $u_j(q)$ an increasing one are called monotonic (see [93, Chapter 26]). It is quite natural to assume that the computation time of a job can only go down by adding more cores (otherwise leave the additional cores idle) and the overhead (and thus the accumulated time on all used cores) can only go up. Jansen and Land [75] show that $P|any|C_{max}$ is still sNP-hard for monotonic jobs. The Handbook of Scheduling [93] has a good overview of approximation algorithms for scheduling problems with malleable and moldable jobs in Chapter 26. If speedup functions are given as lists of different speeds for every possible core number for each job, then m is implicitly polynomially bounded in the input length as we have m values for each job in the input.

When the scheduler has fixed the degree of parallelism p_j for each job j, we get a lower bound of the execution time by $b = \max\{\sum u_j(p_j)/m, \max_j t_j(p_j)\}$. The only thing that remains to do in this case is to use an approximation algorithm for the scheduling problem for fixed-size jobs. We will now take a look at the results of this approach of using fixed-size job scheduling solutions together with estimations of the degree of parallelism. Turek et al. [135] give an easy 2 approximation algorithm for $P|any|C_{max}$ that has no prerequisites on the speedup functions. For each job they start with using the amount of cores that results in the lowest usage of core time, and based on this core assignment they use a factor 2 approximation for the resulting fixed-size problem. In further steps they assign more cores to the job with the highest running time until it reaches the next lowest usage of core time with a shorter running time and use the fixed-size approximation algorithm again. The repeated adding of cores to the job with the highest running time stops when this job cannot use more cores. The fastest among those (at most nm) computed schedules has at most the core time usage for each job as the optimal schedule has. With the approximation for the resulting fixed-size problem this results in a factor 2 approximation algorithm for $P|any|C_{max}$. Ludwig and Tiwari [98] further speed up the schedule computation as they minimize $b = \max\{\sum u_j(p_j)/m, \max_j t_j(p_j)\}$ only once and then pass the jobs with the computed p_j to a fixed-size approximation algorithm. This also leads to a factor two approximation but with a total running time in $\mathcal{O}(n \log^2 m) + L(m,n)$ (instead of $\mathcal{O}(nm) \cdot L(m,n)$ from the algorithm of Turek et al. [135]) where $L(m,n)$ is the running time of the used factor two approximation algorithm for $P|size_j|C_{max}$. The algorithms of Turek et al. and Ludwig and Tiwari rely on the fact that a list scheduling solution for $P|size_j|C_{max}$ has an upper bound of $2b$ for C_{max}. Garey and Graham prove that implicitly in their work [49] already mentioned above. With the result of Johannes [81] (a non-preemptive list scheduling algorithm is a factor 2 approximation algorithm for $P|r_j, size_j, pmtn|C_{max}$) we even get a factor 2 approximation of $P|any, pmtn|C_{max}$.

A better factor $\frac{3}{2} + \varepsilon$ approximation algorithm is given by Mounie et al. [106] for $P|any|C_{max}$ and monotonic jobs. The authors use a knapsack approach to fill two shelves, one of length d and one of length $d/2$. d is a guess of the length of the optimal schedule which is rejected if it is too small (this enables the authors to approximate the optimal schedule length). The already mentioned work of Jansen [74] produces the same approximation in the general case (general speedup functions). Jansen [73] describes an asymptotic fully polynomial time approximation scheme (AFPTAS) for $P|any|C_{max}$. For the case of a fixed m Jansen and Porkolab [76] show a factor $(1+\varepsilon)$ approximation algorithm which runs in time $\mathscr{O}(n)$. Jansen and Land [75] introduce a new technique called compression (reducing the degree of highly parallel jobs slightly) for the case of monotonic jobs. This technique is used by them for an FPTAS for the case $m \geq 8\frac{n}{\varepsilon}$ with running time $\mathscr{O}(n \log m(\log m + \log \frac{1}{\varepsilon}))$. They also use the compression technique to speed up the knapsack approach of Mounie et al. [106] in order to reach a factor $\frac{3}{2}+\varepsilon$ approximation for $P|any|C_{max}$ and monotonic jobs with a running time linear in n and $\log m$.

No approximation algorithms for the problem $P|var|C_{max}$ could be found for general speedup functions or concave speedup functions. Algorithms for some more specialized problems are described in Chapter 26 of the the Handbook of Scheduling [93]. There seems to be not much need as the optimal solutions can be computed quite fast for many kinds of speedup functions. It is also possible to use the approximation algorithms for $P|any, pmtn|C_{max}$ and thus not using the jobs' adaption capabilities to the full extent. The approximation ratios may be worse than those for $P|any, pmtn|C_{max}$ because the optimal schedule for $P|var|C_{max}$ can be shorter than the optimal schedule for $P|any, pmtn|C_{max}$. Our results for the $P|var|C_{max}$ problem are described in Section 5.2.

3.1.4 DAG-Scheduling

Research in DAG-scheduling is much larger and has a much longer history than the research in parallel job scheduling. This is probably due to the fact that DAG-scheduling has applications in production scheduling and is thus not dependent on the existence of parallel computers. The work by Kwok and Ahmad [90] gives an overview about the wide spectrum of results in DAG-scheduling for multicore machines. In another overview article of Kwok and Ahmad [89] different DAG-scheduling solutions are compared by benchmark results. We only take a brief look into DAG-scheduling as it is not in the main focus of this work.

NP-Hardness

Ullman [136] shows that $P|prec, p_j = 1|C_{max}$ is NP-hard, hence the general problem with arbitrary job lengths $P|prec|C_{max}$ is also NP-hard. The NP-hardness also holds for uniform and unrelated cores ($Q|prec|C_{max}$ and $R|prec|C_{max}$).

As the general problem is NP-hard, it is an obviously interesting direction of research how much one has to restrict the problems in order to get a polynomial time scheduling algorithm that finds the optimal solution. The problem $P||C_{max}$ for parallel jobs without precedence constraints is NP-hard for all core numbers $m \geq 2$. This makes the DAG-scheduling problem NP-hard for all cases with general job lengths, parallelism and classes of precedence constraints that allow an unrestricted number of independent jobs. Ullman [136] also shows that the problem $P2|prec, p_j = 1, 2|C_{max}$ (two cores, job length one or two) is NP-hard thus reducing the possibility of polynomial-time solvable problems to those with unit job length (for general precedence conditions).

On the other hand Kwok and Ahmad [90] note in 1999 that there are only 3 results known for optimal DAG-scheduling in polynomial time. The three problems are $P2|prec, p_j = 1|C_{max}$ and $P|prec, p_j = 1|C_{max}$ for the special cases that the precedence conditions form a tree or an interval-ordered DAG.

Fast Algorithms

One of the first articles about scheduling DAGs of jobs was published by Hu [68] in 1961. Hu showed that list scheduling with the ordering "highest label first" leads to the optimal solution for $P|intree, p_j = 1|C_{max}$. Graham [54] shows that list scheduling with any list order produces an approximation algorithm with approximation ratio at most $2 - 1/m$. He also looks at the "quite reasonable" strategy in which a free core always starts to execute the ready job which heads the longest chain of unexecuted tasks. For this strategy he notes that the approximation ratio of this algorithm cannot be better than $2 - \frac{2}{m+1}$.

3.1.5 Energy Scheduling

Given the importance of energy usage of computing systems a large amount of work was done on how to reduce the energy usage by scheduling. According to Albers et al. [4] there are two key mechanisms that can be used by scheduling in order to produce energy savings: speed scaling (in this work: dynamic voltage and frequency scaling DVFS) and sleep states (switching to sleep modes when the system is idle). We only look at speed scaling in this work as it seems to be more densely related to the scheduling of parallel jobs.

A very important work about speed scaling is by Yao et al. [148] which introduced the well-known YDS algorithm. Yao et al. only look at sequential machines and of course sequential jobs. For the power function they only assume that the consumed power is a convex function of the core's speed which can be selected from $\mathbb{R}_{>0}$. The work was later expanded by Li and Yao [94] to the case when only discrete frequency levels are available. The obvious next step for energy scheduling is to schedule sequential jobs on a parallel machine. For the problem with common release times and deadlines, sequential preemptable (and migratable) jobs, Chen et al. [30] give an $\mathcal{O}(n \log n)$ algorithm which computes the energy-optimal schedule. They also look into the problem when tasks cannot be migrated from one core to another. Albers et al. [4] work on the more general problem when jobs can have individual release times and deadlines and they also look into related online problems. As an important restriction they assume that jobs can be preempted but cannot be migrated. With this they reach an optimal polynomial-time algorithm for the case of unit-size jobs and a special deadline/release time structure (later release time of a job compared to another implies a later or equal deadline of this job compared to the other). The special structure is needed because they prove afterwards that the problem becomes NP-hard without it. For arbitrary sized jobs they give two approximation algorithms for the case of a common release time and the special structure. They also give an online-algorithm with constant factor competitive ratio for the case of unit-sized jobs with the special structure.

Energy scheduling of parallel jobs has got less attention than the energy-efficient scheduling of sequential jobs, but still there are some works (mostly heuristics). Kong et al. [87] present heuristics for the energy minimizing scheduling of fixed-size and moldable jobs (common release time and deadline) based on level-packing. They assume that the power usage of a core is proportional to f^{α} when it runs with a speed of f with $2 \leq \alpha \leq 3$. The paper is later enhanced by Xu et al. [147] (two authors are also authors of [87]) with ILP solutions for the level-packing and a discrete frequency model. Chan et al. [29] and Fox et al. [46] look into online problems regarding the energy-efficient scheduling of parallelizable jobs. Heuristics for the energy-efficient scheduling of moldable streaming tasks are investigated by Melot et al. [84].

3.1.6 Parallel Computation of Schedules

As we have already seen the scheduling of parallel machines and the scheduling of parallel jobs gained much attention in scheduling theory. Interestingly the parallel computing of schedules has got much less attention. There are only few works about computing schedules in parallel. Helmbold and Mayr [61], [62], Dolev et al. [38] and Sunder and He [126] focus on polylogarithmic scheduling of sequential unit-sized tasks in the case of either special precedence structures or only two executing cores. Most of these works also show that problems with different precedence constraints become P-complete. The Ph.D.-thesis of Stadtherr [124] gives a good overview of the results reached in the field of parallel schedule computation.

3.1.7 Further Enhancements

There are many other things relevant for scheduling, but they are too small or too loosely connected to this work to justify a detailed description. Little could be found in scheduling theory about scheduling with respect to memory hierarchies. Of course one can treat memory bandwidth or cache space just as another additional resource and use scheduling results for additional resources like the work of Garey and Graham [49], but caches and memory connections are usually no global resource but are shared between some but usually not all cores. Additionally most systems do not have mechanisms to assign fixed portions of cache and memory bandwidth to a certain computation. On the other hand there is some work about optimizing cache misses by scheduling (but the execution time and other parameters are not used as objectives), for example the work of Blelloch et al. [11]. We look in more detail at the scheduling methods to improve the efficiency of the memory in Section 3.2.2 as most of these methods use a practical approach.

Another widely researched topic is online scheduling. Some of the already mentioned articles also include online scheduling algorithms, for example the work of Johannes [81] about online scheduling of fixed-size parallel jobs. This work mainly focuses on malleable jobs which can be easily adapted to new situations, and hence online scheduling of these jobs is less important (but a relevant topic for future work).

3.2 Developments in Scheduling Practice

In this section we look at the the way scheduling is used in practice. One of the most used scheduling systems in practice is probably the thread scheduler of the Linux-kernel. We take a closer look at this scheduler in Section 3.2.1. The already mentioned practical results of scheduling to increase the efficiency of the memory hierarchy (or short memory scheduling) are described in Section 3.2.2. We also take a brief look at other scheduling solutions used in practice in Section 3.2.3.

3.2.1 Linux Completely Fair Scheduler

The Linux Completely Fair Scheduler (CFS) has been the scheduler of the Linux Kernel for non-real-time processes since kernel version 2.6.23. As such it is an important example of a real world scheduler. Due to its importance a lot of effort is put into the Completely Fair Scheduler. Hence we will look at this scheduler as a good example for scheduling practice. The description of the Linux Completely Fair Scheduler is mostly taken from the Linux-kernel books of Mauerer [102] and

Love [96].

The Linux scheduler has 4 goals:

- Dividing the available core time between different jobs in a fair manner. Different priority levels should be recognized for the fairness.
- Trying to run each job within a certain interval to avoid starvation.
- Reducing task switch overhead (for example the loss of cache locality).
- Providing interactive jobs with a low latency.

Although CFS has some desirable properties for operating system schedulers which are explained later, there is no precise objective function for the Linux scheduler. According to Love [96] CFS uses the approach of fair queuing known from queuing theory with additional priorities (Demers, Keshav and Shenker [37] describe a popular algorithm for fair queuing). The kernel has 140 different priority levels for jobs from 0 to 139 where a lower number is equivalent to higher priority. The range from 0 to 99 is reserved for real-time processes. The known nice values from -20 to $+19$ for normal processes are mapped to the range 100 to 139.

General Idea

The goals of the Linux CFS are similar to the goals of a router in a packet-switched network. The router also has to share the available connection bandwidth fairly, starvation (or high latency) should be avoided and interactive connections (for example SSH-connections) should have a low latency. The network scheduler in the router always transfers a full packet which is similar to the operating system scheduler awarding each job a certain amount of uninterrupted time in order to reduce task switch overhead. A popular fair queuing algorithm for routers of packet-switched networks was introduced by Demers, Keshav and Shenker [37] and refined by Greenberg and Madras [56].

If a resource has to be shared fairly between n consumers (all with the same priority), an easy way is to give each customer $1/n$ of the resource. For example a water stream can be shared between farmers in this way. Unfortunately there are resources which do not allow continuous sharing, for example only one packet can travel over a line at the same time, or only one job can run on a single core machine at any given time.

An obvious adaption to exclusive resources would be to serve each customer exclusively for a very short time and then switch to the next customer so that each customer gets his fair share within a short time frame. Unfortunately in computer scheduling a too short service time leads to inefficiencies, the switching overhead has to be accounted for more often, and caches lose efficiency because the locality of computations is reduced.

Hence a fair scheduling system should be able to cope with larger workpackages of even different size as a job can block during a longer workpackage or an interrupt can happen. In the easiest case when all jobs are runnable and all have the same priority, CFS accumulates the running time of each job and always schedules the one with the lowest accumulated running time.

In order to handle priorities, the decision-relevant running time is the virtual running time which is computed from the real running time by dividing by priority-specific weights. If a job has a low priority it gets a low weight and thus gains more virtual running time for the same amount of real time on the core. Hence it is sooner moved away from the core by the scheduler or has to wait longer to be scheduled again than jobs with higher priorities.

If a job blocks because it has to wait for input, it does not accumulate further virtual running time. Of course a job that often blocks because it has to wait for input is probably an interactive job and should get fast service by the system once input arrives. But using the real small virtual running time of it for scheduling decisions might lead to starvation of the other jobs if the blocked period was quite long. Hence CFS takes the smallest virtual running time of the runnable jobs minus some constant as lower bound of the virtual running time of a job that gets unblocked. This provides an advantage for probably interactive jobs without leading to starvation for other jobs.

The task switch overhead is bounded by using a minimal time quantum during which a job is not replaced by another one. If there is a large number of different jobs present in the system this inevitably might lead to starvation.

Implementation and System Integration

The Linux scheduler has 5 policies for scheduling entities:

- `SCHED_NORMAL` for the normal jobs.
- `SCHED_BATCH` for CPU-intensive batch jobs.
- `SCHED_IDLE` for low importance jobs.
- `SCHED_FIFO` for first in, first out scheduled real-time jobs.
- `SCHED_RR` for round robin scheduled real-time jobs.

`SCHED_NORMAL`, `SCHED_BATCH` and `SCHED_IDLE` are handled by the Completely Fair Scheduler, `SCHED_FIFO` and `SCHED_RR` by the real-time scheduler. As long as there are runnable real-time jobs these jobs are executed and the other jobs have to wait. Hence real-time jobs are not important to describe the CFS. For the remainder of this section we have to distinguish between the global Linux scheduler and CFS which is part of the Linux scheduler, but the Linux scheduler also contains other schedulers and general functionality and data structures supporting all schedulers.

An important decision for the design of a scheduler is whether the scheduler schedules processes or single threads. For the Linux scheduler the scheduling entities are threads which coincide with processes for singe-threaded processes. There is no special handling for threads of the same process.

For different applications the Linux scheduler needs the load of a job or a group of jobs. As no other information is available, the load is directly defined through the priority of the jobs. The priority of (non-real-time) jobs which are set by the nice values between -20 and $+19$ are mapped to the range 100 to 139. The according load for priority 100 is 88761, for 139 it is 15. The load values for each priority are set such that one additional priority level means multiplying the load value with a factor of 1.25.

The Linux scheduler maintains one run queue for each core. All jobs (all of them are threads) can only be part of one queue. The Linux scheduler computes the sum of loads for each queue. CFS regularly examines the load of the different queues of the system and moves jobs in order to balance the load between them. Core affinities are respected in this process and other things (for example NUMA-affinities) are regarded to reduce the side effects of the movements. The load balancing between the different queues is quite complex and seems to produce suboptimal results sometimes. The work of Lozi et al. [97] named "The Linux Scheduler: a Decade of Wasted Cores" describes some problems with the balancing.

If there is more than one job in a queue, CFS has to decide which one to run. Every time CFS makes a decision, it selects the job with the lowest *virtual* running time to run next. The jobs in one queue are kept in a Red-Black-Tree in order to be able to select the one with the lowest virtual running time fast. Also insertions and other operations are fast within that data structure. In order to give more running time to jobs with higher priority, the virtual running time of a job is updated after it has run some time by adding the real running time multiplied by the load value of a priority 120 job and divided by its own load value. With this the virtual running times of jobs with high priorities increase slower because of their higher load values and thus gives them more running time. If there are two jobs in the queue and one is one priority level higher than the other, their load values differ by a factor of 1.25, and hence the job with the higher priority will get a fraction of $\frac{1.25}{1+1.25} \approx 0.56$ of the running time and the other a fraction of ≈ 0.44.

The behavior of CFS is influenced by many different parameters. The `sched_latency_ns` parameter specifies the time interval within which each job should run once and the `sched_min_granularity_ns` which is the minimal running time of a job with average priority should get at once. If there are too many jobs to fulfill both restrictions, CFS keeps `sched_min_granularity_ns` and thus prioritizes efficiency over latency. If a new job is put into the queue or a blocked job is unblocked, their virtual running time is set relative to the minimal virtual running time of the queue with some additional enhancements.

CFS is not only able to distribute the processing time fairly between different jobs but also between groups of jobs in order to gain a fair distribution between users or other entities.

Summary of the Linux Completely Fair Scheduler

The Linux CFS scheduler is widely used as it is part of Linux and hence deployed on a large number of different systems. It also is highly relevant for the system performance and user satisfaction. Thus it is likely that there is a lot of effort put into its development in order to improve the results.

On the other hand the Linux CFS scheduler knows little about the objectives of the applications it schedules. The only knowledge the scheduler gets about these objectives are the priorities. Without these objectives the only thing a scheduler can do is to provide fairness (with regard to the priorities) between the different threads, which is done by the Linux scheduler.

When moving threads due to load balancing the Linux scheduler tries to conserve cache- and NUMA-locality if possible by doing load balancing in a hierarchy (Zhuravlev et al. [153]). The hierarchy reflects the common caches and NUMA-nodes of the cores by building appropriate groups. By trying to move a thread in the lowest possible level of hierarchy (within its group), most of data localities can be preserved. Contention of shared resources is not taken into account by the Linux scheduler (Zhuravlev et al. [152]).

The Linux scheduler has no interface to get information from the application or give information to the application. This leads to poor coordination between the applications and the Linux scheduler. An example of the poor coordination is the work of Harris et al. [60] which shows large improvements of the performance of two computing intensive applications by an additional coordination between them.

Also the distribution of work between the different cores seems to leave some room for improvement. The already mentioned work of Lozi et al. [97] named "The Linux Scheduler: a Decade of Wasted Cores" demonstrates how changes might improve the performance. Lozi et al. [97] measured performance degradations for typical Linux workloads between 13 and 24% because of "performance bugs" in the Linux scheduler related to the load balancing between cores.

3.2.2 Scheduling in the Memory Hierarchy

The amount of work about utilizing the memory hierarchy more efficiently is quite huge. We see four main areas of work about scheduling in the memory hierarchy: constructive cache sharing, avoidance of destructive cache sharing, management of memory bandwidth usage and NUMA-awareness. Of course combinations of all areas exist.

Constructive cache sharing was noticed by Blelloch and Gibbons [12] who showed that different threads of the same application can profit from commonly using cache contents. Tam et al. [127] and Chen et al. [31] apply the constructive sharing to different applications and show performance improvements.

The opposite effect, that threads can interfere with the cache usage of other threads can even occur on single core machines (see Agarwal et al. [2]). Different methods can be used to circumvent that problem: scheduling approaches (see for example Zhuravlev et al. [152]) or direct hardware methods like Intel's Cache Allocation Technology (as described in a white paper from 2015 [71]).

Of course there is also negative interference if different threads on different cores compete for the same shared memory bandwidth. Eklov et al. [41] introduce a method to classify different applications regarding their performance degradation if bandwidth is used by other jobs. Yun et al. [150] and Cheng et al. [32] introduce mechanisms on how to manage the bandwidth usage by slowing down or pausing threads that use too much bandwidth.

NUMA-aware placement has a long history, one of the oldest articles addressing this problem is by Bolosky et al. [14] from 1989. A more recent approach to control NUMA-effects through scheduling is described by Dashti et al. [35].

A survey of the different techniques of dealing with the shared components of the memory hierarchy of multicore processors is published by Zhuravlev et al. [153]. An extreme approach of dealing with caches is proposed by Boyd-Wickizer et al. [17] who propose to store memory objects in caches and move the accessing threads around such that they work on the core belonging to the appropriate cache when they access a certain object.

3.2.3 Other Examples for Scheduling Practice

A relevant topic in scheduling (which is not in the main focus of this work) is the management of large clusters or even computing centers. Feitelson et al. [44] give a survey on the theory and practice of scheduling supercomputers. Many different strategies are implemented and tested which use the knowledge about future jobs given through a long job queue, for example backfilling (see Mu'alem et al. [107]).Supercomputers are often even combined to grids which require even different scheduling strategies (see Hamscher et al. [59]). Here we can also see examples of hierarchical scheduling structures (a global scheduler for the grid and one for each supercomputer).

As it is very complicated to program for large supercomputers, frameworks following the MapReduce programming model (see Dean and Ghemawat [36]) were built, for example Spark [151]. A similar development occurs for normal parallel machines. In order to ease parallel programming on these machines, libraries like OpenMP [34] or QUARK [149] (the DAG-scheduler behind PLASMA [21]) are developed. These libraries and frameworks usually contain their own scheduling.

Some more scheduling results are described in Section 4.3.3 where we look especially at hierarchies of schedulers and decision distribution.

3.3 Gap between Theory and Practice

After the presentation of scheduling results from theory and practice let us now take a look at the differences between them. A scheduling problem in theory is typically described with the 3-field problem classification $\alpha|\beta|\gamma$. A scheduling algorithm working on this problem gets precise information about the machine, the jobs and the objective. In contrast the Linux scheduler (for example) has little information about the objectives of the users or the system owner, just the priority values. Also the information about the job's amount of work and future behaviour is unknown to common real system schedulers, although this information might exist. For example an application which has to sort an array usually knows the size of the array and thus the work amount of the used sorting algorithm, but without an interface it is impossible for such an application to give the information to the scheduler. This problem also occurs the other way round. The global system scheduler knows about the amount of threads in the system. If there are several parallel applications which assume to be alone on the machine, all of them might use as many threads as there are cores on this machine. This might lead to a lot of context switches and resulting cache context losses and might be far less efficient than partitioning the available cores into disjoint sets and giving each application such a set (see for example Harris et al. [60]). Especially there is no common system to inform jobs if they can use the whole machine or if they should restrict themselves to parts due to the presence of other jobs. In theory malleable and moldable jobs are a solution for this problem as their degree of parallelism can be adjusted with regard to the load situation of the system.

The efficient common usage of shared memory hierarchy components is an important topic of scheduling practice. Unfortunately there seems to be no widely accepted model of memory hierarchy behaviour in scheduling theory. In scheduling practice many different heuristics are used to optimize the usage of the mem-

	Theory	Practice
job information	detailed	shallow
clear objective function	yes	often ambiguous mixture
adaptivity to system load	moldable jobs malleable jobs	virtualization → bad performance
influence of shared memories	not considered	investigated, but different approaches

Table 3.1. The gap between scheduling theory and practice

ory hierarchy. We summarize our findings about the gap between theory and practice in Table 3.1.

3.4 Own Approach

Most of the research for this work was done while the author was part of the InvasIC-project and concerned with scheduling within this project. Hence the perspective on scheduling is wide and application-oriented as the goal of InvasIC was to implement a real, working system. On the other hand efficiency was an important goal for InvasIC, and the system design of the InvasIC system was different from existing systems in many ways. Thus there were no existing scheduling solutions or performance models for this kind of system and the existing theory also had to be adapted. As the design of the InvasIC system is a conclusion of many current trends in system design (see Section 2.2.3), the main parts of this work reflect those trends in order to get results that are not InvasIC-specific. Hence most of the research was done with the application on current systems (or their further developments along those design trends) in mind, and experiments were also conducted on current systems.

3.4.1 Development Paths

If one wants to design a scheduling system for a real system with parallelism and abstraction layers, one has to decide where to make the decisions and how to get the needed information there. It is also necessary to decide how to make the decisions or more specifically which scheduling algorithm should be used. In Section 2.2.1 we discussed the reasons (Power-Wall, ILP-Wall and Memory-Wall) which hinder sequential computing systems to gain more computing power. Of course one has to keep an eye on these issues for parallel computing systems as well.

- The instruction level parallelism wall is far less important as soon as parallel jobs are used on multicore systems. With explicit parallelism available one can overcome the limitations of instruction level parallelism, hence this problem can be considered more or less solved.
- Increasing transistor density (heat issues) and the increasing cost of eletricity make power consumption still an important topic. Also for devices depending on batteries power consumption is important.
- Also on parallel systems memory accesses cause waiting times. An additional challenge on parallel systems comes from the fact that usually several cores are sharing memory interfaces and last level caches.

As computers are getting more and more complex, the decision space for schedulers becomes larger as well as more decisions have to be made. Hence the decisions have to be split up in order to reduce latency, overloaded decision makers and high efforts for the transfer of system status data. The distribution of decision makers and local decisions are also a key feature of InvasIC (see Section 4.4), but we see our development paths not as InvasIC-specific. We identify 4 main research directions for the scheduling of parallel computing systems:

1. Where are the scheduling decisions made and how do the different schedulers interact? This includes communication between different schedulers as well as the parallelization of scheduling (see Chapter 4).
2. How are the decisions made? This is the main connection to the existing scheduling theory and is mainly focused on efficient decision making (see Chapter 5).
3. How can the cache hierarchy and other parts of the memory subsystem be efficently used? This is especially important as their usage is shared between different cores (see Chapter 6).
4. How can energy consumption be reduced and the Power-Wall be avoided (see Chapter 7)?

This work differs from many other works in the area of scheduling, in the way that it does not only focus on the decision making process of scheduling itself but also looks at the system properties (decision space, properties and constraints, goal and available information) in order to adapt these to get a better scheduling result for the whole system.

3.4.2 Methodology

The research field on scheduling is huge. So one task is to get some kind of survey about which parts can be used for which goal. When looking at such a big field in order to produce useful results for a special project like InvasIC, it cannot be expected to produce a whole new branch of theory and practical results. Instead, we followed the development paths as described in Section 3.4.1 and tried to gain smaller results along these paths while producing a big picture.

Like in many parts of computer science there are two main ways to get meaningful results:

1. Based on a mathematically defined model, one proves certain properties of algorithms by mathematical conclusions.
2. Based on an implementation of an algorithm/technique one proves properties of it by experiments.

The first approach heavily depends on simple models which are also realistic. Simple models require simplifying assumptions which might differ a lot from reality (see our introduction to models in Section 2.6.2). On the other hand models need less implementation effort than real systems as one can just define things instead of implementing them. This makes it possible to make statements about general classes of systems and applications which are not possible by experiments. Also systems that do not exist yet can be examined by the use of an appropriate model. Successful models in engineering can lead the development process for real systems such that these models change the real systems (see below). Hence model-based research can also overcome the problem of path dependency (see Section 2.2.3) as it makes it possible to compare different systems without having to put in a big effort to implement them.

The experimental approach has the advantage that modelling errors are less likely. Modelling problems might still influence the experimental design, but if one uses realistic inputs on realistic machines, algorithms that perform well in experiments will also be good in practice. In most cases it is not possible to test every input that can occur in practice, so usually experiments are not that good to prove properties of an algorithm for a wide range of inputs. Also the experimental approach can be very costly as everything that is needed for a realistic setup has to be bought or implemented.

Algorithm engineering (see Section 2.6.1) combines the prediction strength of both approaches. Algorithm engineering gives performance guarantees through theoretical analysis and results for practical performance through experiments. Unfortunately it also needs the combined effort of both approaches.

Simulation stands somehow between experiments and mathematical conclusions. One needs a model, but it can be more complicated than models used in theory. At the same time the implementation effort is reduced compared to experiments. The way of how results are proven in simulations is much more similar to experiments than to mathematical methods.

So which methods should be used for which kinds of scheduling algorithms?

It is clear that experimental approaches require more effort when more parts of the system to test differ from existing systems. So the investigation of scheduling algorithms that require totally different systems to be useful is much easier with the theoretical approach than with the experimental one. The amount of different new components that have to be implemented for experiments certainly depends on the layer of abstraction (as defined in Section 2.5; hardware, operating system or application software) in which the scheduling algorithm is implemented and how it interacts with scheduling on other layers.

The most easily accessible layer of scheduling for experiments is the application layer. As long as the application runs on a machine without other applications competing for resources, it is only necessary to adapt the application and implement the scheduler in order to do experiments. Hence an in-application scheduler is a perfect fit for the experimental or AE approach.

If the in-application scheduler is able to comply with different resource allocations of the operating system (because there are other applications on the system), things get more complicated. The behavior and the resource demands of the other applications are now part of the experiment similar to the inputs. Also the operating system scheduler and the interface between application and operating system play an important role in this case. This makes meaningful realistic experiments much more difficult.

These problems (application behavior as input and the importance of interfaces) are even more important for scheduling on the operating system layer. The relevant scenario for an operating system scheduler is the scheduling of different applications. Experiments for a scheduler developed for a new system design (for example InvasIC) require the implementation of a new operating system, new applications and other components. Hence realistic experiments require the work of a whole group of people in this case.

In real systems we have decisions made by applications, the operating system scheduler and different hardware components. Decision making on the hardware layer is not studied itself in this work. For the combined scheduling process over all layers the effort for an experiment is huge: one has to adapt the applications to a new scheduler, develop an application scheduler, new interfaces to the operating system, a new scheduler on the operating system and maybe even new hardware parts (and also new hardware interfaces). All these developments have to be good enough to enable meaningful experiments. Of course this is beyond the scope for a single person and also shows the effect of the path dependency (as described in Section 2.2.3) in experiment-based computer science.

So research with limited resources is only possible based on models for operating systems schedulers or scheduling systems. The used models are often specific to the examined problem and are described together with them.

The models used for scheduling describe the reality of existing systems more or less accurately, but they also have another function. Implementing new real systems is expensive, modelling new systems is cheap. For example, if an interface enhancement proves itself to be hugely beneficial within a model, it is likely that it will also be implemented in practice. If, more general, some kind of model is successful in the sense that simple scheduling algorithms lead to an efficient schedule, this model might also influence the further development of applications, operating systems and hardware. So successful scheduling models can shape future computer systems. This is a retroactive effect of modelling which occurs in computer science but is impossible in natural sciences. For example programming languages usually have a formal structure in order to enable compilers to parse them. Here the theory of formal languages created its own application field. Another example is number theory which prepared the basics for parts of the modern cryptology.

3.4.3 Detailed Research Directions

In Section 3.4.1 we identified the four global development directions: distribution and interaction of schedulers, efficient solution computation, efficient usage of the memory hierarchy and energy-efficiency.

Our target systems are parallel computing systems with parallel jobs with flexible adaptivity and the goal of high efficiency. It is immediately clear that the theory of parallel job scheduling will provide an important basis for the global approach. Given that we assume an increase in job flexibility in future systems, the scheduling of malleable jobs seems to be a fitting specialization. Of course being fast is an important condition for the resulting scheduling algorithms. This can be reached for example by parallelization. On the other hand in practice there are no existing malleable jobs as current operating systems usually do not tell their applications which degree of parallelism they should use. Hence it is also important to show that it is possible to develop efficient malleable jobs and to show the benefits of adaptivity. As it is not that easy to implement a new system scheduler which also needs completely new applications and interfaces, we restrict ourselves to the algorithm development of the schedulers.

In practice not all scheduling problems might fit into the model used for malleable job scheduling or other relatively easy models known from scheduling theory. Hence there is a need for useful heuristics in case of complicated systems which do not allow a simple model.

Another topic is the efficient usage of the memory hierarchy. There seems to be little work in scheduling theory about that topic, much more is done in practice. As long as there is no widely accepted model for scheduling in the memory hierarchy, we can only approach this topic from the experimental side. Compared to experiments with application-internal schedulers, experiments with system schedulers are much more difficult and labor-intensive. Hence we approach the topic of efficient scheduling of the memory hierarchy by trying to improve application-internal schedulers. The improved understanding of the properties of the memory hierarchy might even lead to ideas how to build such a model for memory scheduling.

In contrast to the situation of memory-efficiency, there are widely accepted models of energy-efficiency in scheduling theory. Hence it looks promising to investigate the energy-efficient scheduling of malleable jobs as there is still little work about energy-efficient scheduling of parallel jobs.

Given the gap between scheduling theory and scheduling practice, it is beyond the scope of this work to develop models and scheduling algorithms that bridge that gap for upcoming systems. But working on both sides might bring improvements that close this gap in the future and are a progress in theory and in practice. For more about our expectations about the scheduling of future systems see Section 8.2. Our results from this approach are described in Section 3.4.4, which also contains the references to the more detailed descriptions within this work.

3.4.4 Results

The main findings of our research are presented in the remaining Chapters. Chapter 4 discusses the place where a scheduling decision should be made especially in terms of abstraction hierarchy and locality and how to get the needed information there. How these decisions then are made is discussed in Chapter 5. The efficient handling of memory hierarchies and memory accesses is the topic of Chapter 6. In Chapter 7 the topic of energy consumption and the influence of scheduling is handled. Some results (with our contributions) that are part of this work have already been published:

- *Efficient Parallel Scheduling of Malleable Tasks* (joint work with Peter Sanders, [116]) In this work a scheduling algorithm for malleable jobs (with some restrictions for speedup functions) is developed that is faster than previous ones and even parallelizable. The observation that previous algorithms for this problem can be sped up was made by Jochen Speck, and Peter Sanders contributed the idea for the parallelization. The remaining parts of the paper were done in close collaboration. This result is described in more detail in Section 5.2.5.
- *Energy Efficient Frequency Scaling and Scheduling for Malleable Tasks* (joint work with Peter Sanders [117]) Here we develop a scheduling algorithm for malleable jobs (with some restrictions for speedup functions) which computes the schedule with the lowest energy consumption which finishes all jobs by a

given deadline. This article uses a continuous approach to a discrete problem similar to the previous article. The idea for this result was contributed by Jochen Speck who also did the main work on the mathematical techniques used in the paper. Peter Sanders guided the work and helped to make the article understandable. This result is described in more detail in Section 5.2.5. How a malleable job optimally uses the given frequencies depending on its assigned amount of cores is discussed in Section 7.1.1.

- *Malleable Sorting* (joint work with Patrick Flick and Peter Sanders [45]) This work is an implementation of a sorting algorithm that fits into the definition of a malleable job. Especially the effect of providing the application with information about other running jobs is investigated. This information flow enables a much higher efficiency (at least in some cases) showing that informing the application about the system status can be beneficial. It is discussed in Section 4.6. Here Jochen Speck contributed the idea on how to build a malleable application and Peter Sanders contributed knowledge about sorting. Patrick Flick did the implementation as main part of his bachelor thesis under the guidance of Jochen Speck. During the implementation Patrick Flick (also with the guidance of Jochen Speck) developed solutions for sub-problems of the approach like an efficient lock-structure for the work queue and an improved splitting algorithm. The experiments for the article were done after the bachelor thesis was finished.
- *Constraint-Based Large Neighborhood Search for Machine Reassignment* (joint work with Felix Brandt and Markus Völker [18]) This work investigates scheduling in a case with very high complexity when computing the optimal solution is totally out of scope. We also take a deeper look into this result in Section 5.3. The article describes the approach of the program the authors developed for the ROADEF/EURO Challenge 2012 (Machine reassignment) where they reached a second place in the junior category (no group member had his Ph. D. yet). The idea to use a constraint programming approach was contributed by Felix Brandt who also did all of the programming directly related to constraint programming like the propagators and branchers. Markus Völker contributed the ideas for some of the used strategies and Jochen Speck the parallelization. The main part of the work was tuning the strategies and their combinations, experiments and analyzing results and inputs which was done in close collaboration of all authors.
- *Locality Aware DAG-Scheduling for LU-Decomposition* (joint work with Tobias Maier and Peter Sanders [100]) In this work an in-application scheduler is developed and experimentally studied which places and orders computations in a memory-efficient way which is also more energy-efficient. The

ideas and findings are described in Section 6.2.2. The main ideas for this work like common usage of L3-cache contents and the possiblity of improvements for the LU-decomposition were contributed by Jochen Speck who also provided the knowledge about DAG-scheduling. Peter Sanders contributed an idea how to distribute the scheduling and also provided some guidance to improve the quality and comprehensibility of the article. Tobias Maier did the implementation and experiments as main part of his master thesis under the guidance of Jochen Speck. The NUMA-awareness of the approach was developed by Tobias Maier and Jochen Speck together.

In the previous Section 3.4.2 about our methodology we pointed out that the experimental evaluation of operating system level schedulers and whole scheduling systems require big efforts and thus large projects. InvasIC was/is such a large project also dedicated to develop a new system design and with that new scheduling techniques. A more detailed description of the results of InvasIC from a scheduling perspective is given in Section 4.4.

4

Hierarchical and Distributed Scheduling

This chapter is dedicated to the question where scheduling decisions should be made and how they can (and why they should) be divided between different levels of the abstraction hierarchy (see Section 2.5 for an introduction into abstraction hierarchies) and different entities on the same level. Section 4.1 motivates why this is a relevant topic and why it might be helpful to take a look at the world outside of computer science where similar problems arise. The possibly helpful results and definitions from outside are presented in Section 4.2. The reasons and characteristics for distributed scheduling in common systems and the resulting problems are described in Section 4.3. The author of this work contributed to Invasive Computing (InvasIC) which takes a new approach towards scheduling and resource distribution. The general approach of InvasIC and its results are depicted in Section 4.4. Some possible future improvements of of the scheduling system are presented in Section 4.5. A proposal how to make an application malleable by an enhanced application interface for better communication between decision makers and increased flexibility in the resource demand is given in Section 4.6. Hierarchical in the chapter title has a double meaning in this case: on the one hand we look at the scheduling in abstraction hierarchies (hardware, operating system, application) and on the other hand one scheduling unit might be subordinate to another such that they form an organizational hierarchy.

4.1 Motivation

Let us take a look at the decisions that have to be made on a computer while it is performing its tasks. Twenty years ago most computers were simple: they had usually one processor with one core and fixed frequency. Hence only few decisions were needed: the hardware had to control the cache, the operating system had to schedule the application to run next and the application had to decide which task to perform next. These decisions were relatively easy and usually had only small effects on other decisions.

Today we have parallel machines where more decisions have to be made and the results are more important for other decisions (see also Section 2.2.2 and Section 2.4). Today the hardware can control the cache and the operating frequencies of the cores together with the operating system. The operating system has to decide which job to run on how many cores and on which cores, and additionally the operating system has to decide the NUMA node which fulfills memory requirements (the application might overrule some of these decisions). The application itself can interfere with some operating system decisions and also has some internal decisions to make like which thread works on which node of a DAG-structured task. These decisions interfere with each other: threads are scheduled onto cores by the operating system scheduler, but their work is assigned to them by the application-internal scheduler. Different cores of the same processor might work on different jobs but share some common resources like a common last level cache, memory interface or heat capacity. This might lead to interferences between different applications.

The problem is further complicated as no scheduler (decision maker within the application, operating system or hardware) has all relevant information. Much information is not accessible due to abstraction hierarchies (see Section 2.5). For example the operating system knows nothing about the application-internal task-DAG and hence does not know which thread works on the critical path. Another example is that usual applications do not take care of other applications and thus fail to adapt their degree of parallelism to the current resource demand in the system which can lead to unnecessary context switches and resulting cache context losses. A lot of the decision-relevant information is also generated at runtime like data-dependent running times of tasks or the processor temperature (dependent on ambient temperature).

Future systems probably have an even larger decision space. First of all more things can be steered. For example cache partitioning or the operating frequency of the memory controller. Furthermore the degree of parallelism will grow (see Section 2.2.3). Also some applications will become more complex with different parallel subtasks which themselves might be parallelized.

Hence we have many different decisions to make and no scheduler can access all information (see Figure 4.1). On top of this, these decisions have to be made quickly at reasonable cost. Thus we do not want to move too much information around in order to keep a centralized scheduler informed as the information movement might be costly and increases the latency of the decisions. Another problem with information flows between different hierarchy levels is that each flow complicates the interface. Large interfaces are complicated to develop and interface changes (new information flow, possible error) are expensive because of

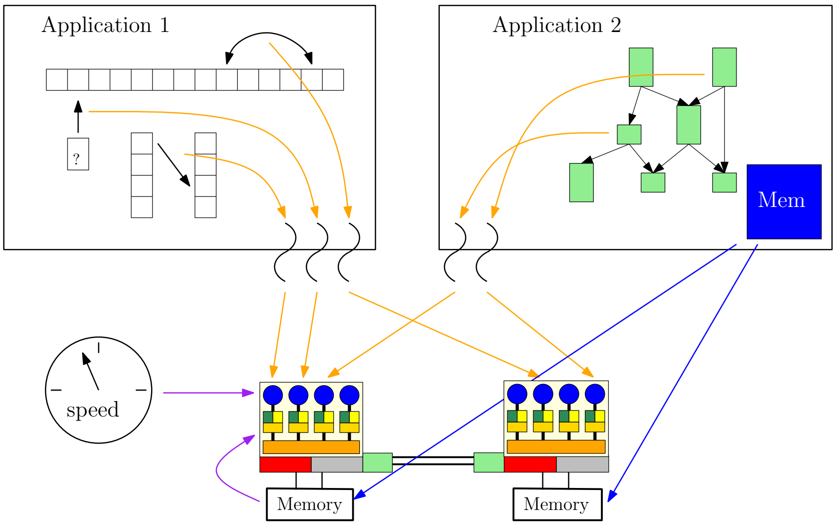


Figure 4.1. Example for different decisions while running two applications on a two socket multicore system: Job-internal: which thread does what; OS: which thread runs on which core; OS: which memory allocation is satisfied on which socket; Hardware: which core runs on which speed; Hardware: which piece of memory is loaded into the cache. All these decisions interfere with each other.

compatibility reasons, they also might hinder the freedom of development on both sides. Hence we want small and simple interfaces and thus only a small information flow through them at least in terms of different kinds of information. Also a central scheduler that works with all this information might also be slow and expensive in terms of computational effort. Thus there is a need to split up decisions between different schedulers and only communicate the things needed. On the other hand it is also possible that additional information passed between different parts of the system might be beneficial for efficient decisions. This might outweigh the additional effort consisting of the higher complexity of interfaces and the additional transmission and computation effort.

Hence it is clear that a computer scheduling system should have an efficient structure that uses upcoming information where it is generated and thus saves communication and preserves the more central scheduling units from overload. The information transmitted between the different decision makers should be near the amount which keeps the effort for the transmission and the facilitated decision improvement balanced.

There is some work in computer science about decision distribution in real-time systems and cluster and computing center scheduling (see also Section 3.2.3) which is discussed in Section 4.3.3. Also agent systems (see for example Kobbe et al. [85], also mentioned below in the description of InvasIC) and work stealing (see for example Blumofe and Leiserson [13]) are typical methods of scheduling decision distribution in computer science.

But computer science is not the first field where such decision organization problems arise. The probably oldest large organization which needed a distribution of decisions was the military. The military domain also provides a lot of examples for the need of decision distribution which are easy to understand. Another field with a need for decision distribution which is not as old but intensively researched is corporate organization. There is an abundance of books about management which usually include the delegation of decisions. We took a look into these areas and some definitions and ideas that might be helpful in computer science are presented in Section 4.2. The main difference between these fields and computer science is that it is usually assumed and probably the case in practice that the decisions are made by humans rather than computers. An important property of human decision makers is that the quantity of information they can absorb and process is limited. This somehow fits our goal stated above that the information amount and computational complexity of decisions should be kept small in order to facilitate fast decisions. Even as a processor core might be able to store and process structured information much faster than a human, the scheduling decisions it has to make are needed within a much smaller time frame. Hence as the decision maker becomes faster, also the decisions have to be made faster and the need for fast decisions remains. Of course decision tasks for human deciders have to be designed in such a way that motivation is kept up and selfishness and fraud are kept down. As computer programs do not need motivation and we only consider cooperative scheduling in this work we do not need to consider these problems.

The author of this work was part of the InvasIC project from 2010 to 2014 in which the distribution of decision making also played an important role. Of course the discussions within the InvasIC team influenced this work (especially this chapter) and the author of this work contributed to the discussions within InvasIC as well. We present our view on InvasIC and especially its scheduling structure in Section 4.4. A lot of discussions during the work on InvasIC contained the topic of interacting decisions of different components. Within these discussions the author of this work sometimes had a problem with using proper names for his more abstract remarks. Hence while writing this work, he took some time to look up similar problems in other fields where names already exist for these abstract problems. The results of this research are presented in Section 4.2. On the other hand these general problems do not only exist for InvasIC but are a general problem for all kinds of modern computing platforms. Hence the second purpose of this chapter is to point out reasons why these problems exist on many platforms and that it is worthwhile to think about them. The proposed approaches and solutions are meant as an initial step and not as a final solution.

4.2 Results in Other Areas

In this section we present some definitions and ideas from outside computer science that seem helpful for scheduling from our perspective. Not only ideas but also definitions are presented as it is often very helpful if a thing has a name that can be used in discussions or descriptions. Although there is an abundance of books about management and corporate organization most of them do not present their results on an abstract level which makes it difficult to transfer the results to computer science. An important exception is Frese [47]. Hence most of the ideas presented in this section are from this book. We give more detailed references directly with each definition or idea. We also add some computer science interpretation to each item in order to demonstrate its applicability in computer science.

Decision structure: field, action, goal (see Frese [47, page 39 ff]) According to Frese each decision consists of 3 components:

- Field: State of reality at the time the action we have to decide about takes place. The field is further divided into resources and environment, where the resources are the part of the field that is under the control of the decision maker and the environment is the part that is outside this control.
- Action: Usage of resources. Leads to a final state.
- Goal: The set of final states that are pursued by the decision maker combined with a preference structure among them.

This is similar to our definition of scheduling in Section 1.2 (except for the information part which is introduced later in the book): the field are the properties and constraints and the decision space is the set of actions. This similarity makes the definitions and ideas from the book of Frese look promising.

Interdependence (see Frese [47, page 58 ff]) We look at two decision makers A and B which are not in a hierarchical relationship with each other. If decisions of A change the decision field of B such that the optimality order of B's actions is changed, then we have an interdependence.

Hierarchical here means the organizational hierarchy and has nothing to do with the abstraction hierarchies in computer science. An example where an interdependence might occur in computer science is: If A is the operating system scheduler and B an application-internal scheduler which coordinates the work on a task-DAG, A decides which threads that belong to the application will run next. If B has a task on the critical path, it wants to assign this task to a thread of those which run next. Hence the best decision of B is dependent on the decision of A.

Another example is the usage of common resources. Let A and B each manage a different core of the same processor with a common last level cache. A makes the decision whether to run a cache consuming job or a job with small cache footprint on its core. B has to decide to run a job that profits from a large cache or a job that does not depend on the available cache size. There is an interdependence between the decisions of A and B because there is an influence of A's decisions on the optimality order of B's decisions through the common last level cache. Hence interdependence is important between abstraction hierarchy levels as well as within them.

Coordination (see Frese [47, page 69 ff]) Coordination means the orientation of the actions of single decision makers towards the common global goal. It consists of two parts: the definition of which unit is allowed to make which decisions and the definition of communication connections between the decision makers. These parts are not independent of each other. More coordination means less autonomy.

A typical example of coordination in computer science are cache coherence protocols. The common goal is to provide fast memory operations to all cores by the usage of local caches. This must be done in a way such that different cores modifying the same memory address produce the same result as without the caches. The different cache controllers are separate decision making units, but they communicate with each other in order to provide a coherent cache system. Another example for coordination is the work of Harris et al. [60] in which different parallel runtime systems are coordinated in order to improve efficiency.

Separation of resource potentials (see Frese [47, page 14]) A global resource is split up into smaller parts which then are managed independently by different units. This can lead to an inefficient usage of the resource. We will also use the term separation of potentials.

An example for the separation of resource potentials in practice is presented in the work of Lozi et al. [97]. As described in Section 3.2.1 the Linux-scheduler manages one run-queue for each core. These queues have to be balanced, otherwise there would be a severe problem with the separation of potentials. This balancing between cores uses a hierarchical structure which groups for example cores of one NUMA-node and also tries to balance the work between the different groups. Lozi et al. [97] have found a problem with this approach they call "The Group Imbalance Bug". When running one heavy (with regard to the Linux load balancing) thread and another program with a large number of light threads, the situation can occur that cores on the same NUMA-node as the core working on the heavy thread stay idle even if there are a lot of other waiting threads ready to run. The resources (cores) are separated by the scheduler structure and thus not used optimally.

Tradeoff: autonomy cost ↔ communication cost (see Frese [47, page 124 ff])

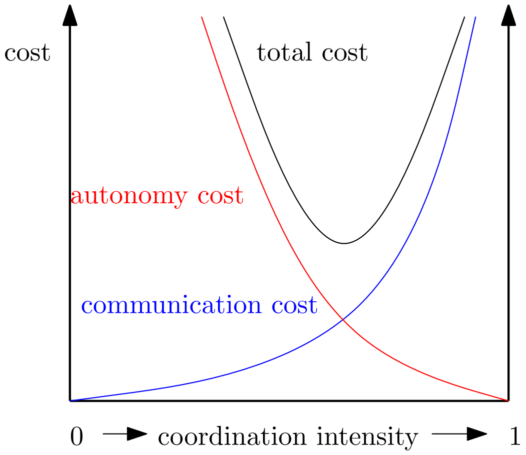


Figure 4.2. Connection between coordination intensity and autonomy and communication cost. Symbolic picture according to the one in Frese [47, page 126].

If a decision-making problem is divided between different units, communication is important to reach the common goal. When all interdependences are included in a coordination system, then this might be optimal for the common goal but leads to high workloads of the central decision units (and also to high communication cost). Hence we probably will not have coordination for all possible interdependences. Thus some decisions of local units might be suboptimal with respect to the theoretically possible optimal global goal. The difference between the theoretically possible optimal global goal and the real result with some not coordinated interdependences is the cost of the autonomy of the local decision makers. In order to reduce the autonomy cost, the local decision makers need to communicate more (or use coordination systems which also include communication). Communication also leads to cost. Hence we have a tradeoff between communication cost and autonomy cost. If additional communication is added, one usually starts with the interdependence which produces the highest autonomy cost in relation to the additional communication effort. Hence the positive effects of less autonomy cost are decreasing and the negative effects of more communication cost are increasing with the total amount of communication. Figure 4.2 shows a symbolic effect of this. Thus there is an optimal tradeoff of autonomy and communication which most likely will be between communication for every interdependence and no communication at all.

In computer science a large part of the communication cost comes from the effort of building appropriate interfaces. For example it might be helpful in practice if an application-internal scheduler can inform the operating system about the importance of different threads (see example of the application-internal DAG-scheduler above). The possible benefits in this case might be larger than the cost for the additional interface. On the other hand there might be small interdependences between the application-internal scheduler and the operating system scheduler that do not justify the additional effort of constructing an interface. For example an application-internal DAG-scheduler might know that there are slight differences in the power consumption of different workpackages and the threads of the application run on different sockets. The transmission of this knowledge might help the operating system scheduler to manage the heat capacity on the different sockets, but the possible improvement might be too small to justify the additional effort to implement an interface. Another implicit cost of communication might come from the fact that more information leads to more complex decision processes in order to use the additional information.

Mission-type tactics (Auftragstaktik) Armies were probably among the first institutions to realize that there is a benefit in making decisions where the most information is available and the latency is small enough that the situation has not changed much when the resulting orders arrive. Thus most decisions have to be made at a low level in military hierarchy. This also helps to protect the higher levels of command from overload. Especially the German military developed a way of leading which is called 'Auftragstaktik'. A military commander should give orders to his subordinates in a way such that the command includes goals and the means to be used to reach them but leaves the freedom how to do this to the subordinates. This reduces the communication between the commander and his subordinates as well as the latency of the decisions. The historic development of the 'Auftragstaktik' is described by Oetting [108]. Creveld [137, page 269 ff] analyzes the military information-processing and its implications to the command structure and where to make decisions. This also includes a short analysis of the 'Auftragstaktik'.

The typical example in computer science might be the distribution of the CPU-cores of the system among the different applications. Today an application exposes a number of threads to the operating system, and if there are more threads (maybe from different applications) than cores, the operating system decides which threads can run. Usually the operating system knows very little about a thread's work within an application. An application of the mission-type tactics would be to give each application a number of cores and let the application itself decide which of its own threads to run on these cores. A similar result can be reached if the application is informed by the OS about upcoming scheduling decisions. Both ways of setting the degree of parallelism and informing the application about it are closely related to our understanding of malleable jobs.

4.3 Decision Distribution in Computer Science

In this section we take a look at the decision-making structure of common computer systems. We start with the reasons why there must be a distribution of scheduling decisions in computer systems in Section 4.3.1. The characteristics of scheduling systems in computer science that differ from other decision-making systems are described in Section 4.3.2. We also take a look at the previous work of scheduling decision distribution in computer science in Section 4.3.3 and analyze its differences to the approach of this work. Some of the problems due to the current uncoordinated scheduling systems which are the basis for InvasIC and our own work are described in Section 4.3.4.

4.3.1 Reasons

"The most fundamental problem in software development is complexity. There is only one basic way of dealing with complexity: divide and conquer. A problem that can be separated into two sub-problems that can be handled separately is more than half solved by that separation." (Stroustrup [125, page 693]). The decomposition of large systems into simpler sub-systems is a typical method of software (and system) design that can be found in textbooks of software engineering, for example see Sommerville [123, page 242]. An important example is the splitting of computer systems into hierarchies from the hardware over the operating system up to the application which is described in more detail in Section 2.5. So the splitting of the software in a computer system into an operating system and different applications is the result of a globally accepted design principle which is to partition big systems into smaller sub-systems.

The goal is to have related parts of the system in the same partition of the system in order to reduce the relations between different partitions and thus the interface complexity. Here we have some kind of dilemma. The parts of the applications, the operating system and the hardware that are involved in scheduling decisions are closely related to the component (application, operating system, hardware unit) they are part of. The application-internal scheduler which controls the behavior of the application was developed by the application developers and uses code and data structures of the application to manage its internal algorithms and data processing. The operating system scheduler (or schedulers for different resources) uses the means of the operating system to control the resources which it divides between the applications. Also hardware-internal decision makers are closely related to their means by which they exercise their control. On the other hand the schedulers are related to each other as the combination of their decisions is the scheduling of the machine which should lead to an efficient result. Hence scheduling is a cross-sectional part of the system as the relations of the different schedulers with their components are so dense that the combination of all schedulers within one system component seems impossible. Thus computer systems have scheduling-related parts in many different system components.

In the old times when sequential programs ran on sequential machines the scheduling decision space *within* an application was very limited. Also the application did not need to adapt that much to the situation on the system as no other application could run in parallel. Hence such applications had no possibility to contribute much to the overall scheduling of the machine. Today with parallel applications on parallel machines applications can decide upon their degree of parallelism, which operation to perform, by which thread and the use of common resources and thus have an impact on the overall scheduling and its results. In the future the decision space and the impact of the application-internal schedulers is likely to grow even more. On the other hand the operating system scheduler is and will still be important as it controls the resources and their division between different applications, and systems will be shared by more than one application.

The degree of parallelism is also an example that decisions are also divided between development/compile time and runtime. The degree of parallelism of an application can be fixed within the implementation, or it can be left to one or more decisions during runtime. Compile time decisions are not part of scheduling and are not considered in this work. For reasons of portability between machines with different numbers of cores and an increased adaptivity to the load situation on the machine it is likely that such decisions will be moved from compile time to runtime. This increases the decision space and the importance of the application-internal scheduler even more.

Altogether scheduling decisions are spread over different components of modern computer systems, and different components play an important role. Thus scheduling is (and will be even more in the future) a problem with distributed decisions.

We already gave some examples why many of these decisions have interdependences (application-internal DAG-scheduler and operating system thread scheduling and the work on the critical path). Hence there is a need to look at possible ways of coordination.

The reasons for decision distribution that are common to most organizations also apply to computer scheduling: the limited capacity of central decision makers and the cost of communication lead to a more distributed decision organization. The number of overall decisions within a computer system grows thus increasing the decision space (see Section 4.1). Also the amount of information needed for these decisions is likely to increase. As the speed of a single core is not likely to increase much in the future, the problem with the limits of central decision makers will become even more visible. As the number of cores increases, the interconnection networks and their latencies will also increase which increases the cost and latency of communication.

4.3.2 Characteristics

The distributed decision making for scheduling in computer science has some characteristics that are different from other distributed decision making problems. As already explained in Section 4.1 the decisions in computer scheduling are not made by humans which makes the problem somewhat simpler as we can ignore selfishness, motivation and possibilities for fraud. But there are also some differences that have nothing to do with the decision makers themselves.

Unit splits for reasons independent of decision making Most organizations split their decisions into smaller units mainly because of reasons that lie in the decision making itself: units are kept small to make it possible for one manager/officer to know enough about their status for useful decisions; sales organizations are usually split into units for every country to reduce latency and to adapt to local habits; companies and the military are split into specialized branches in order to allow the branch manager/commander to develop specialized knowledge about his branch to enable better decisions. In computer scheduling decision making is also distributed because of reasons *not* directly related to the decision making itself. As described in the previous section decisions are split up because of engineering reasons. The most important example for such a split is the division between applications and operating systems.

Stable and small interfaces There are good reasons to have a small interface between applications and the operating system with little (or no) changes for different versions of the operating system: the development of applications and operating systems is independent in large parts which makes the work easier for their respective developers, and applications are portable between different operating systems (at least within the same operating system family). The effort for a big change in the application interface is very high as all applications and operating systems have to be adapted and also the knowledge of the developers about the interfaces is lost. This might lead to a path dependence problem with suboptimal results as described in Section 2.3. Of course operating systems can offer different interfaces for different applications. The interfaces in computer science are usually developed to provide some functionality to the other side and usually not to enable coordination for distributed scheduling decisions. Hence the distributed scheduling problem has the additional difficulty of interfaces that are developed for other reasons and that are expensive to change.

Speed and number of decisions On most machines there is a decision for each core which thread to run next every couple of milliseconds (see Section 3.2.1). One reason is to divide the work of the core fairly between different applications and to enable a small response time to user input for each application. Hence the number of decisions is huge, and it is impossible to spend too much time on them as this would lead to a machine spending much of its work *deciding what to work on* instead of actually *doing work*.

Decisions about resources and work contents rooted in different units In most organizations the top level decider decides (at least in principle) what to do and also decides about the division of resources between different goals. For example a military commander might decide which targets to attack and how many troops take part in each attack. In computer scheduling the applications decide about the work contents, for example which algorithm to use or which data to store, but the division of resources is usually done by the operating system. The operating system (or some kind of middleware for clusters) is the only part of the system that knows all applications and all resource demands. It also controls the resources. Hence making the decision about resource distribution in the operating system makes sense, but it needs some information about a global objective.

Decision distribution between compile time and runtime An important part of the scheduling problem is the distribution of decisions between compile time and runtime. For example a parameter to adapt the displayed image quality to the available computing power has to be implemented at compile time to enable its usage at runtime. The same is true for the degree of parallelism that can be fixed at compile time, or the program is implemented in a way which leaves the decision to a runtime decision maker. Of course if all scheduling decisions of a program are made by the developer at compile/implementation time, there is no decision space left for an application-internal scheduler.

4.3.3 Previous Work

There are several other areas in computer science that are concerned with hierarchical scheduling and similar things.

In real-time scheduling the term hierarchical scheduling is used for methods that distribute the available core time between different sets of applications. Each set then does a sub-distribution between its components. An example for this is the work of Lipari and Bini [95]. This approach differs from our problem as the problem is much more specialized, and also flexibility of jobs is no issue.

Hierarchical scheduling also occurs in the area of computing center scheduling (an already mentioned example is the work of Hamscher et al. [59]). The computing center scheduling usually works on one level of the abstraction hierarchy and also is different in many other respects.

An article which argues for a hierarchical control of the system through a tree of controllers is the publication of Feitelson and Rudolph [42]. This work concentrates on a possible additional hardware system for load balancing and control and is restricted to decisions typically done on the OS level. It additionally looks on fault tolerance and fairness but does not look into the coordination with application-internal deciders.

The work of Peter et al. [110] discusses the relation of interfaces between application and operating system and the system scheduler. This article first gives arguments that future parallel workloads will be more dynamic and will run simultaneously on the same machine. It then discusses the runtime decisions needed for this and that the application $\leftrightarrow$ OS interface has to be enhanced. The actual decision making processes, their distribution and hierarchies are not discussed.

4.3.4 Problems

The split between operating system and applications and the small application interfaces which are designed for the provision of functionality but not for coordination lead to some problems:

- Operating system decisions without knowledge about the application-internal status: Examples for this problem are described during this chapter. An application-internal scheduler distributes work among the different threads of the application, some packets might be bigger or more important than others. With optimal coordination the operating system would give the threads working on these packets more time on the cores or even a higher operating frequency (on DVFS systems). When the operating system does not know about these facts, it can only treat all threads equally.
- Inefficient resource competition between different applications: The typical example for this is the distribution of the available cores between two applications. Assume two applications which together have more runnable threads than the machine has cores are running on the machine. First we get unnecessary context switches if the operating system scheduler tries to run all threads which might also lead to cache context losses. A second problem are possible additional waiting times as one thread waits for the results of another thread that is currently blocked because no core is available. Also the usage

of common resources like last level caches or memory interfaces might be suboptimal.

- Application-internal decisions without knowledge about the system load situation: For example in a system with high load it might be beneficial for an application that displays images to the user to use a lower quality in order to reduce waiting times. Hence it can be beneficial for an application to know about the current load in order to decide the optimal tradeoff between quality and waiting time.

An example of the negative effects due to a lack of coordination is given in the Ph. D. thesis of Johannes Singler [120, page 40]. Singler developed a new sorting algorithm and tested its behavior when one core was permanently blocked by another program. Running the sorting algorithm with seven threads on an eight core machine was more than twice as fast as with eight threads. He also compared his algorithm with another sorting algorithm (with better internal load balancing) which was slower in general but had a much smaller loss when running with eight threads and one blocked core. But there was still a loss in performance when eight threads were used instead of seven. This is a typical example where coordination between the operating system scheduler and the application-internal scheduler can improve the efficiency of the system. This was also the basic observation that led to the idea of creating a malleable sorting algorithm (see Section 4.6).

Today such problems are often solved by running only one program at the same time on the same machine or other static resource partitioning schemes that reduce (or even eliminate) the interdependences between different applications and reduce the relevant decision space of the operating system scheduler. This works against the trend of consolidation which means that more applications run in parallel on one powerful machine instead of several smaller ones in order to reduce energy consumption and to prevent a separation of potentials. Also fluctuating resource demands and loads cannot be efficiently met by this measure (see Section 2.4). Other measures are compile time decisions (in the application) like thread pinning where the application programmer reduces the decision space of the operating system scheduler. The downside of this approach is that the system efficiency can be very bad if something happens at runtime that was not expected by the application programmer. The operating system also tries to estimate the application actions in order to make decisions beneficial for efficiency. For example the Linux scheduler tries to conserve cache- and NUMA-locality if possible by doing load balancing in a hierarchy (Zhuravlev et al. [153]). A better coordination can replace estimations by knowledge and thus improve the situation. An example is the work of Harris et al. [60] in which different parallel runtime systems are coordinated in order to improve efficiency.

Altogether we have seen that there is a problem because of a lack of coordination across the application ↔ operating system interface. Hence more coordination and an improved interface will lead to better scheduling decisions. Of course these decisions are only useful if they can be made during runtime. Thus decision movement from compile time to runtime within the applications is an important precondition for better coordination. In the future the increased decision space and degree of parallelization will also enforce a decision distribution because of the overload of single decision makers.

4.4 Invasive Computing Results

This section is devoted to the Invasive Computing (InvasIC) project in which the author of this work took part for most of the time of his Ph. D. studies (2010-2014). The outline of InvasIC is described in Section 2.7. The author of this work was part of the scheduling sub-project A3 as the only full-time employee of this sub-project (the list of InvasIC sub-projects can be found in the annual reports [132] or [133]). In this section we only look at the scheduling-related parts of InvasIC. We present the main ideas and development directions of InvasIC in Section 4.4.1. We also take a look at the important question which scheduling decision makers are planned in InvasIC. The main basis for the description of InvasIC is an overview article by some of the project leaders (Teich et al. [131]), the annual reports 2011 [132] and 2012 [133] and the memories of the author of this work who was a participant of InvasIC and worked on the scheduling of the system and thus took part in a lot of internal discussions and other information exchanges. Also other InvasIC-related publications are considered.

In Section 4.4.2 we then take a look at the developments within InvasIC. The section is finished with an analysis of the results of InvasIC. As the development process within InvasIC was never documented in total (it seems unlikely that this would even be possible for such a large project), the contents of Section 4.4.2 are based on the memories of the author of this work.

4.4.1 Main Ideas and Approaches

InvasIC aims to develop a complete new system, including new hardware, operating system, compiler and a new way of writing applications. The hardware is planned as a large chip consisting of different kinds of tiles with different computation capabilities or IO and off chip memory connections. The different tiles are connected through a network on chip (NoC). The typical picture used to illustrate the InvasIC hardware is shown in Figure 4.3. The computing tiles can consist of (several) normal cores (CPU), reconfigurable cores (iCore) or combinations and some tile local memory. Another option for a tile are tightly-coupled processor arrays (TCPA) for special computations.

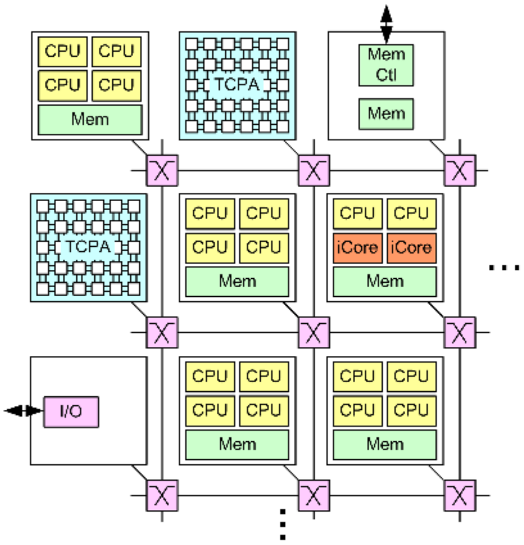


Figure 4.3. The planned architecture of the InvasIC system. The picture is copied from the InvasIC overview paper [131].

The system hierarchy (hardware, operating system, application) follows the common approach similar to Section 2.5 with one important exception: the operating system is not planned with a central scheduler but relies on an agent system for the distribution of resources [63]. The applications communicate with and through these agents and provide a lot of performance-relevant information and detailed requirements for additional resources.

Applications are not only required to provide more information interfaces, it is also required that they are able to adapt to the available resources. This is even the central idea of InvasIC called Invasive Programming (definition from [131]):

> **Definition:** Invasive Programming denotes the capability of a program running on a parallel computer to request and temporarily claim processor, communication and memory resources in the neighborhood of its actual computing environment, to then execute in parallel the given program using these claimed resources, and to be capable to subsequently free these resources again.

Hence InvasIC requires and facilitates applications that are able to fit their resource usage to the given resources at runtime. The goals of InvasIC are to provide a way to deal with systems consisting of thousands of cores, fault tolerance, higher resource utilization and (hopefully) performance gains (annual reports 2011 [132] and 2012 [133], both page 8).

Another important part of InvasIC is the distribution of scheduling decisions. As the InvasIC ideas target systems with hundreds or more cores, a central scheduling decision maker is not appropriate (annual reports 2011 [132] and 2012 [133], both page 8) as it would become overloaded and too slow. Thus decisions must be distributed especially by self-organizing applications. Of course this comes along with a higher overhead and a bigger programming effort, but these additional costs are acceptable compared to the expected benefits. This is expressed in the following quotation "there is a price to pay in order to exploit the benefits of invasive computing" on the last page of the overview article [131].

In order to describe a scheduling system, it is important to identify the decision making units. For InvasIC we can identify four decision makers:

- Applications are adaptive in InvasIC. Thus they are of course capable of making decisions how to use the offered resources. They also decide which resources to request.
- The agent system/operating system gets the requests from the applications. It also has to decide how to distribute the given resources between the applications. For more information about the architecture of the operating system and the agent system see the annual reports 2011 [132, page 49 ff] and 2012 [133, page 56 ff].
- On each tile of the InvasIC architecture there exists a Dynamic Many-Core i-let Controller (CiC). The CiC does scheduling in hardware in order to speed up the decisions. It is also responsible for gathering system status data. Also more detailed descriptions of the CiC and its responsibilities can be found in the annual reports 2011 [132, page 35 ff] and 2012 [133, page 38 ff].
- The network on chip controllers (iNoC) can make decisions about network routing in order to optimize the connections and provide support for the embedding of application-specific communication topologies. See also the annual reports 2011 [132, page 45 ff] and 2012 [133, page 51 ff].

The interface between the applications and the agent system enables the application to constrain the set of requested resources. A large number of constraints were developed within InvasIC and are depicted in Figure 4.4. Apart from different constraints to specify the requested hardware (e. g. FpuAvailable), the hardware status (e. g. MaximumTemperature) or orderings of resource collections according to these specifications we also have an important information transmission from the application to the operating system/agent system: hints. These hints provide the agent system/operating system with the information how much the application profits from additional resources. A typical example of these hints are speedup curves. Hence hints provide the information basis for decisions about resources for which different applications compete. Hints are especially important if the goal of scheduling is to increase the overall efficiency or to reach performance gains.

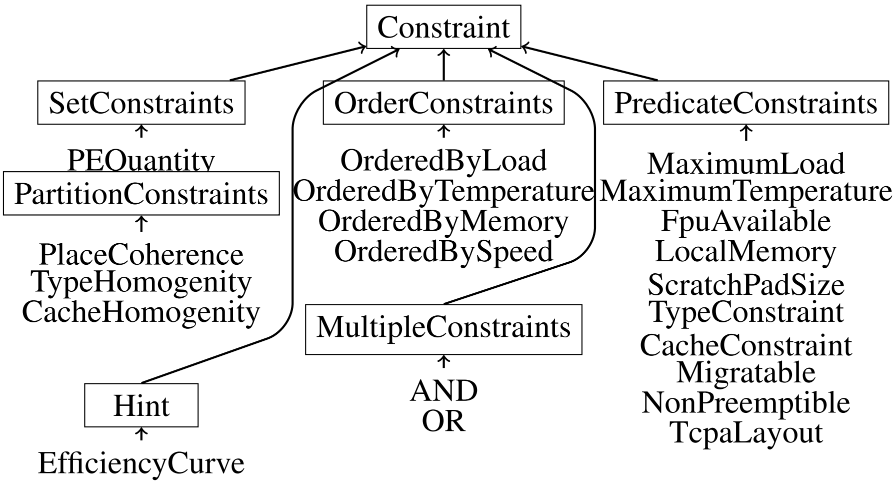


Figure 4.4. The structure of possible constraints for new resources that the application can pass to the agent system. The picture is copied from the InvasIC annual report 2011 [132, page 19].

4.4.2 Development

Goos [53, page 73] discusses two different approaches for the development of computing systems: *top down design* and *bottom up design*.

Top down design is the approach that one starts with a global goal (e. g. efficient computation on thousands of cores) and then divides the global goal into subgoals. The recursive division of goals leads to small tasks that then can be solved and work as components for a solution of the global goal. The risk of this approach is that one of the small tasks might turn out to be unsolvable.

In **bottom up design** one starts with already known or easy to get solutions for subproblems. After that these small solutions are combined until a solution for the global goal is found.

The main difference of these design approaches is the chronological order between the work within the subproblems and their combination. The top down approach puts the work on the combination before the work on the subproblems, the bottom up approach does it the other way round.

The InvasIC development process included both approaches (as all real development processes do), but the bottom up approach was dominating. Some central ideas were fixed at the beginning, and there was a division into sub-projects, but the main work of the sub-projects was done within them. Also the initial focus of InvasIC was to provide a working system and thus putting efficiency and performance concerns into the second place. An important reason for that was the comprehensible desire to be able to present something working in the first evaluation after four years. Five of the twelve accepted sub-projects (see Section 2.7) were hardware development projects and the other seven were divided into applications, language, compiler, operating system, simulation and scheduling thus giving the hardware development the most importance. Thus scheduling in software which has to coordinate different subsystems (sub-solutions) for a non-functional objective was a rather marginal topic of the development. Unfortunately the simulators developed within InvasIC were not built for the simulation of non-functional parameters (e. g. memory latencies, task switching overhead, NoC latencies) and were rather dedicated to test the functional correctness of some system components. In a bottom up development with no experimental or simulation results for non-functional parameters it is also difficult to build a meaningful model to compare different scheduling strategies. Hence there was no possibility for testing a combined scheduling process over all system layers within InvasIC and hence no possibility for the application of the algorithm engineering methodology (see Section 2.6.1) on scheduling as described in Section 3.4.2. On the other hand it was possible to observe some upcoming scheduling problems of the InvasIC approach. As the author of this work was part of the InvasIC development team, it was possible to discuss with other team members if these problems are due to fundamental reasons or if they can be solved or moderated by small changes of interfaces or other components. We will now look into some of these scheduling problems encountered in InvasIC and discuss how they were solved or if they are still open (we refer to the project status at the beginning of 2014 when the author of this work left the project).

Application interface In computer scheduling and many other kinds of decision making organizations there is a tradeoff between autonomy cost and communication cost (see Section 4.2). To decide anything in this tradeoff one must be able to estimate the cost. Without models, simulation or experiments the autonomy cost or communication cost could not be estimated in InvasIC. Together with the rather functional than non-functional approach there was a risk of building an application interface that leads to heavily constrained resource requests (see Figure 4.4). These too constrained requests bring the risk of fragmentation as exact fulfillment becomes impossible without leaving many components idle. Also there is a risk that the possibility of detailed constraints will lead to decisions made within the application that are better made by the operating system because decisions within the applications constitute the risk of separation of resource potentials. In order to increase the decision space of the operating system/agent system and to provide the necessary information for good decisions, performance hints were introduced. The performance hints provide information that can be helpful to make good decisions regarding the overall goal. Especially if these hints are combined with modestly restricted resource requests, they leave the operating system with a greater freedom to divide the resources among the applications. Performance hints like speedup curves are commonly assumed in the models in scheduling theory. With the help of some other colleagues the author of this work could convince the InvasIC developers to include hints (especially speedup curves) into the application interface. When the author of this work left the project, the final interface design was still open.

Complexity of scheduling The computational complexity of finding an optimal or at least good schedule is an important part of scheduling (see Section 5.1). It is not clear yet if the rather big decision space (scheduling decisions can be made about memory, communications, cores) and the large set of constraints lead to a complicated scheduling problem in practice. Due to the lack of possibilities for experiments this problem could not be investigated.

Vertical coordination As described in Section 4.4.1 there are many different decision makers within the InvasIC system. Interdependences between the decisions of the operating system/agent system and the CiCs and iNoCs certainly exist as the CiCs should make decisions on behalf of the operating system and intra-application communication between tiles through the network on chip are obviously performance-relevant. There were some discussions about coordination and information exchange between these decision makers, but no solution could be found during the author's participation in InvasIC.

Horizontal coordination A central problem in scheduling computer systems is to distribute the available resources between different applications. Doing this by an agent system that makes all decisions with only local knowledge might lead to a separation of potentials and thus to a suboptimal usage of resources. On the other hand scalability of such an agent system is easy. A centralized scheduler can perform a global coordination but might become overloaded. Kobbe et al. [85] (all authors were working on InvasIC) compared a simple centralized heuristic (running on one core) for the distribution of cores among jobs with a decentralized heuristic developed by them. The comparison was conducted in a simulation environment (not especially InvasIC-related) by comparing the results of both approaches on some generated scheduling instances. The comparison showed that the decentralized approach saves a lot of communication and computation effort while the reached average speedup among the jobs is 84% of the centralized approach (average over all instances). This work shows that decentralized approaches can save a lot of communication effort compared to centralized approaches which might become a bottle neck. Unfortunately neither the centralized nor the decentralized algorithm comes along with a guarantee for the solution quality.

A solution approach that does the scheduling decentralized but aggregates a central value (e. g. the current global demand of a resource) to prevent the separation of potentials was developed by the author of this work together with Peter Sanders [116]. Hence our approach does not become a bottle neck but also produces optimal scheduling results. This work (which is the parallelization of our main algorithm) is described in more detail in Section 5.2.4.

Altogether InvasIC worked on the main problems of scheduling within modern machines (as described in Section 4.3.4): within the applications performance-relevant decisions were moved from compile time to runtime; the application interface was enlarged in order to prevent information loss and to enable coordination; decision making was decentralized to prevent the overload of central decision makers. But there were also some problems that remained unsolved (at the time the author of this work left the project): the coordination of different decision makers; the possibility of a separation of potentials due to local decisions; the possibility of fragmentation due to too tight constraints; the possibility of too complex scheduling problems.

This work is clearly influenced by InvasIC, but unfortunately the author's hope to compare different scheduling systems in experiments or simulations as described in Section 3.4.2 could not be fulfilled as the basis for such experiments was not given. On the other hand InvasIC provided a lot of open scheduling problems and insights to the problems of different parts of a computer system. Also the author was able to support and influence the InvasIC development process with a scheduling perspective.

4.5 Ideas for Improvement

In this section we describe how the scheduling decision makers should be organized in our view in order to get good scheduling results without too much overhead. We also give an overview where scheduling organization ideas influenced our work.

Move decisions from compile time to runtime Changing the applications such that more things can be decided during runtime is very important in order to have a big enough decision space at runtime to be able to react to things that can occur during runtime (e. g. load situation, machine temperature or special inputs). We see several possibilities for application decisions that can be made during runtime:

- Degree of parallelism.
- Usage of other resources especially cache and local memories.
- Dynamic work distribution among the threads that belong to the application (e. g. task DAGs within the application).
- The used algorithm. Being able to use different algorithms is expensive in terms of development effort as all of these algorithms have to be implemented.
- The solution quality (e. g. image resolutions or objective function values of optimizations).

Moving decisions from compile time to runtime can be beneficial for scheduling, but it also comes along with more implementation effort. Hence it is especially useful when the runtime costs (hardware and energy usage) are much higher than its development costs. The movement of decisions from compile to runtime was also a core feature of InvasIC and thus was intensively discussed there.

Better decision coordination An important problem of today's computing systems is the lack of coordination between the applications and the rest of the system. We give some examples in Section 4.3.4, and also this topic was discussed within InvasIC. For future systems with more distribution and self-organization also the coordination within the operating system and across the hardware interface becomes more important. On the other hand the coordination should not grow too much as it comes along with communication overheads and possible latencies. Hence one has to find a good tradeoff between autonomy cost and communication cost (see Section 4.2). But the direction is clear that the application interface has to be enhanced to allow more coordination although this comes along with large development costs.

Move decisions to the information Information plays a key role in making decisions. Of course all (with reasonable cost) available relevant information should be used in order to make good decisions. If possible, the decision maker should be moved to a place where most information can be made available cheaply and with low latency. For example a military commander of a company does not work in an office far away from the battle but is usually located close to his soldiers. Hence a decision maker should be placed where the supply with relevant information can be enabled in the cheapest way. A development that moves decisions from the operating system to the application can lead to effects contradictory to this approach. For example core pinning moves the decision about which core is used by which thread from the operating system to the application. But the application probably does not know about the current load and which other applications use the core to which it has pinned its thread. If the application-internal scheduler should make such decisions, it has to know something about the system status, and thus the system status is replicated among all application-internal schedulers which leads to overhead and communication cost. Thus global decisions should be made within a system component with global span which is usually the operating system. On the other hand scheduling decisions mainly influenced by the application-internal status should be made within the application. The mission-type tactics (Auftragstaktik) pattern can be helpful to structure such decisions and thus lead to the encapsulation of local decisions with all benefits like interchangeability of sub-solutions and low latency and communication cost. For example the operating system usually knows far less about the work of different threads of an application than the application itself. Hence the decision which work (assigned to a thread) should be stopped and which should be continued is better made within the application. On the other hand the operating system knows more about the global demand for cores and thus can make better decisions about the number of running threads for each application.

Distributed decisions without separation of potentials As already noted in previous sections there is a problem with central decision makers becoming overloaded in large systems. Hence decisions must be divided even within the same component (in most cases the operating system). On the other hand dividing decisions between different autonomous units can lead to a separation of potentials and autonomy costs. A distributed decision system with a small amount of coordination might be the best compromise. For example the decision about the usage of a global resource can be made in a distributed way, but something like a price for each unit of the resource that reflects the (global) demand can be added. Thus the price of the resource works as a lightweight coordination system. The aggregation of supply and demand can be done via broadcast and reduction operations which leads to a fast and efficient realization. See for example the parallelization of our main algorithm in Section 5.2.4.

Tradeoffs An important task when designing a scheduling system is finding good tradeoffs between contradictory approaches. We have already seen in Section 4.2 that there is a tradeoff between autonomy cost and communication cost for all kinds of organizations. In computer science an increased latency can also be considered as communication cost. Another well-known tradeoff exists between the solution quality of the scheduling and its computational effort. There are many works on approximation in scheduling (see Section 3.1.3). Finding good tradeoffs for a specific system is a complicated task. Thus such tradeoffs are usually investigated by performing experiments on real systems or simulations on system models which are detailed enough.

Examples in this work The idea of distributed decisions without the separation of potentials is part of the article *Efficient Parallel Scheduling of Malleable Tasks* (joint work with Peter Sanders, [116]). The parallelization of our scheduler is discussed in Section 5.2.4 in greater detail. An example of a job with a flexible degree of parallelism (can be changed from outside) is presented in Section 4.6 (the next section).

4.6 Malleable Sorting

Malleable jobs occur in scheduling theory (see Section 3.1.1) and are a good example for programs in which decisions are moved from compile time to runtime (the degree of parallelism). Unfortunately such jobs seem to be without good examples in practice. Hence the development of such a job is interesting. However, in order to prove that the additional malleability does not cost too much efficiency, one has to compare the result with existing implementations for the same problem. Sorting algorithms are relatively easy to compare in their performance. Thus we implemented a malleable sorting algorithm and compared it with state of the art parallel sorting algorithms in a joint work with Patrick Flick and Peter Sanders (*Malleable Sorting* [45]). The competitors were the multiway merge sort from the multicore standard template library (MCSTL) as developed by Singler et al. [121] and the sorting algorithm from Intel's TBB [112, page 78] (called TBB in the remainder of this section). This section is based on the article *Malleable Sorting* [45] and the measurements and implementations done for the article (for the individual contributions to the articles used in this work see Section 3.4.4). The idea of creating a sorting algorithm that can adapt its degree of parallelism to the system load is based on an observation that was described by Johannes Singler in his Ph. D. thesis [120, page 40]. Singler developed the multiway merge sort and tested its behavior when one core was permanently blocked by another program. Running the multiway merge sort with seven threads on an eight core machine with one blocked core was more than twice as fast as with eight threads. Hence it seems obvious that adaption can lead to big efficiency gains.

We call our algorithm malleable merge sort (MALMS). It is based on the multiway merge sort from the MCSTL (the abbreviation STLMS for standard template library merge sort is used in the remainder of this section). Hence we start with an explanation of STLMS and our changes in order to build MALMS. Then we look at the background and the details of the malleable interface that was developed by us. After that we explain the experiments and present some results of the comparison of the different sorting algorithms. We finish the section by fitting the malleable sorting results into the broader perspective of this chapter.

Merge sort basics We first briefly explain the multiway merge sort (STLMS).

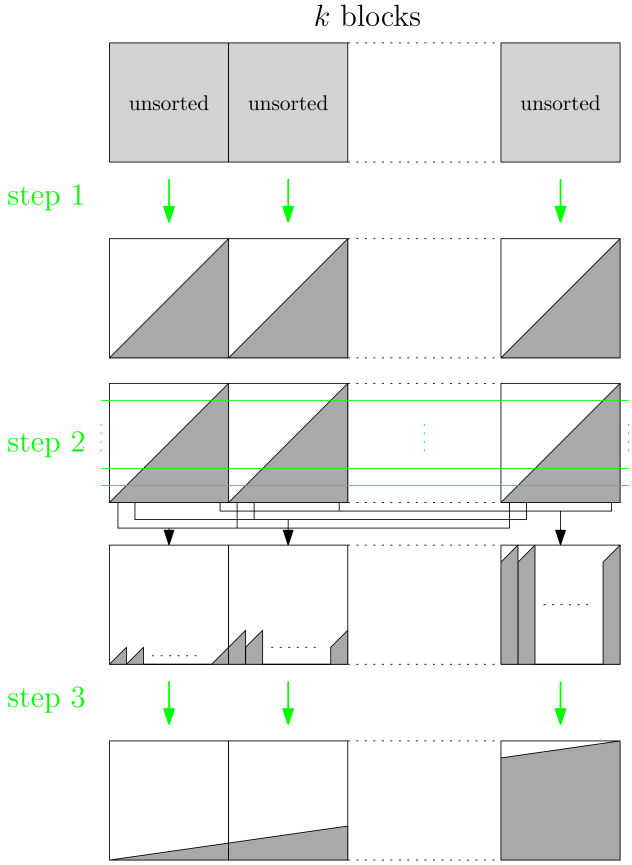


Figure 4.5. The three steps of multiway merge sort. Johannes Singler uses a similar picture in his Ph. D. thesis [120, page 11] to describe the multiway merge sort.

The multiway merge sort consists of three steps which can all be split up into k (respective $k-1$) workpackages:

1. Local sort. The elements to sort are split into k equal-sized packages. Each of the packages is sorted sequentially.
2. Split. $k-1$ splitters are computed which split each sorted sequence into k parts.

3. Merge. The k sub-sequences between two adjacent splitters are merged into one sequence.

Figure 4.5 provides a graphical overview over the three steps. The reordering of sub-sequences between step 2 and step 3 does not cost actual effort as we are on a machine with common memory, and thus all cores can access each sub-sequence with the same speed. The k (respective $k-1$) workpackages in each step are independent of each other and can thus be done in parallel. Workpackages of different steps are not done in parallel to each other.

Let us now look into the three steps in more detail. We use n as the number of elements to sort, k as the number of blocks (workpackages in step 1 and step 3, $k-1$ workpackages in step 2) and p as the number of used cores. We also assume that k divides n and p divides k.

In step 1 a workpackage consists of sorting n/k elements by the usage of a sequential sorting algorithm. Hence the running time of one workpackage is in $\Theta(\frac{n}{k}\log\frac{n}{k})$ which leads to a running time of $\Theta(\frac{n}{p}\log\frac{n}{k})$ for the whole step.

A workpackage of step 2 is the computation of a splitter for all k sorted sequences such that an amount of $r \cdot \frac{n}{k}$ elements is below the splitter (for $r \in \{1,\ldots,k-1\}$). We developed a new splitting algorithm with a complexity of $\Theta(k\log^2\frac{n}{k})$ for the sequential finding of one splitter. The running time for step 2 is thus in $\Theta(\frac{k^2}{p}\log^2\frac{n}{k})$.

Each workpackage in step 3 merges all k sub-sequences between two adjacent splitters. The complexity of this merge is in $\Theta(\frac{n}{k}\log k)$, and thus the complexity of the whole step is in $\Theta(\frac{n}{p}\log k)$.

For STLMS k equals the number of used cores. Hence each core first does one sequential sort, then computes one splitter and then merges k sequences. The malleable merge sort (MALMS) does the same things, but k is no longer set to the number of cores but is an optimization parameter instead (and usually much larger than the number of used cores). At the beginning of each of the three steps a single thread builds the k ($k-1$ for splitting) workpackages and inserts them into the global queue. Each active worker thread takes workpackages from the queue until it is empty and then waits until all workers have finished. When all workers have finished the next step starts. For more details about the implementation of the workpackages of the different steps see our article [45]. The code of MALMS and some test-code can be found under https://github.com/patflick/malms.

Malleability One of the first questions that arises when talking about experiments with malleable jobs is: how can this be done without an available system for malleable jobs? A malleable job can adjust its degree of parallelism according to external orders. Thus there has to be an external system that gives these orders. In the plans of InvasIC and maybe in other future systems as well these orders come from the operating system. This has the additional benefits, that the operating system can also enforce the changes and that it can use its knowledge about the global demand for parallelism for its decisions. As we do/did not have such an operating system available for our tests we chose a different way to test the malleability of MALMS and its possible benefits. Another job called Loadtask sends the orders, to change the degree of parallelism, to MALMS. As Loadtask cannot enforce these orders (especially not for the other sorting algorithms we compare MALMS to), it has to do something different to produce some kind of penalty for the sorting algorithms which use a core which should have been abandoned. After it has sent an order to free a core to MALMS, Loadtask starts a computation on the respective core. Also Loadtask stops its computation on a core when it tells MALMS to (re-)use this core. Hence running on a core that should have been abandoned leads to the penalty that the resources of this core must be shared with the computation done by Loadtask. This sharing is controlled by the operating system. Hence for the 'standard' sorting algorithms like STLMS and TBB the operating system is solely responsible to share the resources between them and Loadtask. MALMS retreats from resources (cores) used by Loadtask in order to supersede or simplify the decisions of the operating system.

As we run the experiments on Linux, we use Linux signals to transfer the orders between Loadtask and MALMS. One signal is used to decrease the degree of parallelism which contains the core number of the core to abandon as additional value. Another signal to increase the degree of parallelism works in the same manner. When we compare other sorting algorithms to MALMS, Loadtask does the same things regarding its computations but does not send signals as the other sorting algorithms have no interface to use them. The resulting system architecture then looks like Figure 4.6.

The adaption of MALMS to the signals from Loadtask is organized by an application-internal scheduler. For each core available on the whole system we have a thread pinned to that core which works on the central queue when it is active (sorting-threads). All sorting-threads work in the following manner: first the thread checks if it is blocked by reading its status from an array managed by the application-internal scheduler (scheduling-array). If it is blocked, then it goes to sleep, otherwise it fetches a workpackage and starts working on it. After finishing a workpackage, the thread restarts by checking if it is blocked. If the application-internal scheduler of MALMS gets the signal to use a currently not used core c, it wakes up the sorting-thread assigned to c. This thread immediately takes a workpackage from the queue and starts working on it. If the application-internal scheduler of MALMS is ordered to release a currently used core c, it writes a block signal into the scheduling-array at the position assigned to the sorting-thread pinned to c. This thread then goes to sleep after it has finished its current workpackage. If the workpackages are small enough, MALMS can

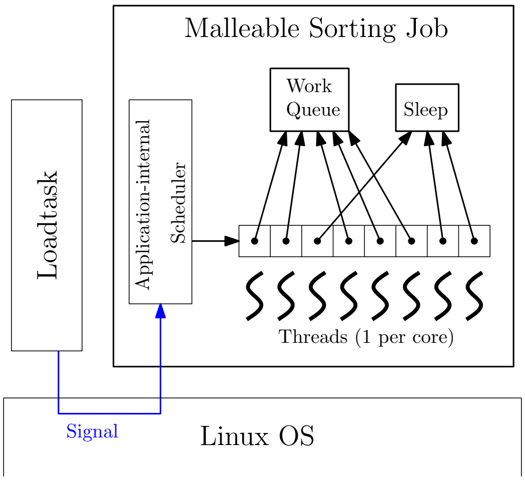


Figure 4.6. The structure and system integration of our malleable sorting application.

adapt to the orders of Loadtask with a fine granularity. The application-internal scheduler is built as a thread that does nothing else except waiting for signals of Loadtask, waking up threads and updating the scheduling-array.

Experiments and results We experimentally evaluated the approach on a machine with two quad-core Intel Xeon 5345 (Clovertown, 2.33 GHz) which is a unified memory machine as both processors share a common memory controller. Thus we do not have NUMA effects in our experiments. The machine is running a Linux Kernel version 2.6.32-45-generic x86_64 (different kernel versions seem to lead to slightly different results) and a GCC version 4.4.3. The TBB version is 4.1 Update 1. The Linux scheduler of the system has the standard parameters of the Ubuntu 10.04. installation.

First the different sorting algorithms are compared on an otherwise empty machine. We also include the running times of the sequential GCC std::sort (STD-SORT). Table 4.1 contains the running times when sorting different numbers of uniformly distributed 32-bit integers on an empty machine. In order to have a better comparison between different input sizes, the running time is given as t/n where t is the total running time in nanoseconds and n the input size. All experiments were run 100 times and table 4.1 contains the average and the standard deviation (normalized by 99) of these measurements. For each test a new process is created, which reads a randomly generated input file, and measures the running time for one of the algorithms. The measured time includes initialization and starting of threads for MALMS. For TBB and STLMS the running time of the sort routine is measured which includes all initialization times as well.

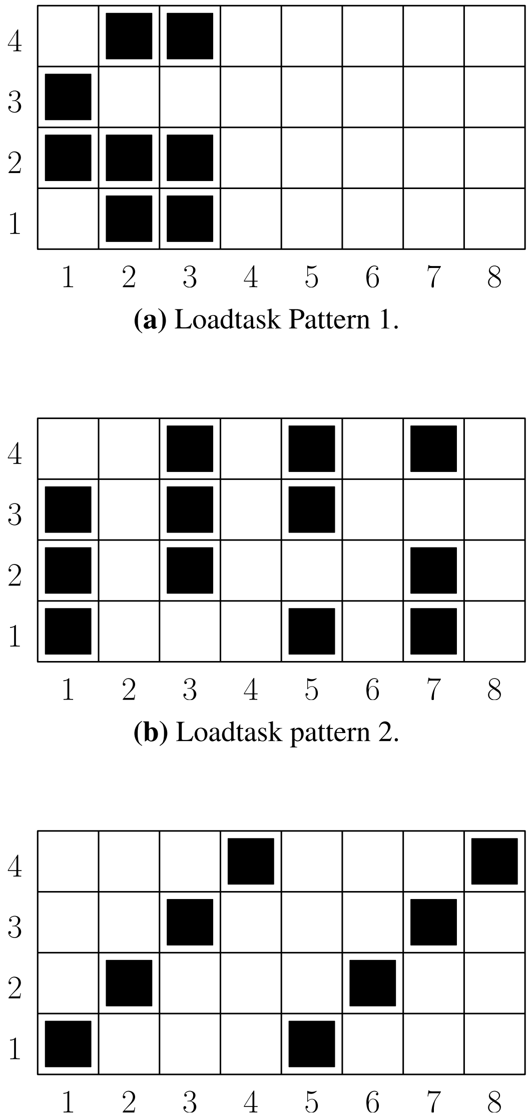
For 10^4 integers none of the parallel sorting algorithms is faster than the se-

Input Size	10^4	10^5	10^6	10^7	10^8
STDSORT	66.5	79.9	94.9	111	126
std	0.36	0.082	0.080	0.0080	0.0023
TBB	151	37.7	31.0	28.1	25.1
std	50	7.3	3.3	1.3	0.96
STLMS	67.2	22.0	19.5	21.1	23.6
std	3.6	1.9	2.0	2.2	1.9
MALMS $k = 8$	111	26.5	19.1	20.4	22.6
std	3.8	2.1	1.7	1.6	1.7
MALMS $k = 24$	147	28.7	20.4	20.8	23.4
std	9.0	0.67	2.7	0.35	0.43
MALMS $k = 48$	212	33.2	20.9	21.1	23.5
std	16	1.1	0.32	0.26	0.18
MALMS $k = 100$	374	46.6	23.3	22.1	24.2
std	27	2.3	0.32	0.10	0.14
MALMS $k = 200$	731	88.3	29.8	23.2	23.5
std	44	3.0	0.29	0.11	0.14
MALMS $k = 400$	1760	237	52.0	28.8	25.4
std	68	5.4	0.30	0.18	0.057

Table 4.1. Running times and standard deviations (std) for STDSORT, STLMS, TBB and MALMS for sorting 32-bit uniformly distributed integers. The running time is given as t/n where t is the total running time in nanoseconds. The table is generated from the data of the experiments done for our article (joint work with Patrick Flick and Peter Sanders [45]).

quential sorting algorithm. Hence only input sizes larger than that are interesting. TBB is usually slower than MALMS (at least for small k) or STLMS. Especially for 10^6 and 10^7 integers to sort and $k = 8, 24, 48, 100$ MALMS and STLMS are clearly faster than TBB. On the other hand STLMS has a similar speed as MALMS for these values of k and input sizes. For smaller input sizes and larger values of k STLMS has a clear advantage which shows the higher initialization overhead of MALMS and the larger effort due to large numbers of workpackages. Hence the choice of k for MALMS is a tradeoff between a good malleability and a good performance.

Now we take a look at the intended use case of the malleable sorting algorithm (MALMS), running with another job in parallel. The other job is Loadtask which uses the available cores in special patterns (the three tested patterns are given in Figure 4.7). We also use two different lengths for the time slots: 2 ms (for 10^6 elements) and 6 ms (for 10^7 elements). Loadtask computes some integer operations on each integer of an array of size 1000 during its activity. The threads of Loadtask are set to the highest priority during the computation.



4		■	■					
3	■							
2	■	■	■					
1		■	■					
	1	2	3	4	5	6	7	8

(a) Loadtask Pattern 1.

4			■		■		■	
3	■		■		■			
2	■		■				■	
1	■				■		■	
	1	2	3	4	5	6	7	8

(b) Loadtask pattern 2.

4				■				■
3			■				■	
2		■				■		
1	■				■			
	1	2	3	4	5	6	7	8

(c) Loadtask pattern 3.

Figure 4.7. The three patterns used by Loadtask. The numbers below the pattern indicate the used core, the numbers on the left the time slots. The black boxes show when Loadtask is using which core. After 4 time slots the pattern is repeated.

All parallel scheduling algorithms (MALMS,TBB and STLMS) are tested in parallel to the same Loadtask patterns (patterns 1,2 and 3). The pattern execution is started directly before the invocation of the sorting algorithm in case of TBB and STLMS. In case of MALMS the pattern execution is started directly after the malleable scheduler is initialized as the scheduler has to be set up to receive the load information from Loadtask. In all cases (MALMS,TBB and STLMS) the initialization is included in the measured running time. We also test how valuable the information from Loadtask is for MALMS by running it without this information (the algorithm is then called MALMS.noinfo). In all cases we run MALMS with $k = 100$.

All test combinations of pattern and sorting algorithm were run 100 times. The running times had a quite high variance, thus we used boxplots to display the information in Figure 4.8. The input data for the sorting algorithms consisted of random uniformly distributed 32-bit integer elements. Looking at the results in Figure 4.8, one can see that MALMS is clearly faster than STLMS for pattern 1 and 2 and in all cases not much slower than the fastest algorithm. The most interesting thing related to this chapter is that MALMS is always faster than MALMS.noinfo (at least slightly). Hence the additional information is a clear advantage.

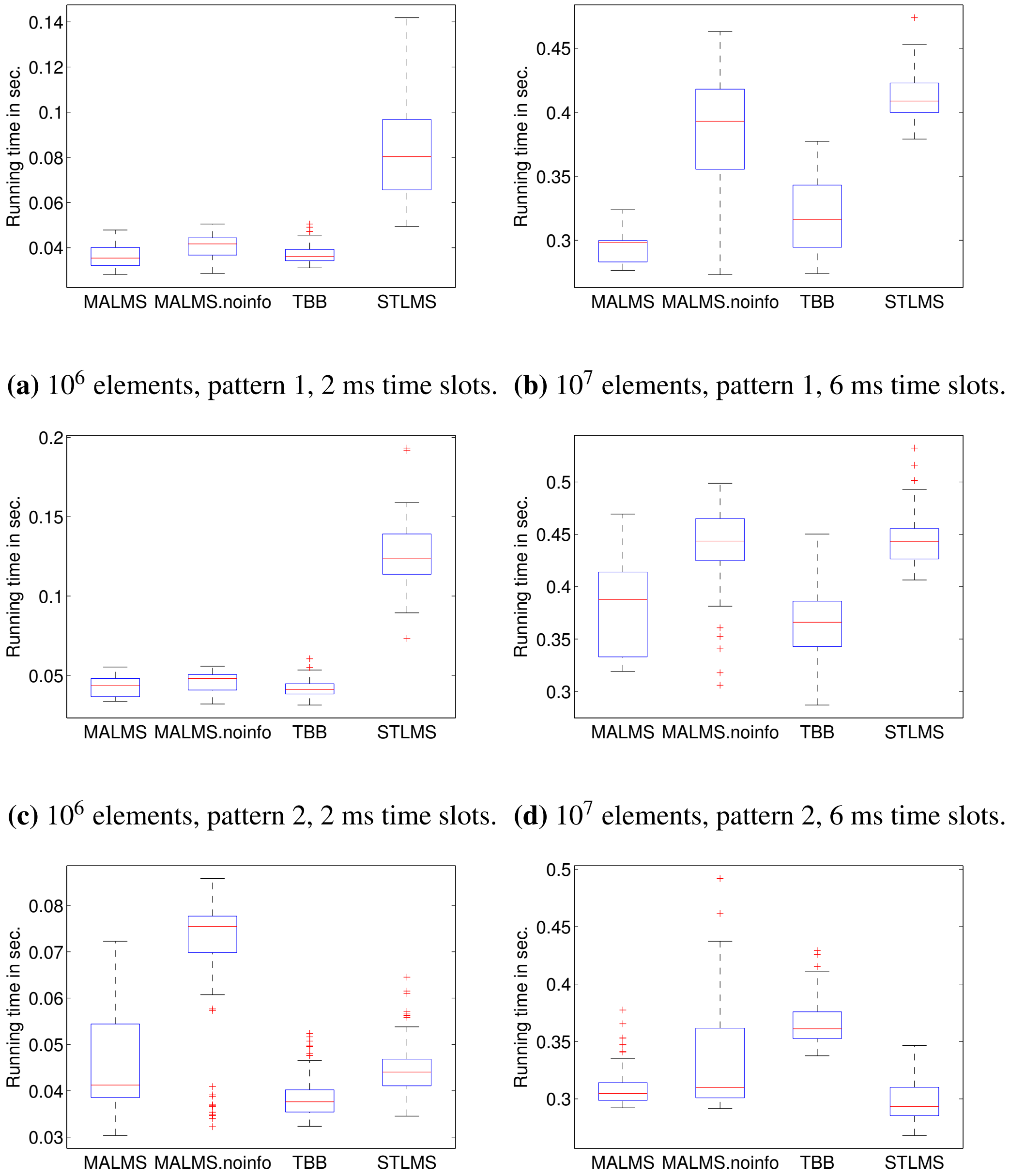
The original experiments used in the publication were done on an Ubuntu 10.04. installation. In order to check the reproducibility, we performed the experiments on four different Ubuntu versions (10.04., 12.04., 14.04., 16.04.), all installed on the same hardware used for the original experiments. The code and test scripts are identical on all OS versions, but we use the compilers and libraries (especially TBB and STLMS) that come along with these distributions for the comparisons. Compared to the results on Ubuntu 10.04. (which are very similar to the original results), STLMS is much faster on the newer distributions when run in parallel to the patterns 1 and 2 of Loadtask. The resulting advantage of MALMS compared to STLMS when run in parallel to these patterns becomes small. Also the advantage of MALMS compared to TBB without a running Loadtask is reduced on the newer distributions but still existent (with $k = 100$). On the other hand the advantage of MALMS compared to MALMS.noinfo especially for pattern 1 and 2 with 6 ms time slots (and 10^7 elements) is very clear on all distributions (for pattern 3 and 6 ms time slots (and 10^7 elements) we have a tie or a small advantage of MALMS.noinfo). Altogether the differences between different distributions (on the same hardware!) are larger than expected.

Altogether MALMS has an advantage compared to STLMS when running in parallel to Loadtask with pattern 1 or 2 (more like a tie on Ubuntu 14.04.) and is close to STLMS when running on an otherwise empty system. Compared to TBB MALMS has an advantage on the otherwise empty system and is close to TBB when running in parallel to Loadtask. The result varies depending on the used Ubuntu version and is less strong on the newer versions. Hence a better understanding about what exactly leads to the advantages of MALMS is needed to get a stable effect over different Linux versions. Also Loadtask is no useful program that might run in parallel to a sorting algorithm in a production environment. Thus understanding the relevant characteristics of jobs which might be competing for resources with the sorting algorithm is needed in order to build a Loadtask that reflects those characteristics. Even though MALMS retreats from cores that will be used by Loadtask, it is not clear that the workpackages of MALMS can be finished without interruption because it is possible that MALMS starts a workpackage on a core immediately before Loadtask wants to use this core. As the workpackages are always finished on the same core and the Linux scheduler cannot be informed, it is possible that the work on this package is interrupted by work of Loadtask.

Malleable sorting in the context of flexible parallel systems MALMS is an example for a malleable job and thus is a result of scheduling theory used as guidance for application development. Also MALMS is an example for a flexible job with a (very simple) malleable interface which makes adaption to the system status possible.

MALMS is a basic example for some of the ideas described in Section 4.5. Especially the decision about the degree of parallelism of the sorting algorithm is moved from compile time to runtime. It is also an example of moving decisions to the information. The number of threads MALMS should use is decided outside of MALMS (within Loadtask) where the information about the work pattern of Loadtask is available. At the same time the knowledge and the decisions about the workpackages are kept internally. Thus MALMS can also be seen as an example for the mission-type tactics ('Auftragstaktik') as the available cores are given but their usage is decided internally. On the other hand when we run Loadtask and MALMS on top of a standard Linux operating system, we have two independent uncoordinated decision makers for the distribution of the cores between the two jobs: the coordination through the malleable interface *and* the scheduler in the Linux kernel. This problem can be solved by using an operating system which includes something similar to the malleable interface instead of establishing coordination between the jobs directly. Such an operating system is developed by the InvasIC project. Another view on malleable applications is the work of Buchwald et al. [20] which describes the programming language support for malleable applications developed within the InvasIC project.

For the future development of malleable applications and systems for these applications a better understanding of the main influences of application performance is needed (caches, interrupts, ...). Such an understanding will also help to guide the development of new malleable applications. Closely related to this is the modelling of the speedups by assigning more resources (for example cores) to a flexible application.



(a) 10^6 elements, pattern 1, 2 ms time slots. **(b)** 10^7 elements, pattern 1, 6 ms time slots.

(c) 10^6 elements, pattern 2, 2 ms time slots. **(d)** 10^7 elements, pattern 2, 6 ms time slots.

(e) 10^6 elements, pattern 3, 2 ms time slots. **(f)** 10^7 elements, pattern 3, 6 ms time slots.

Figure 4.8. The running times of the different sorting algorithms for different input sizes and different Loadtask patterns. Due to the variance we present boxplots. The images are generated from the data measured for our article (joint work with Patrick Flick and Peter Sanders [45]). MALMS is the malleable sorting algorithm, MALMS.noinfo is the same algorithm but without notifications from Loadtask, TBB is the sorting algorithm from Intel's TBB and STLMS is the Multiway Merge Sort. (Big line in the middle of the box: Median, Size of box: Two middle quartiles. The whiskers reach to the most extreme value which is not farther than 1.5 times the box length away from the box. All values farther away are marked as outliers.)

5

Fast and Efficient Schedule Computation

This chapter is devoted to the core of scheduling in computer science, the efficient decision making. Efficient scheduling algorithms not only produce a good schedule according to the given goals but also keep the effort for finding such a schedule low. The tradeoff between effort and schedule quality is depicted in Section 5.1. Solutions for problems with malleable jobs are a major part of this work, the core findings are presented in Section 5.2. Section 5.3 describes an application of general optimization methods to scheduling.

Developing scheduling algorithms is often connected with developing systems as a whole. Possible changes like making all jobs malleable instead of moldable lead to easier scheduling algorithms and shorter schedules but also to some additional effort for the application programmers. Hence developing schedules for a new system always involves some design decisions that are important for scheduling and other parts of the system as well. Especially the distribution of scheduling decisions is an important design decision in this respect (see Chapter 4). An important reason for the distribution of decisions is the possible overload of single decision makers. Within this chapter we access this problem by taking a look into the decision effort. We especially take the system design decisions and the decision distribution between different system components as fixed and look at the scheduling decision making itself.

5.1 Complexity and Justifiable Efforts

Imagine we have to develop a scheduling algorithm for some scheduling problem. Before we start with building a model or collecting the basic properties of the scheduling problem (decision space, properties and constraints, goal and information see Section 1.2), we have to know the justifiable effort for the solution. One effort is the time we are able to spend on the computation of the schedule, the other is the development effort for the scheduling algorithm.

The computation time is closely related to the computational complexity of the scheduling algorithm. The possible computation time differs a lot between different kinds of systems. Scheduling a computing center with jobs that have running times measured in hours or the pre-computation of some schedule for an embedded device that is never changed during runtime are examples where the computation time of a schedule does not matter much. On the other hand we have interactive devices like PCs or mobile phones or interactive embedded devices where a fast schedule computation is important for the user satisfaction or even the proper function of the device.

The possible development effort depends on the importance of good schedules for the operation purpose of the computation device and the number of produced units as a large number of units reduces the development cost per unit.

Altogether we have a tradeoff between three objectives: high schedule quality (regarding the optimization goals), low running time (low computational effort) of the scheduling algorithm and low development effort. Let us first look at the tradeoff between running time and solution quality. Many scheduling problems are NP-hard and trading off running time and solution quality is usually done in the development of approximation algorithms (see Section 3.1.3 and Section 3.1.4). Approximation algorithms often lead to longer running times of the computed schedules or a higher resource usage in order to have a lower running time of the scheduling algorithm itself. Hence for finding a good tradeoff it is important to compare the cost of running the scheduling algorithm to the benefits of a good schedule for the rest of the system. The development of a scheduling algorithm is usually based on a model of the machine and jobs. Upon this model the different possible algorithms are developed: exact computations, approximation algorithms and heuristics. In order to reduce the complexity of a scheduling problem, it is possible to ignore some less relevant parts of the used model or to use a less complicated model instead. There are several reasons why the usage of a sub-complex model might be justified. One example is to ignore caches in models. Of course caches are important for the performance and efficiency of computers, but they are usually not manageable by software-based scheduling algorithms. Thus they are outside of their decision space and can be ignored (although the schedule might influence their effectiveness). This is a common approach for many publications on scheduling, but we are aware that this is not always justified (see Chapter 6). Another example where sub-complex models are used is memory. If several threads work at one problem with the input data coming from a hard drive, the amount of memory used might be proportional to the number of used threads. Hence scheduling memory and cores separately might be possible but not useful. Thus it is easier to schedule resource units that consist of a core and some memory. Hence the usage of simple models can be beneficial to get less complex scheduling problems. The question if the simplified model fits reality good enough can then be tested by experiments. Modifying models and algorithms based on experimental results in a repeated manner is the basic approach of Algorithm Engineering (see Section 2.6.1).

The objective of low development effort is less important in a scientific work like this. But of course also science has to be efficient somehow and thus researchers also try to work on problems which offer the most insights for the least effort. Hence researchers in scheduling theory usually start with simple models because they are easy to justify, easy to work with and the results are hopefully also useful for more complex models and reality. For example results from scheduling theory often use the minimization of C_{max} as goal even though in practice there are many other relevant objectives. Other things that are relevant in practice like throughput, good resource usage and efficiency are often much more difficult to formulate and justify than C_{max}. In order to get a low C_{max}, it is often beneficial to have a good throughput, good resource usage and efficiency. Hence the developed techniques are often also helpful for these goals. This is also a source of differences between theory and practice as described in Section 3.3.

Similar to scheduling a low development effort is an objective for most hard algorithmic problems. Hromkovič [67, Section 7.2] gives some guidance how to approach hard algorithmic problems. He distinguishes between two methods of developing algorithms for hard problems: use of general robust algorithmic design techniques and the development of specialized algorithms for the problem. Usually the development effort for a solution based on general techniques is lower than the development effort of a specialized algorithm. The decision which approach to use is mainly based on the consideration if a possibly higher development effort is justified by the possible gains of a specialized solution. We look into both approaches: in Section 5.2 we describe our approach for specialized scheduling algorithms and in Section 5.3 we present our results of applying general techniques to scheduling. In the remainder of this section we motivate the general approaches used in Section 5.2 and 5.3.

In Chapter 4 we give reasons why improved flexibility of jobs and their coordination through the operating system will be important for future efficient systems. Especially adapting the resource usage of a job during its runtime is an important part of this improved flexibility. The job class in scheduling theory that fits into this requirement of additional flexibility is the class of malleable jobs. Hence one of the main goals of this work is to improve the scheduling of malleable jobs and to show how malleable job scheduling can be used to improve system efficiency. This makes malleable job scheduling a central algorithmic problem of this work. Malleable job scheduling was already successfully investigated in previous work (see Section 3.1.3) as fast algorithms with optimal results already exist. Hence further improvements by the application of general algorithm design techniques are unlikely. Also the nature of schedules for malleable jobs that allow preemptions and migrations every time lead to a large decision space which is also an argument for specialized algorithms instead of general algorithm designs. On the other hand malleable jobs are definitely a sub-complex model. Migration and parallelism degree change costs are ignored, only one kind of resource is considered and changes are possible in an extremely fine granularity. Nevertheless, the model is widely accepted (see previous work in Section 3.1.3) and inspired our work about a sorting algorithm as malleable job ([45], see Section 4.6 for a description). Thus the malleable job model (see Section 3.1.1) is sub-complex but useful. General problems with malleable jobs are often NP-hard, we give an example in [116] (joint work with Peter Sanders). If the problem has the additional property of being a convex problem, it is often possible to find fast algorithms that compute an optimal solution. Convexity is plausible for many problems from reality for two reasons: First, the set of feasible resource distributions between different jobs is often a convex set. Second, the contribution of a job to the overall objective might improve for each added resource unit, but usually the improvement becomes smaller for each additional unit. Hence the objective function is often a convex function. Even if there are some local deviations from the convexity, the global behavior is often very similar to a convex function. Hence we work on convex scheduling problems for malleable jobs because they are interesting themselves and can serve as a global estimation for many other problems. Our approach to malleable job scheduling is described in Section 5.2 where we also describe our results and some further possible applications.

We also take a look at the computation of schedules by general methods in Section 5.3 and present some results there. The main advantage of a scheduling approach with general algorithmic techniques is the lower development effort for the scheduling algorithms. The general techniques make it also easier to work on a more complex model and thus to reduce the modelling errors. On the other hand the general techniques often come along with a high running time and a low efficiency (solution quality divided by running time) as they are not optimized for the particular problem. Hence their usage in practice is restricted to cases in which it is important to have a low development effort or to cases for which specialized algorithms are especially difficult to develop. The main problem of heuristics (for example by the usage of general methods) is to evaluate their solution quality. In most cases optimal solutions are not available, and also the selection of meaningful test instances is often difficult. Hence the quality of heuristics is often evaluated by comparison with other heuristics on a widely accepted set of instances. Another possibility is to compare heuristics in competitions.

5.2 Solutions for Malleable Jobs

As already stated in Section 3.4.3 and Section 5.1 scheduling problems with malleable jobs are one of our main research directions. Malleable jobs are able to use the given resources more efficiently because of their adaptivity, and the arising scheduling problems are often less complex.

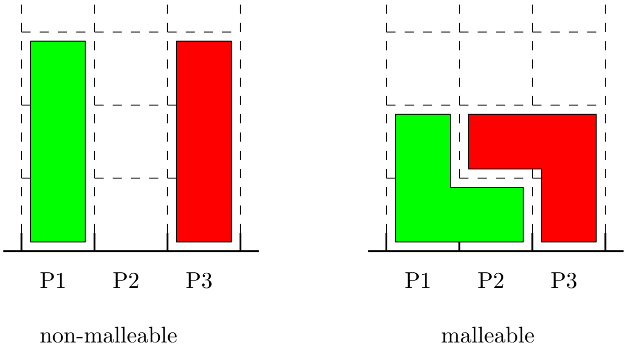


Figure 5.1. An example for a better schedule made possible by malleable jobs.

Example: better efficiency through malleable jobs Assume we are given two identical jobs, a machine with three identical processors and an amount of work for each job which will take three time units in case of sequential execution on each processor. We also assume linear speedup and a maximal degree of parallelism of two. Then fixed-size and moldable parallel jobs lead to an optimal running time of three time units (even with preemption). Only if both jobs are malleable, the running time can be reduced to two time units.

This example shows that through the use of malleable jobs instead of moldable or fixed-size jobs the optimal schedule can be improved. This is due to the higher flexibility of malleable jobs which makes it possible to use otherwise wasted resources.

Malleable jobs are not only helpful to improve the *optimal* schedule. For many scheduling problems it is possible to *find* an optimal schedule with a reasonable complexity for the case of malleable jobs (with somehow restricted speedup functions) as opposed to other kinds of jobs for which finding the optimal schedule for a similar problem is an NP-hard and often computationally infeasible problem (see Section 3.1.3). Especially restrictions that lead to convex problems are an important example which is investigated here.

In previous work ([116] and [117] both joint work with Peter Sanders, for the individual contributions to the articles used in this work see Section 3.4.4) we investigated scheduling algorithms for malleable jobs. In *Efficient Parallel Scheduling of Malleable Tasks* ([116], joint work with Peter Sanders) we investigated the minimization of the maximal finishing time of a set of malleable jobs with concave speedup functions ($P|var|C_{max}$). In this work we speed up and parallelize a solution given in Błażewicz et al. [26] and Błażewicz et al. [24]. In *Energy Efficient Frequency Scaling and Scheduling for Malleable Tasks* ([117], joint work with Peter Sanders) we investigated the minimization of the sum of used energy of a set of malleable jobs with concave speedup functions (plus another more complicated restriction). Although these two articles seem very different, they share a common approach how to find the optimal solution very quickly. In this work we take a different look at the approach: instead of analyzing the problem and then form an algorithm on the basis of the analysis, we start here with the core algorithm that depends on some general conditions and prove that the conditions hold for some scheduling problems afterwards. This leads to a generalization of the algorithmic approach and thus to a possible applicability to other problems and also gives deeper insights into the relevant properties of malleable scheduling problems that allow a fast optimal solution with this approach. The description of this part is structured into four sections. We start by introducing some general properties of convex problems which will be used later in Section 5.2.1. In Section 5.2.2 we introduce an important basic idea for transforming continuous domain scheduling solutions for malleable jobs into solutions for discrete problems which was initially developed by Błażewicz et al. [24] for the problem $P|var|C_{max}$ (with concave speedup functions). The main algorithm is described in Section 5.2.3. In Section 5.2.4 we describe the parallelization of the main algorithm which was initially developed by us ([116], joint work with Peter Sanders) for the problem $P|var|C_{max}$ (with concave speedup functions). Application examples of the main algorithm are given in Section 5.2.5. These examples include the scheduling problems from the two initial works ([116] and [117]), another new example and one important example ($P|var|\sum \omega_i C_i$) for which the conditions of our approach are violated and thus it cannot be applied (we prove the NP-hardness of the problem instead).

The abstract problem that is investigated throughout most of this section is: given n independent malleable jobs and m identical resource units, how should these resources be distributed among the jobs in order to reach an optimal solution? We focus on problems in which the individual cost caused by a job is a convex function of the assigned resources. Convex cost functions are somehow natural as for example the running time of a job might decrease for each further added core, but the running time improvement becomes smaller with each additional core.

5.2.1 Continuous Convex Problems

Scheduling problems with malleable jobs (with convex individual objective functions) often lead to continuous convex problems which are easy to solve. We will describe such problems and how scheduling problems with malleable jobs fit into this class. Convexity considerations are also an important part of the already mentioned work of Błażewicz et al. [26].

There are many textbooks about convex optimization, one of those is by Boyd and Vandenberghe [16], who define a convex problem as follows:

Convex problem

$$\begin{array}{lll} \text{minimize} & f_0(x) & \\ \text{subject to} & f_i(x) \leq b_i, & i = 1, \ldots, k \end{array}$$

where the functions $f_0, \ldots, f_k : \mathbb{R}^n \to \mathbb{R}$ are convex, which means they satisfy

$$f_i(\alpha x + \beta y) \leq \alpha f_i(x) + \beta f_i(y)$$

for all $x, y \in \mathbb{R}^n$ (or the domain of f_i) and all $\alpha, \beta \in \mathbb{R}$ with $\alpha + \beta = 1, \alpha \geq 0, \beta \geq 0$. Functions that instead fulfill $f_i(\alpha x + \beta y) \geq \alpha f_i(x) + \beta f_i(y)$ are called concave. With $<$ instead of $\leq$ and $>$ instead of $\geq$ in case of $x \neq y$ and $\alpha, \beta > 0$ we call the functions strictly convex or strictly concave respectively.

In scheduling, the target function is usually something like C_{max}, the *maximum* of resource dependent functions C_i for each job (for example finishing time) or $\sum E_i$ the *sum* of resource dependent functions E_i for each job (for example energy consumption). If only one kind of resource is important, C_i or E_i usually only depend on x_i which is the i-th component of x. Problems with more than one relevant resource are not covered in this section as we are working on a simple model as described in Section 5.1. The constraints are usually the restrictions given by the fact that the sum of the resources used by the jobs is bounded by the available amount of these resources and that the shares of resources cannot be negative. Hence the restrictions are usually linear for scheduling problems. In order to show that these scheduling problems are often convex problems if the resource dependent functions for each job are convex, we will prove two small lemmata:

Lemma 5.2.1. *Let $f_i(x)$ be convex for $x \in \mathbb{R}^n$ and $i \in \{1, \ldots, n\}$ then $F(x) = \sum_{i=1}^{n} f_i(x)$ is also convex for $x \in \mathbb{R}^n$.*

Proof. We have to prove $F(\alpha x + \beta y) \leq \alpha F(x) + \beta F(y)$ for all $x, y \in \mathbb{R}^n$ and all $\alpha, \beta \in \mathbb{R}$ with $\alpha + \beta = 1, \alpha \geq 0, \beta \geq 0$.

$$\begin{aligned}
F(\alpha x+\beta y) &= \sum_{i=1}^{n} f_i(\alpha x+\beta y) \\
&\text{all} \quad f_i \quad \text{are convex} \\
F(\alpha x+\beta y) &\leq \sum_{i=1}^{n} (\alpha f_i(x)+\beta f_i(y)) \\
&= \alpha \sum_{i=1}^{n} f_i(x)+\beta \sum_{i=1}^{n} f_i(y) \\
&= \alpha F(x)+\beta F(y)
\end{aligned}$$

□

Lemma 5.2.2. *Let $f_i(x)$ be convex for $x \in \mathbb{R}^n$ and $i \in \{1,\dots,n\}$ then $F(x) = \max_{i\in 1\dots n} f_i(x)$ is also convex for $x \in \mathbb{R}^n$ (pointwise maximum).*

Proof. We have to prove $F(\alpha x+\beta y) \leq \alpha F(x)+\beta F(y)$ for all $x, y \in \mathbb{R}^n$ and all $\alpha, \beta \in \mathbb{R}$ with $\alpha+\beta = 1, \alpha \geq 0, \beta \geq 0$.

Let j be such that $f_j(\alpha x+\beta y) = \max_i f_i(\alpha x+\beta y)$, then

$$\begin{aligned}
F(\alpha x+\beta y) &= f_j(\alpha x+\beta y) \\
&f_j \quad \text{is convex} \\
F(\alpha x+\beta y) &\leq \alpha f_j(x)+\beta f_j(y) \\
&\leq \alpha F(x)+\beta F(y)
\end{aligned}$$

□

With these lemmata we can see that typical scheduling problems are convex problems if the relevant job properties are convex in the amount of assigned resources.

There are some properties of convex functions which will be useful for the remainder of this Section:

Lemma 5.2.3. *We now restrict us to functions on $\mathbb{R}$, ($x \in \mathbb{R}$). For a convex function $g : x \mapsto g(x)$ the functions $x \mapsto c_1 g(x)$ and $x \mapsto g(x)+c_2$ with $c_1 \in \mathbb{R}_{>0}$ and $c_2 \in \mathbb{R}$ are also convex.*

For a concave function $h : x \mapsto h(x)$, the function $r : x \mapsto 1/h(x)$ is convex on the domain D of h with $h(x) > 0\ \forall x \in D$ and D being a closed or open (possibly unbounded) interval.

If h is strictly monotonic on D, then r is strictly convex.

Proof. Only the (strict) convexity of $r : x \mapsto 1/h(x)$ needs to be proven.

In order to show the strict convexity, we just need $h(x) \neq h(y)$ for all $x, y \in D$ with $x \neq y$. We show the strict convexity for this case and $\alpha, \beta > 0$ with $\alpha + \beta = 1$, the proposition for the general convexity then only needs to be checked for the case of $h(x) = h(y)$. We have to show:

$$\begin{array}{rl} & r(\alpha x+\beta y) < \alpha r(x) + \beta r(y) \ \text{ strict convexity of } r \\ \Leftrightarrow & \dfrac{1}{h(\alpha x+\beta y)} < \alpha \dfrac{1}{h(x)} + \beta \dfrac{1}{h(y)} \end{array}$$

As h is concave, we have $h(\alpha x + \beta y) \geq \alpha h(x) + \beta h(y)$ and it is thus sufficient to show:

$$\begin{array}{rrl} & \dfrac{1}{\alpha h(x)+\beta h(y)} & < \alpha \dfrac{1}{h(x)} + \beta \dfrac{1}{h(y)} \\ \Leftrightarrow & 1 & < \alpha^2 + \alpha\beta \dfrac{h(y)}{h(x)} + \alpha\beta \dfrac{h(x)}{h(y)} + \beta^2 \\ & \text{with } 1 & = (\alpha+\beta)^2 \\ \Leftrightarrow & 2\alpha\beta & < \alpha\beta \dfrac{h(y)}{h(x)} + \alpha\beta \dfrac{h(x)}{h(y)} \\ \Leftrightarrow & 2h(x)h(y) & < h^2(x) + h^2(y) \\ \Leftarrow & 0 & < (h(x) - h(y))^2 \ \text{ as } h(x) \neq h(y) \end{array}$$

For the case of $h(x) = h(y)$ we have to show:

$$\begin{array}{rl} & \dfrac{1}{h(\alpha x+\beta y)} \leq \alpha \dfrac{1}{h(x)} + \beta \dfrac{1}{h(y)} \\ \Leftrightarrow & \dfrac{1}{h(\alpha x+\beta y)} \leq \dfrac{1}{h(x)} \\ \Leftrightarrow & h(\alpha x+\beta y) \geq h(x) \\ \Leftrightarrow & h(\alpha x+\beta y) \geq \alpha h(x) + \beta h(y) \ \text{ concavity of } h \end{array}$$

It is worth to note that this proof does not work the other way round (convex h and concave r) and not for h with a negative image. □

In order to solve some convex scheduling problems, we also need to use the derivatives of the occurring convex functions. Thus we present some lemmata which connect convex functions with the properties of derivatives. First we take a lemma from Walter [140, page 303] (the lemma is shortened and simplified for our needs):

Lemma 5.2.4. *Let f be a convex function on the interval I and I^0 an open interval contained in I. Then f is continuous on I^0. The left $\overrightarrow{f}$ and right $\overleftarrow{f}$ derivatives exist in I^0 and it holds:*

$$\overrightarrow{f}(x) \leq \overleftarrow{f}(x) \leq \overrightarrow{f}(y) \leq \overleftarrow{f}(y) \qquad \text{for } x < y \qquad x,y \in I^0$$

Especially, $\overrightarrow{f}$ and $\overleftarrow{f}$ are monotonically increasing in I^0.

Lemma 5.2.5. *For a strictly convex function f we even have*

$$\overrightarrow{f}(x) \leq \overleftarrow{f}(x) < \overrightarrow{f}(y) \leq \overleftarrow{f}(y) \qquad \text{for } x < y \qquad x,y \in I^0$$

in Lemma 5.2.4.

Proof. The proof is done by contradiction. Assume a strictly convex function f and $x,y \in I^0$ with $x < y$ and $\overleftarrow{f}(x) = \overrightarrow{f}(y)$. Then we have for each $z \in (x,y)$ that $f'(z)$ exists and $f'(z) = \overleftarrow{f}(x) = \overrightarrow{f}(y)$. Hence f is differentiable and its derivative is integrable on the interval $[x,y]$. We now set $w = (x+y)/2$. Then we get by the fundamental theorem of calculus (see for example Walter [140, page 260], *Zweiter Hauptsatz*) that

$$\int_x^w f'(z)dz = f(w) - f(x) \qquad \text{and} \qquad \int_w^y f'(z)dz = f(y) - f(w)$$

and thus $f(w) - f(x) = f(y) - f(w)$ which leads to $f(1/2 \cdot x + 1/2 \cdot y) = 1/2 \cdot f(x) + 1/2 \cdot f(y)$ which is a contradiction to the strict convexity of f. □

For the examples it is often necessary to prove that a function is convex or even strictly convex. Thus we give a lemma that is useful for this task.

Lemma 5.2.6. *Given a continuous function f on the interval $(0,m]$ which is continuously differentiable on the whole interval except for a finite set of points $0 < b_1 < \cdots < b_\ell \leq m$ called bend points. For each of these bend points b_i the left and right derivatives exist and it holds $\overrightarrow{f}(b_i) \leq \overleftarrow{f}(b_i)$. If the derivatives on the intervals $(0,b_1),(b_1,b_2),\ldots,(b_\ell,m)$ are (strictly) monotonic increasing, then f is (strictly) convex.*

Proof. In order to prove the (strict) convexity of f, we assume three arbitrary points $x,y,z \in (0,m]$ with $x < y < z$. Let be $\alpha,\beta \in (0,1)$ with $\alpha x + \beta z = y$ and $\alpha + \beta = 1$. We now have to show $f(y) \leq \alpha f(x) + \beta f(z)$ or $f(y) < \alpha f(x) + \beta f(z)$ to prove that f is (strictly) convex.

From the fundamental theorem of calculus (see for example Walter [140, page 260], *Zweiter Hauptsatz*) we know that $\int_{b_i}^{b_{i+1}} f'(w)dw = f(b_{i+1}) - f(b_i)$. We also know that for the appropriate i,j,k ($x \leq b_i$ and no bend point in between,

$b_j \le y \le b_{j+1}$ and $b_k \le z$ and no bend point in between) we have $\int_x^{b_i} f'(w)dw = f(b_i) - f(x)$ and $\int_{b_j}^{y} f'(w)dw = f(y) - f(b_j)$ and $\int_y^{b_{j+1}} f'(w)dw = f(b_{j+1}) - f(y)$ and $\int_{b_k}^{z} f'(w)dw = f(z) - f(b_k)$ if the respective bend points are not 0 or m.

Hence we can now compute $f(y) - f(x)$ and $f(z) - f(y)$. If x and y are in the same interval, we have $f(y) - f(x) = \int_x^y f'(w)dw$ otherwise we have

$$f(y) - f(x) = \int_x^{b_i} f'(w)dw + \sum_{h=i}^{j-1} \int_{b_h}^{b_{h+1}} f'(w)dw + \int_{b_j}^{y} f'(w)dw$$

As a finite number of points are irrelevant for the integral, we can just write $f(y) - f(x) = \int_x^y f'(w)dw$ in both cases. Similarly $f(z) - f(y) = \int_y^z f'(w)dw$.

Let $C = f'(y)$ or $C = \overrightarrow{f}(y)$ if $f'(y)$ does not exist. Then we have $f'(w) < C$ for all $w \in (x,y)$ and $f'(w) > C$ for all $w \in (y,z)$ in the strictly monotonic case (in the monotonic case we have $\le$ and $\ge$ instead) except for bend points where $f'(w)$ does not have to exist. The finite number of bend points do not change the integral. Thus we have $f(z) - f(y) > C \cdot (z - y) = C \cdot \alpha(z - x)$ and $f(y) - f(x) < C \cdot (y - x) = C \cdot \beta(z - x)$. Thus $\beta(f(z) - f(y)) > C \cdot \alpha\beta(z - x) > \alpha(f(y) - f(x))$. This leads to $\alpha f(x) + \beta f(z) > f(y)$ which was to prove. In the non-strict case we only have to replace $<$ by $\le$ and $>$ by $\ge$ and also get the desired result. □

Lemma 5.2.7. *A convex function $f: (0,m] \to \mathbb{R}$ with $f(x) \xrightarrow[x \to 0]{} \infty$ is strictly monotonically decreasing on $(0,\hat{x})$ where $\hat{x}$ denotes the smallest value for which f reaches its minimum.*

Proof. **Assume:** f is not strictly monotonically decreasing on $(0,\hat{x})$. Hence there exist x,y with $0 < x < y < \hat{x}$ and $f(x) \le f(y)$ but $f(x), f(y) > f(\hat{x})$. Let $\alpha, \beta > 0$ with $\alpha + \beta = 1$ and $\alpha x + \beta \hat{x} = y$. Then $\alpha f(x) + \beta f(\hat{x}) < f(y)$ which contradicts the convexity. Due to the continuity of convex functions (see Lemma 5.2.4) and $f(x) \xrightarrow[x \to 0]{} \infty$ the minimum of f is realized for an $\hat{x} \in (0,m]$. □

5.2.2 Using Continuous Domain Solutions for Discrete Problems

An important difference of convex problems to typical scheduling problems is that the variables x are continuous whereas the variables in scheduling problems are often discrete (amount of cores a job gets for example). For large quantities like the number of 4KB tiles of the 16GB main memory of a workstation that are divided between different jobs (4 million units!) one can just round the optimal solution to the next smaller integer value and assume that this does not change the result much. In case of resources like cores where most machines have maybe tens or hundreds of units the rounding errors might be not that small any more. A solution how to overcome this problem in case of malleable tasks (with one relevant resource) is given by Błażewicz et al. [24]. The idea was used and simplified by us in *Efficient Parallel Scheduling of Malleable Tasks* ([116], joint work with Peter Sanders). We present a version of this idea adapted to our approach.

Let $f: \mathbb{N} \to \mathbb{R}$ be the function from resource usage to some scheduling-relevant property of a job, for example the running time or the energy consumption on x cores ($x \in \mathbb{N}$). The domain of f can also be $\{k \in \mathbb{N} \mid k \leq \ell\}$ for an $\ell \in \mathbb{N}$. A function f defined on $\mathbb{N}$ is convex if $f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$ for all $x, y \in \mathbb{N}$ and all $\alpha, \beta \in [0,1]$ such that $\alpha x + \beta y \in \mathbb{N}$. Such functions are typical for scheduling problems with malleable jobs. An example of such a discrete convex function can be seen in Figure 5.2. We now show what a natural extension on a continuous domain looks like for such functions and that it is useful.

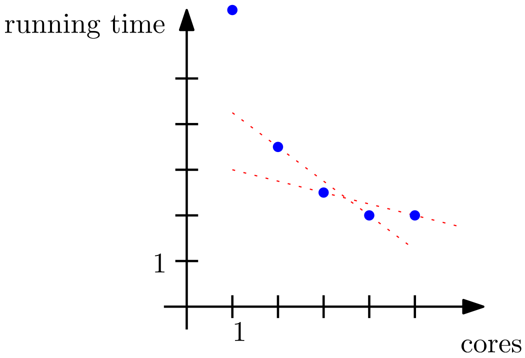


Figure 5.2. An example of a discrete convex function which maps the number of cores to the running time (the red dashed lines indicate the convexity).

First we have to define what is meant if a job is given a resource amount of x with $x \in \mathbb{R}_{>0} \setminus \mathbb{N}$ for a time interval of length T in the case of indivisible resources like cores or memory tiles.

Definition 5.2.8. *Let $f: \mathbb{N} \to \mathbb{R}$ be a discrete function from resource usage to some scheduling-relevant property of a job. The job runs for some time interval of length T (running time of the whole schedule) with an average amount of resources x. Then we define the continuous domain extension $\overline{f}: \mathbb{R}_{>0} \to \mathbb{R}$ by the resulting value for the property if the job runs on $\lfloor x \rfloor$ resource units for time $(1 - x + \lfloor x \rfloor) \cdot T$ and on $\lfloor x \rfloor + 1$ resource units for the remaining time ($(x - \lfloor x \rfloor) \cdot T$).*

There are several things to note about Definition 5.2.8:

- The definition leads to $\overline{f}(x) = f(x)$ for $x \in \mathbb{N}$. Depending on the property $\overline{f}(x)$ this might not be just the weighted average of $f(\lfloor x \rfloor)$ and $f(\lfloor x \rfloor + 1)$. Especially the definition does not define how the values of $\overline{f}$ can be computed as the value for running on $\lfloor x \rfloor$ resource units for time $(1 - x + \lfloor x \rfloor) \cdot T$ and on $\lfloor x \rfloor + 1$ resource units for the remaining time $((x - \lfloor x \rfloor) \cdot T)$ is dependent on the scheduling problem.

- As we assume the jobs to be malleable, the scheduling-relevant property has a value for each possible distribution of the times (during T) the job runs on $\lfloor x \rfloor$ or $\lfloor x \rfloor + 1$ resource units. For example a job which uses four cores for half of the time and five cores for the other half of the time has a certain energy consumption.
- For our main algorithm we will require that the value of the scheduling-relevant property stays the same for all possible sequences of running on $\lfloor x \rfloor$ or $\lfloor x \rfloor + 1$ resource units as long as the average resource usage stays the same (condition 1 in Definition 5.2.10, see below). Thus the placement explained below will never change the objective value of the schedule.
- The restriction to $\lfloor x \rfloor$ or $\lfloor x \rfloor + 1$ resource units to reach an average of x might lead to suboptimal results for some scheduling problems. Condition 1 in Definition 5.2.10 only allows scheduling problems which still allow an optimal result with this restriction.
- Examples of scheduling problems (each with a problem-specific $\overline{f}$) can be found in Section 5.2.5. Each of these problems results in a different (problem-specific) computation for $\overline{f}$.

The next step is to prove that such a resource allocation $(x_1, \dots, x_n) \in \mathbb{R}^n_{>0}$ during a time interval of length T with $\sum_{j=1}^n x_j \leq m$ for a maximum of m available resource units is possible for malleable jobs and indivisible resources. We will show how to place all n malleable jobs within the time interval $[0, T]$ with each job j using $\lfloor x_j \rfloor$ and $\lfloor x_j \rfloor + 1$ resource units for some fraction of the time interval such that the average resource usage is x_j.

Discretization of a solution for malleable jobs We are given a resource allocation $(x_1, \dots, x_n) \in \mathbb{R}^n_{>0}$ during the time interval $[0, T]$. With some simple instructions we can build a discrete resource allocation for malleable jobs. All we have to do is to fit all jobs within the $[0, T] \times [0, m]$ time resource rectangle. A more formal version of this is given by Błażewicz et al. [24].

Job 1 gets $\lfloor x_1 \rfloor + 1$ units of the resource for the time interval $[0, (x_1 - \lfloor x_1 \rfloor)T]$ and $\lfloor x_1 \rfloor$ units of the resource for the time interval $[(x_1 - \lfloor x_1 \rfloor)T, T]$ (a duration of $(1 - x_1 + \lfloor x_1 \rfloor)T$ which leads to an average resource usage of x_1 for the full time interval $[0, T]$. After placing job 1 we have a step at time $(x_1 - \lfloor x_1 \rfloor)T$ (possibly time 0 for $x_1 = \lfloor x_1 \rfloor$) where the number of used resources is reduced by one, otherwise the resource usage remains constant.

Job 2 similarly runs on $\lfloor x_2 \rfloor + 1$ resources for $(x_2 - \lfloor x_2 \rfloor)T$ time units and on

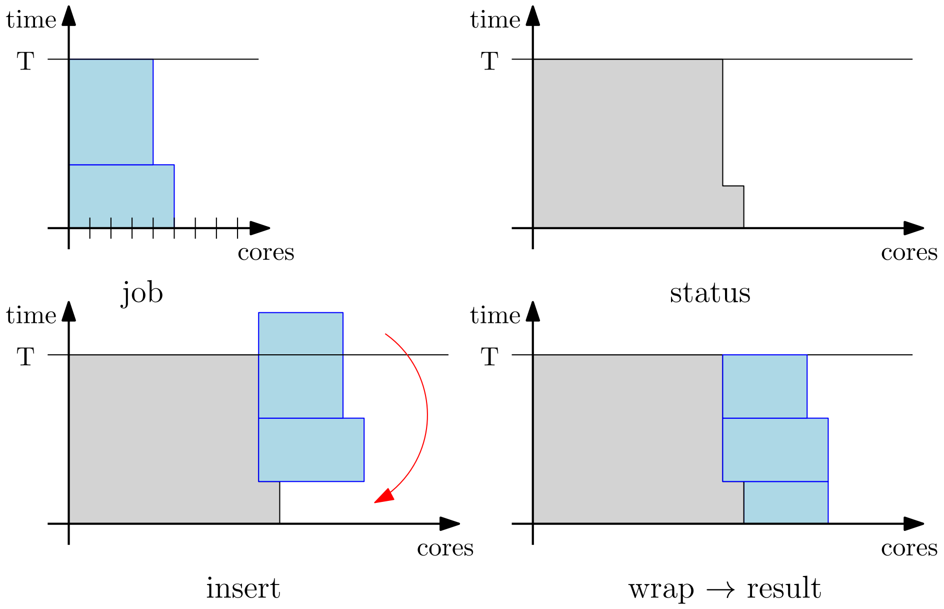


Figure 5.3. A malleable job which runs on 5 or 4 cores during its running time is placed on an already partially loaded machine.

$\lfloor x_2 \rfloor$ resources for $(1-x_2+\lfloor x_2 \rfloor)T$ time units leading to an average resource usage of x_2. We put job 2 into the time resource rectangle starting at the time of the step (with $\lfloor x_2 \rfloor + 1$ resources) and the remaining part of job 2 is wrapped around to start from time 0. If job 2 uses two different amounts of resources during its running time, the original step is removed, but a new one can be created if $(x_2 - \lfloor x_2 \rfloor)T$ is different from the time between the original step and T. If $x_2 = \lfloor x_2 \rfloor$, the original step is just shifted within the time resource rectangle. Hence job 2 can be placed without holes and such that the border between the used part of the time resource rectangle and the remaining part contains at most one step. Figure 5.3 gives an example where the resource type is cores.

The remaining jobs can be placed like job 2 and thus after every placed job, there is at most one step between the used and not used part of the time resource rectangle. As every job j uses an area of $x_j \cdot T$ in the time resource rectangle and there are no holes, all jobs can be placed within $[0,T] \times [0,m]$ because $\sum_{j=1}^{n} x_j \leq m$. Every job can be placed in time $\mathcal{O}(1)$. Hence we have proven the following lemma which also fits Definition 5.2.8 for the continuous domain extension:

Lemma 5.2.9. *A resource allocation $(x_1,\ldots,x_n) \in \mathbb{R}^n_{>0}$ for malleable jobs during the time interval $[0,T]$ with $\sum_{i=1}^{n} x_i \leq m$ can be satisfied within the $[0,T] \times [0,m]$ time resource rectangle with each job j running on $\lfloor x_j \rfloor$ or $\lfloor x_j \rfloor + 1$ resource units throughout $[0,T]$. The placement can be computed in time $\mathcal{O}(n)$.*

For the placement of a job we just need the position of the step after which it is placed. Hence we can parallelize the placement by computing (as a collective operation) the prefix sum of the resource usages of the jobs in the order of their placement. More details on the parallelization are given in Section 5.2.4.

With this we have now shown how to transform a non-integer resource allocation into an integer allocation for malleable jobs. This is the basis to form continuous domain extension functions $\overline{f}_j : \mathbb{R}_{>0} \to \mathbb{R}$ from the given discrete functions $f_j : \mathbb{N} \to \mathbb{R}$. In order to get a convex problem, we have to show that the resulting functions $\overline{f}_j$ are convex (and thus continuous, see Lemma 5.2.3) for the investigated problem. Some problems where this holds are described in Section 5.2.5.

5.2.3 Core Algorithm

This section is a generalization of our work presented in [116] and [117] (both joint work with Peter Sanders). In order to shorten the description, we will call the case in which the objective is to minimize a common maximum the max-case and the case with the objective of minimizing a sum the sum-case. The sum-case was developed in [117] (joint work with Peter Sanders) for the special case of energy minimization and to our knowledge was not used before. In both cases we have a set of independent malleable jobs j and for each job a function $f_j : \mathbb{N} \to \mathbb{R}$ which is convex or even strictly convex and provides an objective value for each amount of assigned resources. We do not assume that our algorithm has to read all possible function values for each possible resource amount ($n \cdot m$ values, see discussion below). Hence running times sublinear in nm are useful. As the jobs are malleable it is possible that a job runs on different resource amounts during the duration of the schedule $[0, T]$.

In the beginning of this section we introduce some conditions that a scheduling problem has to fulfill to be solvable with our approach. After the conditions we state the main theorem and then we give a description of the core scheduling algorithm. The core algorithm uses interval splitting as the basic technique to find a $D^* = \overline{f_1}(x_1^*) = \cdots = \overline{f_n}(x_n^*)$ (max-case) or $D^* = \overline{f_1}'(x_1^*) = \cdots = \overline{f_n}'(x_n^*)$ (sum-case). It is shown below that this leads to an optimal solution.

With Definition 5.2.8 on the $f_j : \mathbb{N} \to \mathbb{R}$ we get continuous domain functions $\overline{f_j} : \mathbb{R}_{>0} \to \mathbb{R}$ which map the average resource usage throughout the running time of the schedule to an objective value. The $\overline{f_j}$ and their computation are problem-specific. Thus without having a specific problem we can only define/require some general properties. These continuous domain functions need to be well-defined which means in our case that the function value is independent of the times when one resource amount is used or the other. All function values with the same average resource usage must be the same (as long as only two neighboring resource amounts are used). Special care needs to be taken in case of problems in which the function value depends on the finishing time of a job for jobs which get an average resource amount of less than 1. Also we need to make sure that an optimal solution can be found within the set of solutions in which each job only uses two neighboring resource amounts throughout the running time of the schedule. This implicitly means that there is an optimal solution in which all jobs which get an resource amount of 1 or more run thoughout the whole duration of the schedule.

Altogether we get the following **condition 1**: the continuous domain functions built according to Definition 5.2.8 are well-defined. The set of schedules in which a job uses only two neighboring resource amounts throughout the running time of the schedule forms a dominant set for the problem and it does not matter for the overall objective value when during $[0,T]$ a job uses the one amount or the other (as long as the average is correct).

The last part of the condition comes from the problem that the translation from continuous domain to discrete schedules given in Section 5.2.2 does not guarantee the times during which a job will run on one resource amount or the other. That the overall objective value (which is the sum or the maximum of the $\overline{f_j}$) cannot change due to the times a job uses one resource amount or the other is already given as the $\overline{f_j}$ are well-defined and thus cannot change their objective value due to a change of time when they use one resource amount or the other (as long as the average stays the same). We keep the restriction in the condition to make things better understandable. With that condition we now have a continuous domain scheduling problem. The scheduling problem has the objective to find a resource distribution $(x_1,\dots,x_n) \in \mathbb{R}^n_{>0}$ (with $\sum_{j=1}^n x_j \leq m$) which minimizes the sum or the maximum of the function values $\overline{f_j}(x_j)$ for each job j.

We now take a deeper look at the functions $\overline{f_j} : \mathbb{R}_{>0} \to \mathbb{R}$ built according to Definition 5.2.8. The definition extends the function on $\mathbb{N}$ to a function on $\mathbb{R}_{>0}$ by problem-specific interpolations on the intervals $(k,k+1)$. The points where two interpolations meet (usually a $k \in \mathbb{N}$) are called **bend points** (their number is in $\mathcal{O}(m)$). We assume that the interpolations are done in a way that leads to a continuously differentiable function on the intervals $(k,k+1)$ between the integers for $k \in \{0,\dots,m-1\}$. A slightly generalized condition is that the $\overline{f_j}$ are continuously differentiable on $(0,m]$ except for the set of bend points, in which the left and right derivative might differ but the left derivative is continuous from the left and the right derivative from the right. We also assume that the whole function on $(0,m]$ is continuous. We further assume that a job that gets no resources at all will produce a very bad objective value. Thus we assume $\overline{f_j}(x) \xrightarrow[x\to 0]{} \infty$. The main assumption is that the objective functions $\overline{f_j}$ are convex. From Lemma 5.2.3 we know that for example if the objective function is the running time of the job, a concave speedup function guarantees the assumed convexity. Lemma 5.2.4 shows that convex functions are continuous and thus continuousity is always fulfilled for convex functions. Due to their convexity each function $\overline{f_j}$ must be strictly monotonically decreasing between 0 and its first minimum $\hat{x}_j$ and monotonically increasing for all $x \geq \hat{x}_j$ (see Lemma 5.2.7). Also due to the convexity the derivatives are monotonically increasing (see Lemma 5.2.4). In fact we do not really need the left and right derivatives for the bend points. Throughout this work having $\lim_{x \to b} \overline{f_j}'(x)$ for bend points b would be sufficient as the proofs use the integrals over $\overline{f_j}'$ which are not affected by the change of a single value. On the other hand the presentation becomes simpler by using the left and right derivatives.

As the objective is minimization and we do not have to assign all resources, we can also assume that the functions are constant for $x \geq \hat{x}_j$ as assigning more than $\hat{x}_j$ resources brings no benefit. For the sum-case we additionally require strict convexity on the interval $(0, \hat{x}_j)$. Altogether we assume the following **condition 2** to be fulfilled for all continuous domain functions $\overline{f_j}$: all functions $\overline{f_j}$ are convex, defined on $(0,m]$, fulfill $\overline{f_j}(x) \xrightarrow[x \to 0]{} \infty$ and their number of bend points is in $\mathcal{O}(m)$. In the sum-case we additionally require strict convexity between 0 and their first minimum $\hat{x}_j$ and that the functions are continuously differentiable on $(0,m]$ except for the bend points, in which the left and right derivative might differ but the left derivative is continuous from the left and the right derivative from the right.

In order to compute an optimal solution (find D^* with $D^* = \overline{f_1}(x_1^*) = \cdots = \overline{f_n}(x_n^*)$ (max-case) or $D^* = \overline{f_1}'(x_1^*) = \cdots = \overline{f_n}'(x_n^*)$ (sum-case)), we have to provide a way to invert the functions $\overline{f_j}$ (max-case) or their derivatives $\overline{f_j}'$ (sum-case) on the interval $(0, \hat{x}_j]$. As we do not assume the algorithm to read all possible function values of the f_j (or even worse the $\overline{f_j}$) for input values $x_j \leq \hat{x}_j$ we have to define how our algorithm can access these values and the effort for such an access.

The inversion is also the reason why the functions/derivatives cannot be only monotonically decreasing/increasing but have to be strictly monotonically decreasing/increasing. If there is a difference between the left and right derivative of $\overline{f_j}$ at a bend point b, the inversion maps all values between the left and right derivative to b. We assume that the interpolations on the intervals $(k, k+1)$ are given in a closed algebraic form and that we can access the value of $f_j(k)$ in time $\mathcal{O}(1)$ for each job j and each $k \in \{1, \ldots, m\}$. Thus the inverse can be computed in $\mathcal{O}(\log m)$ by finding the correct interval $(k, k+1)$ (or more generally the appropriate interval between neighboring bend points) through interval halving and then inverting the function given on $(k, k+1)$ in constant time. Similarly the minima $\hat{x}_j$ can be computed in time $\mathcal{O}(\log m)$. From Definition 5.2.8 we have $\overline{f_j}(x) = f_j(x)$ for $x \in \mathbb{N}$.

By interval halving we can try to find a $k \in \mathbb{N}$ with $f_j(k-1) > f_j(k) \leq f_j(k+1)$. Due to the convexity of f_j the smallest minimum $\hat{x}_j$ can be found in $(k-1, k+1)$ if such a k exists or in $(m-1, m]$ in the other cases. Usually it is possible to compute the number of bend points of a function between two points $x, y \in (0, \hat{x}_j]$ in time $\mathcal{O}(1)$ because the number of bend points in (x, y) is just the number of integers in that interval. This is **condition 3**: A computation is given for $\overline{f_j}$ (max-case) or $\overline{f_j}'$ and $\overline{f_j}$ (sum-case) and the respective inversion on the interval $(0, \hat{x}_j]$ which runs in time $\mathcal{O}(\log m)$. This includes computations for the left $\overrightarrow{f_j}$ and right $\overleftarrow{f_j}$ derivatives for the sum-case. For all minimum points $\hat{x}_j < m$ we have $\overleftarrow{f_j}(\hat{x}_j) \geq 0$ as the function cannot decrease after the minimum. For $\hat{x}_j = m$ the right derivative $\overleftarrow{f_j}(\hat{x}_j)$ does not matter as it is outside of the usable definition space. To simplify the exposition we set $\overleftarrow{f_j}(m) = 0$ in the max-case when $\hat{x}_j = m$ to avoid some unnecessary distinctions of cases. For a bend point x we set $(\overline{f_j}')^{-1}(D) = x$ for any D with $\overrightarrow{f_j}(x) \leq D \leq \overleftarrow{f_j}(x)$. We also assume that the $\hat{x}_j$ are given or can be easily computed (in time $\mathcal{O}(\log m)$). For the bend points we assume that the number of bend points within the interval (x, y) or $[x, y]$ for $0 < x < y \leq \hat{x}_j$ can be computed in time $\mathcal{O}(1)$ and that we can find the k-th bend point of a job in time $\mathcal{O}(1)$ for each k.

Because of the interval splitting approach, the general algorithm can also only narrow down the optimal solution. For many scheduling problems with malleable jobs, an exact solution can be computed very quickly if the optimal resource assignments for each job are known to be contained in an interval between two neighboring bend points or known to be the value of a single bend point. This is usually the case if no $\overline{f_j}(b)$ (max-case) or $\overrightarrow{f_j}(b)$ or $\overleftarrow{f_j}(b)$ (sum-case) for any bend point b is contained within the upper $\overline{D}$ and lower $\underline{D}$ bound for D^* (boundaries not included). If such a computation exists for a problem we call it *final solution*. Otherwise we can only compute an approximate solution. Hence there is one last condition for finding the optimal solution for a scheduling problem, **condition 4**: a final solution can be computed in time $\mathcal{O}(n(\log n + \log m)\log(nm))$ if no $\overline{f_j}(b)$ (max-case) or $\overrightarrow{f_j}(b)$ or $\overleftarrow{f_j}(b)$ (sum-case) is contained within $(\underline{D}, \overline{D})$ for any bend point b. The maximal running time stems from the running time of the core algorithm which will not be slowed down by such a final solution computation in the $\mathcal{O}$-calculus. The general idea why such a final solution might exist is that the functions/derivatives $\overline{f_j}$ or $\overline{f_j}'$ are often given as a closed formula between two neighboring bend points which often enables the computation of a final solution by using some algebra.

Altogether we have to provide four things (or proofs for them) to obtain an optimal schedule through the usage of the core algorithm for a scheduling problem:

Definition 5.2.10. *A scheduling problem is said to fulfill the basic conditions if the following four conditions are met for this problem:*

1. *The continuous domain functions built according to Definition 5.2.8 are well-defined. The set of schedules in which a job uses only two neighboring resource amounts throughout the running time of the schedule forms a dominant set for the problem and within that set it does not matter for the overall objective value for which part of $[0,T]$ a job uses one amount or the other (as long as the average is correct).*
2. *All functions $\overline{f_j}$ are convex, defined on $(0,m]$, fulfill $\overline{f_j}(x) \xrightarrow[x\to 0]{} \infty$ and their number of bend points is in $\mathcal{O}(m)$. In the sum-case we additionally require strict convexity between 0 and their first minimum $\hat{x}_j$ and that the functions are continuously differentiable on $(0,m]$ except for the bend points. In the bend points the left and right derivative might differ, but the left derivative is continuous from the left and the right derivative from the right.*
3. *A computation is given for $\overline{f_j}$ (max-case) or $\overline{f_j}'$ and $\overline{f_j}$ (sum-case) and the respective inversion on the interval $(0,\hat{x}_j]$ which runs in time $\mathcal{O}(\log m)$. This includes computations for the left $\overrightarrow{f_j}$ and right $\overleftarrow{f_j}$ derivatives for the sum-case. To simplify the exposition we set $\overleftarrow{f_j}(m) = 0$ in the sum-case when $\hat{x}_j = m$ and for a bend point x we set $(\overline{f_j}')^{-1}(D) = x$ if $\overrightarrow{f_j}(x) \leq D \leq \overleftarrow{f_j}(x)$. We also assume that the $\hat{x}_j$ are given or can be easily computed (in time $\mathcal{O}(\log m)$). For the bend points we assume that the number of bend points within the interval (x,y) or $[x,y]$ for $0 < x < y \leq \hat{x}_j$ can be computed in time $\mathcal{O}(1)$ and that we can find the k-th bend point of a job in time $\mathcal{O}(1)$ for each k.*
4. *Assume we know a lower $\underline{D}$ and an upper $\overline{D}$ bound for D^* and no $\overline{f_j}(b)$ (max-case) or $\overrightarrow{f_j}(b)$ or $\overleftarrow{f_j}(b)$ (sum-case) for any bend point b is contained within $(\underline{D},\overline{D})$. Then there exists a final solution computation for the problem which can compute the final optimal solution in time $\mathcal{O}(n(\log n + \log m)\log(nm))$.*

With these conditions we get the following result:

Theorem 1. *We consider a class of scheduling problems where an amount m of one discrete resource has to be distributed among n different malleable jobs. Every job j has a convex, discrete function f_j which maps a given resource amount $x \in \mathbb{N}$ to some value $f_j(x) \in \mathbb{R}$. By using the continuous domain extension $\overline{f_j}$ of f_j as defined in Definition 5.2.8, the resource distribution between different jobs can be defined as $x = (x_1,\ldots,x_n) \in \mathbb{R}^n_{>0}$ with x_j being the average assigned resource amount of job j during a time interval $[0,T]$. This class of scheduling problems is partitioned into two sub-classes. The first sub-class are scheduling problems where the objective is to find a resource distribution which minimizes* *the maximum value* $\max_j \overline{f_j}(x_j)$ *of the functions. The second sub-class consists of scheduling problems which have the goal to minimize the sum* $\sum_{j=1}^{n} \overline{f_j}(x_j)$ *of these functions.*

In both cases Algorithm 5.1 computes the optimal schedule (and the optimal resource distribution $x^* = (x_1^*, \dots, x_n^*)$*) in time* $\mathcal{O}(n(\log n + \log m)\log(nm))$ *if the conditions from Definition 5.2.10 (henceforth called the conditions of the theorem) are met.*

If no fast final solution computation is given, an upper bound of the degradation against the optimal solution can be halved in time $\mathcal{O}(n \log m)$ *per halving step.*

The proof of this theorem will be given following a description of the algorithm and some preliminary lemmata.

Optimality conditions The basic idea of the general algorithm is that an optimal solution $x^* = (x_1^*, \dots, x_n^*)$ fulfills $\overline{f_1}(x_1^*) = \dots = \overline{f_n}(x_n^*)$ in the max-case and $\overline{f_1}'(x_1^*) = \dots = \overline{f_n}'(x_n^*)$ in the sum-case (if $\overline{f_j}'$ does not exist see details in Lemma 5.2.14 below). First we have to prove two technical lemmata for the max-case and the sum-case which then will be used to prove that such solutions exist:

Lemma 5.2.11. *Given an optimization problem where* $\max_j \overline{f_j}(x_j)$ *is minimized under the condition* $\sum_{j=1}^{n} x_j \le m$, $x_j \ge 0$ *and all functions* $\overline{f_j}$ *fulfill condition 2 for the max-case. We set* $\underline{D} = \max_j \overline{f_j}(\hat{x}_j)$ *and* $\overline{D} = \max_{j:\, \hat{x}_j \ge m/n} \overline{f_j}(m/n)$*. If* $\overline{D}$ *is defined (which means that we have at least one* j *with* $\hat{x}_j \ge m/n$*) and* $\underline{D} \le \overline{D}$*, then*

$$F^{-1} : [\underline{D}, \overline{D}] \to \mathbb{R}_{>0} \qquad D \mapsto \sum_{j=1}^{n} (\overline{f_j})^{-1}(D)$$

is well-defined, continuous, strictly monotonically decreasing (as well as the single $(\overline{f_j})^{-1}$*) and* $F^{-1}(\overline{D}) \le m$*. If* $\overline{D}$ *is undefined or* $\underline{D} > \overline{D}$ *then* $F^{-1}(\underline{D}) \le m$*.*

Proof. Each function $\overline{f_j}$ is strictly monotonically decreasing and continuous on $(0, \hat{x}_j]$.

Walter [140, page 124] contains a helpful theorem: The function f is continuous and strictly monotonic on an interval I. Then the inverse function on the interval $J = f(I)$ is continuous and strictly monotonic in the same sense.

Hence $(\overline{f_j})^{-1}$ exists on the interval $[\overline{f_j}(\hat{x}_j), \infty)$ and is continuous and strictly monotonically decreasing. As $\underline{D} = \max_j \overline{f_j}(\hat{x}_j)$ and the sum of a finite number of continuous and strictly monotonically decreasing functions is itself continuous and strictly monotonically decreasing, we have that F^{-1} is well-defined, continuous and strictly monotonically decreasing on $[\underline{D}, \infty)$.

The only things left to prove are $F^{-1}(\overline{D}) \leq m$ in case $\overline{D}$ is defined and $\underline{D} \leq \overline{D}$, and $F^{-1}(\underline{D}) \leq m$ in the other cases. Let us first look at the functions with $\hat{x}_j \geq m/n$. Here we have $(\overline{f_j})^{-1}(\overline{D}) \leq m/n$ as $\overline{D} \geq \overline{f_j}(m/n)$ and $\overline{f_j}$ is strictly monotonically decreasing. For the functions with $\hat{x}_j < m/n$ we have two cases: 1. $\overline{D} \geq \overline{f_j}(\hat{x}_j)$, then we have $(\overline{f_j})^{-1}(\overline{D}) \leq \hat{x}_j < m/n$. 2. $\overline{D} < \overline{f_j}(\hat{x}_j)$, then we have $\overline{D} < \underline{D}$. Hence if $\overline{D}$ is defined and $\overline{D} \geq \underline{D}$, we have $F^{-1}(\overline{D}) \leq m$. If $\overline{D}$ is not defined, we have $\hat{x}_j < m/n$ for all j. Thus $(\overline{f_j})^{-1}(\underline{D}) < m/n$ and $F^{-1}(\underline{D}) \leq m$. If $\underline{D} > \overline{D}$ we have two cases: 1. For jobs with $\hat{x}_j < m/n$ it is clear that $(\overline{f_j})^{-1}(\underline{D}) < m/n$. 2. For jobs with $\hat{x}_j \geq m/n$ we have $(\overline{f_j})^{-1}(\underline{D}) < (\overline{f_j})^{-1}(\overline{D}) \leq m/n$ due to the strict monotonicity. And thus altogether $F^{-1}(\underline{D}) \leq m$. □

Lemma 5.2.12. *Given an optimization problem where $\sum_{j=1}^{n} \overline{f_j}(x_j)$ is minimized under the condition $\sum_{j=1}^{n} x_j \leq m$ and all functions $\overline{f_j}$ fulfill condition 2 and 3 for the sum-case. We set $\underline{D} = \min_j \overrightarrow{f_j}(m/n)$ and $\overline{D} = 0$. If $\underline{D} \leq \overline{D}$ we have that*

$$F^{-1} : [\underline{D}, \overline{D}] \to \mathbb{R}_{>0} \qquad D \mapsto \sum_{j=1}^{n} (\overline{f_j}')^{-1}(D)$$

is well-defined, continuous, monotonically increasing (as well as the single $(\overline{f_j}')^{-1}$) and $F^{-1}(\underline{D}) \leq m$. If $\underline{D} > \overline{D}$ we have $F^{-1}(\overline{D}) \leq m$.

Proof. As $\overline{f_j}(x) \xrightarrow[x\to 0]{} \infty$ we also have that $\overline{f_j}'(x) \xrightarrow[x\to 0]{} -\infty$. We know from Lemma 5.2.5 that all $\overrightarrow{f_j}$ are strictly monotonically increasing on $(0, \hat{x}_j]$. With the assumptions $\overleftarrow{f_j}(m) = 0$ when $\hat{x}_j = m$ (for other $\hat{x}_j$ we always have $\overleftarrow{f_j}(\hat{x}_j) \geq 0$) and $(\overline{f_j}')^{-1}(D) = x$ if $\overrightarrow{f_j}(x) \leq D \leq \overleftarrow{f_j}(x)$ for a bend point x the inverse $(\overline{f_j}')^{-1}$ is well-defined on $(-\infty, 0]$. Hence F^{-1} is well-defined on $(-\infty, 0]$.

Let us assume $\underline{D} \leq 0$. We have that all $\overrightarrow{f_j}$ are strictly monotonically increasing (see Lemma 5.2.5) on $(0, \hat{x}_j]$. Let D_1, D_2 be such that $D_1 < D_2 \leq 0$. Then let $x_1 \in (0, \hat{x}_j]$ be the largest x with $\overrightarrow{f_j}(x) \leq D_1$ and let $x_2 \in (0, \hat{x}_j]$ be the largest x with $\overrightarrow{f_j}(x) \leq D_2$ (the left derivatives are continuous from the left). As $D_1 < D_2$, we have that $x_1 \leq x_2$. For all D we have that $(\overline{f_j}')^{-1}(D) = x_D$ where x_D is the largest x with $\overrightarrow{f_j}(x) \leq D$. Hence $(\overline{f_j}')^{-1}$ is monotonically increasing. Thus F^{-1} is monotonically increasing. As $\underline{D} \leq \overrightarrow{f_j}(m/n)$, we have $(\overline{f_j}')^{-1}(\underline{D}) \leq m/n$ and thus $F^{-1}(\underline{D}) \leq m$.

If $\underline{D} > \overline{D}$, we have $\hat{x}_j < m/n$ as $\overrightarrow{f_j}(\hat{x}_j) \leq 0$. Thus $\sum_j \hat{x}_j = F^{-1}(\overline{D}) \leq m$.

The only thing left to prove is the continuity of F^{-1}. As F^{-1} is the sum of a finite number of functions, it is sufficient to prove the continuity of each of these functions. Let $(\overline{f_j}')^{-1}$ be an arbitrary one of these functions. So given an $\varepsilon > 0$

and a $D \in (-\infty, 0]$, we have to show that there exists a $\delta > 0$ such that for all $\tilde{D} \in (-\infty, 0]$ with $|D - \tilde{D}| \leq \delta$ we have $|(\overline{f_j}')^{-1}(D) - (\overline{f_j}')^{-1}(\tilde{D})| \leq \varepsilon$.

Walter [140, page 124] contains a helpful theorem: the function f is continuous and strictly monotonic on an interval I. Then the inverse function on the interval $J = f(I)$ is continuous and strictly monotonic in the same sense.

$\overline{f_j}'$ is continuous and strictly monotonic between 0 and the first bend point b_1, between two neighboring bend points b_k, b_{k+1} and the last bend point b_ℓ and $\hat{x}_j$ (if $\hat{x}_j$ itself is not the last bend point). Then for $D \in (0, \overrightarrow{f_j}(b_1)) \cup (\overleftarrow{f_j}(b_1), \overrightarrow{f_j}(b_2)) \cup \dots \cup (\overleftarrow{f_j}(b_\ell), \overrightarrow{f_j}(\hat{x}_j))$ the continuity is proven. For $D \in [\overrightarrow{f_j}(b_k), \overleftarrow{f_j}(b_k)]$ the function $(\overline{f_j}')^{-1}$ is constant and thus continuous. It remains to show the continuity for $D = \overrightarrow{f_j}(b_k)$ and $D = \overleftarrow{f_j}(b_k)$ (we show it for the first case, the second is analogous). $\overrightarrow{f_j}$ is strictly monotonically increasing, hence $\overrightarrow{f_j}(b_k - \varepsilon) < \overrightarrow{f_j}(b_k)$ (we assume $0 < b_k - \varepsilon$, otherwise we use a fitting smaller ε). If we set $\delta = \min\{\overrightarrow{f_j}(b_k) - \overrightarrow{f_j}(b_k - \varepsilon), \overleftarrow{f_j}(b_k) - \overrightarrow{f_j}(b_k)\}$, then this $\delta > 0$ fulfills the continuity condition. If $\overleftarrow{f_j}(b_k) - \overrightarrow{f_j}(b_k) = 0$, we use $\overleftarrow{f_j}(b_k + \varepsilon) - \overleftarrow{f_j}(b_k)$ instead (for $b_k + \varepsilon \leq \hat{x}_j$). □

With these rather technical lemmata we can now formulate the basic properties for optimal solutions as two lemmata:

Lemma 5.2.13. *Given an optimization problem where* $\max_j \overline{f_j}(x_j)$ *is minimized under the condition* $\sum_{j=1}^{n} x_j \leq m$, $x_j \geq 0$ *and all functions* $\overline{f_j}$ *fulfill condition 2 for the max-case. For a solution* $x^* = (x_1^*, \dots, x_n^*)$ *we define the solution-condition:*

$$\overline{f_1}(x_1^*) = \dots = \overline{f_n}(x_n^*) = D^*$$

Solutions which fulfill the solution-condition are optimal solutions if one of the following two additional conditions are fulfilled: 1. There is at least one job j *with* $\overline{f_j}(x_j^*) = \overline{f_j}(\hat{x}_j)$ *(one job has reached its minimum, solution-condition-1). 2.* $\sum_{j=1}^{n} x_j^* = m$ *and* $x_j^* \leq \hat{x}_j$ *for all* j *(all resources are used, but no job has more resources than needed to reach its minimum, solution-condition-2). We set* $\underline{D} = \max_j \overline{f_j}(\hat{x}_j)$ *and* $\overline{D} = \max_{j:\, \hat{x}_j \geq m/n} \overline{f_j}(m/n)$*. If* $\overline{D}$ *is defined (which means that we have at least one* j *with* $\hat{x}_j \geq m/n$*) and* $\underline{D} \leq \overline{D}$*, then there exists a solution which fulfills the solution-condition and is optimal with* $D^* \in [\underline{D}, \overline{D}]$ *(solution-condition-2 or solution-condition-1). In the other cases such a solution exists for* $D^* = \underline{D}$ *(solution-condition-1). Especially if* $D^* = \underline{D}$ *produces a feasible solution (*$\sum_{j=1}^{n} x_j \leq m$*), this solution is always optimal. If an optimal solution* $x^* = (x_1^*, \dots, x_n^*)$ *fulfills the solution-condition and* $x_j^* < \hat{x}_j$ *for all* j*, then the solution uses all resources (*$\sum_{i=1}^{n} x_i^* = m$*) and is unique.*

Proof. Assume we have a solution $x^* = (x_1^*, \dots, x_n^*)$ which fulfills $\overline{f_1}(x_1^*) = \dots = \overline{f_n}(x_n^*) = D^*$. If there is a job j which has reached its minimum $\overline{f_j}(x_j^*) = \overline{f_j}(\hat{x}_j)$ for this solution, it is clear that no better solution can exist and thus the solution is optimal (solution-condition-1). If all resources are used ($\sum_{i=1}^n x_i^* = m$), we have two cases: either a job has already reached its minimum (then we have an optimal solution, see above) or we have $0 < x_i^* < \hat{x}_i$ for all jobs. Let $y = (y_1, \dots, y_n)$ be a feasible solution different from x^*. Then there exists at least one j with $y_j < x_j^*$ as $\sum_{i=1}^n y_i \leq m$. As $\overline{f_j}$ is convex, we know that it is strictly monotonically decreasing on the interval $(0, \hat{x}_j)$ (see Lemma 5.2.7). Thus it follows from $y_j < x_j^* < \hat{x}_j$ that $f_j(y_j) > f_j(x_j^*) = D^*$ and thus that y has a worse global objective value than x^* (solution-condition-2).

Assume we have an optimal solution $y^* = (y_1^*, \dots, y_n^*)$ with $\sum_{j=1}^n y_j^* < m$ and $y_j^* < \hat{x}_j$ for all j. As $\overline{f_j}$ is strictly monotonically decreasing on the interval $(0, \hat{x}_j)$, there exists an $\varepsilon > 0$ such that by adding δ with $0 < \delta \leq (m - \sum_{j=1}^n y_j^*)/n$ to each y_j^*, we have $\overline{f_j}(y_j^* + \delta) \leq \overline{f_j}(y_j^*) - \varepsilon$ for all j which leads to an improved solution (by ε). Thus an optimal solution uses all available resources if $y_j^* < \hat{x}_j$ holds for all j. As we have seen above that different solutions are always worse in this case, the solution is even unique if $y_j^* < \hat{x}_j$ holds for all j.

It remains to show that an optimal solution which fulfills the solution-condition with $D^* \in [\underline{D}, \overline{D}]$ must always exist. If $\overline{D}$ is undefined or $\underline{D} > \overline{D}$, we know from Lemma 5.2.11 that $F^{-1}(\underline{D}) \leq m$ and thus that a feasible solution with $D^* = \underline{D}$ exists which fulfills the solution-condition. Feasible solutions for $D^* = \underline{D}$ always fulfill solution-condition-1 and thus are optimal solutions. If $\overline{D}$ is defined and $\underline{D} \leq \overline{D}$, we know from Lemma 5.2.11 that $F^{-1}(\overline{D}) \leq m$. If $F^{-1}(\underline{D}) \leq m$, we have still a feasible optimal solution with $D^* = \underline{D}$ (solution-condition-1). If $F^{-1}(\underline{D}) > m$, we know from Lemma 5.2.11 that $F^{-1} : [\underline{D}, \overline{D}] \to \mathbb{R}_{>0}$ is a continuous strictly monotonically decreasing function with $F^{-1}(\overline{D}) \leq m$ and $F^{-1}(\underline{D}) > m$. Hence there exists a $D \in [\underline{D}, \overline{D}]$ with $F^{-1}(D) = m$ (intermediate value theorem, see Walter [140, page 123]). For $D^* = D$ we get an optimal solution which fulfills the solution-condition and uses all resources (solution-condition-2). □

Lemma 5.2.14. *Given an optimization problem where $\sum_{j=1}^n \overline{f_j}(x_j)$ is minimized under the condition $\sum_{j=1}^n x_j \leq m$ and all functions $\overline{f_j}$ fulfill condition 2 and 3 for the sum-case. For a solution $x^* = (x_1^*, \dots, x_n^*)$ we define the solution-condition:*

$$\overline{f_1}'(x_1^*) = \dots = \overline{f_n}'(x_n^*) = D^*$$

As there can be different left ($\overrightarrow{\overline{f_j}}$) and right ($\overleftarrow{\overline{f_j}}$) derivatives for $\overline{f_j}$ at x_j^*, we define the condition $\overline{f_1}'(x_1^*) = \dots = \overline{f_n}'(x_n^*)$ as fulfilled if there exists a D^* such that $\overrightarrow{\overline{f_j}}(x_j^*) \leq D^* \leq \overleftarrow{\overline{f_j}}(x_j^*)\ \forall j \in \{1 \dots n\}$.

A solution which fulfills $\overline{f_1}'(x_1^*) = \cdots = \overline{f_n}'(x_n^*)$ *is an optimal solution if one of the following two additional conditions are fulfilled: 1. We have* $\overline{f_j}(x_j^*) = \overline{f_j}(\hat{x}_j)$ *for all jobs (all jobs have reached their minimum, solution-condition-1). 2.* $\sum_{j=1}^{n} x_j^* = m$ *and* $x_j^* \leq \hat{x}_j$ *for all* j *(all resources are used, but no job has more resources than needed to reach its minimum, solution-condition-2). We set* $\underline{D} = \min_j \overrightarrow{f_j}(m/n)$ *and* $\overline{D} = 0$*. If* $\underline{D} \leq \overline{D}$*, we have that there exists a solution which fulfills solution-condition-1 or solution-condition-2 (and is thus optimal) with* $D^* \in [\underline{D}, \overline{D}]$*. In case of* $\underline{D} > \overline{D}$ *such a solution exists for* $D^* = \overline{D}$ *and fulfills solution-condition-1. Especially if* $D^* = \overline{D}$ *produces a feasible solution (*$\sum_{j=1}^{n} x_j \leq m$*), this solution is always optimal. If the right derivative is negative at* x_j^* *for at least one* $\overline{f_j}$ *for an optimal solution* x^**, then the solution uses all resources (*$\sum_{j=1}^{n} x_j^* = m$*) and is unique.*

Proof. It is clear that a feasible resource distribution for which each job reaches its minimum is optimal (solution-condition-1). For all solutions we can assume $x_j^* \leq \hat{x}_j$. As all $\overrightarrow{f_j}(x) \leq 0$ and are strictly monotonically increasing for $x \in (0, \hat{x}_j]$ (which means that for $D^* = 0$ all jobs have reached their minimum), we can also restrict ourselves to $D^* \leq 0$. Let us now look at a solution $x^* = (x_1^*, \ldots, x_n^*)$ which fulfills $\overline{f_1}'(x_1^*) = \cdots = \overline{f_n}'(x_n^*) = D^*$ and $\sum_{j=1}^{n} x_j^* = m$ and $x_j^* \leq \hat{x}_j$ for all j. Let $y = (y_1, \ldots, y_n)$ be a different feasible solution ($\sum_j y_j \leq m$). We can assume $y_j \leq \hat{x}_j$ as reducing a larger y_j to $\hat{x}_j$ cannot increase the objective value but frees resources. Let us now compare the objective values of the solutions y and x^* by comparing the objective values of each job. We have $\overline{f_j}(x_j^*) - \overline{f_j}(y_j) = \int_{y_j}^{x_j^*} \overline{f_j}'(x)\mathrm{d}x$ as the values of the bend points can be ignored. Let us first look at the jobs j with $x_j^* > y_j$. Here y produces a worse solution than x^* as the $\overline{f_j}$ are strictly monotonically decreasing. $\overline{f_j}'$ is strictly monotonically increasing on $(0, \hat{x}_j]$ and $\overrightarrow{f_j}(x_j^*) \leq D^*$, thus we have

$$\overline{f_j}(x_j^*) - \overline{f_j}(y_j) = \int_{y_j}^{x_j^*} \overline{f_1}'(x)\mathrm{d}x \leq D^* \cdot (x_j^* - y_j)$$

As $\overline{f_j}'$ is strictly monotonically increasing on $(0, \hat{x}_j]$ and $\overleftarrow{f_j}(x_j^*) \geq D^*$, we have analogously for the jobs j with $x_j^* < y_j$:

$$\overline{f_j}(y_j) - \overline{f_j}(x_j^*) = \int_{x_j^*}^{y_j} \overline{f_1}'(x)\mathrm{d}x \geq D^* \cdot (y_j - x_j^*)$$

which is equivalent to $\overline{f_j}(x_j^*) - \overline{f_j}(y_j) \leq D^* \cdot (x_j^* - y_j)$.

For $x_j^* = y_j$ we have $\overline{f_j}(x_j^*) - \overline{f_j}(y_j) = 0$ as well as $D^* \cdot (x_j^* - y_j) = 0$. Hence we have in all cases $\overline{f_j}(x_j^*) - \overline{f_j}(y_j) \leq D^* \cdot (x_j^* - y_j)$ and we can now compare the objective values of y and x^*:

$$\sum_{j=1}^{n} \overline{f_j}(x_j^*) - \sum_{j=1}^{n} \overline{f_j}(y_j) \le D^* \cdot \left(\sum_{j=1}^{n} x_j^* - \sum_{j=1}^{n} y_j \right) \le 0$$

The last inequality follows from the fact that $\sum_{j=1}^{n} x_j^* = m$ and $\sum_{j=1}^{n} y_j \le m$ and $D^* \le 0$. Hence the objective value for y is at least as large as the objective value for x^*. As y is a general feasible solution with no further restrictions, this shows that x^* is an optimal resource allocation (solution-condition-2).

For the existence of the optimal solutions let us first look at the case of $\underline{D} \ge \overline{D}$. $\overline{f_j}'$ is strictly monotonically increasing on $(0,\hat{x}_j]$ and $\overrightarrow{f_j}(\hat{x}_j) \le 0$. Hence for $\overrightarrow{f_j}(m/n) \ge 0$ we have $m/n \ge \hat{x}_j$. If this is the case for all jobs j, it is clear that we can assign each job an amount of $\hat{x}_j$ resources and thus $D^* = \overline{D} = 0$ produces an optimal solution. As long as $\sum_{j=1}^{n} \hat{x}_j \le m$, we have that $D^* = \overline{D} = 0$ produces an optimal solution because $\hat{x}_j = (\overline{f_j}')^{-1}(0)$. Let us now look at the case $\sum_{j=1}^{n} \hat{x}_j > m$ which implies $\underline{D} < \overline{D}$ as the $\overrightarrow{f_j}$ are strictly monotonically increasing. Lemma 5.2.12 shows that $F^{-1}(\underline{D}) \le m$, and thus that there exists a feasible solution which fulfills the solution-condition for $D^* = \underline{D}$ (the solution does not need to be optimal). We know from Lemma 5.2.12 that $F^{-1} : [\underline{D}, \overline{D}] \to \mathbb{R}_{>0}$ is a continuous, monotonically increasing function. With $F^{-1}(\underline{D}) \le m$ and $F^{-1}(\overline{D}) > m$ it follows that there exists a $D \in [\underline{D}, \overline{D}]$ with $F^{-1}(D) = m$ (intermediate value theorem, see Walter [140, page 123]). For $D^* = D$ we get an optimal solution which fulfills the solution-condition and uses all resources (solution-condition-2).

If the right derivative $\overleftarrow{f_j}$ of a function $\overline{f_j}$ is negative at a point x_j, we can decrease the function value of $\overline{f_j}$ further by adding more resources. Hence in an optimal solution $x^* = (x_1^*, \ldots, x_n^*)$, there are no further resources available or all right derivatives $\overleftarrow{f_j}$ are non-negative at x_j^*. We continue with the case of a solution $x^* = (x_1^*, \ldots, x_n^*)$ which has at least one j with $\overleftarrow{f_j}(x_j^*) < 0$. Let us assume a different optimal solution $y^* = (y_1^*, \ldots, y_n^*)$ and that both solutions fulfill $x_i^*, y_i^* \le \hat{x}_i$ for all i. As x^* and y^* are different and x^* uses all available resources we have $x_i^* > y_i^*$ for at least one index i. All functions $\overline{f_i}$ are strictly monotonically decreasing between 0 and their minimum $\hat{x}_i$, thus $\overline{f_i}(x_i^*) < \overline{f_i}(y_i^*)$. As both solutions have the same objective value, there has to be at least one index k with $y_k^* > x_k^*$ and $y_k^* \le \hat{x}_k$ (otherwise y^* can be further improved). As the left derivatives and right derivatives are strictly monotonically increasing on $(0,\hat{x}_j]/(0,\hat{x}_j)$ and x^* fulfills the solution-condition, we have $\overleftarrow{f_i}(y_i^*) < D^* < \overrightarrow{f_k}(y_k^*)$. The left derivatives are continuous from the left and the right derivatives are continuous from the right. Hence there exists a $\delta > 0$ with $\delta \le x_i^* - y_i^*, y_j^* - x_j^*$ for which $\overline{f_i}'$ is continuous on $(y_i^*, y_i^* + \delta)$ and $\overline{f_k}'$ is continuous on $(y_k^* - \delta, y_k^*)$ and an $\varepsilon > 0$ with $\overline{f_i}'(x) + \varepsilon \le \overline{f_k}'(y)$ for all

$x \in (y_i^*, y_i^* + \delta)$ and $y \in (y_k^* - \delta, y_k^*)$. If an additional resource amount of δ is given to job i, we have $\overline{f_i}(y_i^* + \delta) = \overline{f_i}(y_i^*) + \int_{y_i^*}^{y_i^*+\delta} \overline{f_i}'(x)\mathrm{d}x$. If we remove a resource amount of δ from job k, we have $\overline{f_k}(y_k^* - \delta) = \overline{f_k}(y_k^*) - \int_{y_k^*-\delta}^{y_k^*} \overline{f_k}'(y)\mathrm{d}y$. Hence moving a resource amount δ from job k to job i leads to an improvement of the objective function of at least $\delta \cdot \varepsilon$. Thus y^* is not optimal and an optimal solution different from x^* cannot exist. □

Lemma 5.2.14 shows that the optimal solution in the sum-case is always unique and uses all resources as long as not all functions have reached their minimal values.

Basic idea of the core algorithm The pseudo-code of the main algorithm is given in Algorithm 5.1. The basic idea is to use the properties proven in Lemma 5.2.13 and Lemma 5.2.14 that there are optimal solutions for which there exists a common value D^* for all functions (max-case) or derivatives (sum-case). As long as there is no final solution or the properties for the final solution computation are not met, we test new values for D^* (the tested values are denoted as $\tilde{D}$). If the solution that fits the tested $\tilde{D}$ uses more resources than available, we know that the optimal solution must use less resources than the tested solution (monotonicity of the functions and their derivatives). The analogue holds if the solution for the tested $\tilde{D}$ uses less resources than available and the respective solution-condition-1 is not met. Especially we find new lower or upper bounds for D^* in each step. Also the bounds for the resource usage of a job are updated in each step.

Detailed algorithm description The algorithm (Algorithm 5.1) begins with the initialization of the intervals $[\underline{D}, \overline{D}]$ and $[\underline{x_i}, \overline{x_i}]$ (lines 1-7). First the lower $\underline{D}$ (max-case) and upper $\overline{D}$ (sum-case) bounds for D^* are computed according to Lemma 5.2.13 or Lemma 5.2.14. These values are then used to compute the upper $\overline{x_j}$ bounds for the x_j^*. If the sum $\sum_{j=1}^{n} \overline{x_j}$ of these upper bounds is $\leq m$, we have found an optimal solution according to solution-condition-1 in Lemma 5.2.13 or Lemma 5.2.14. Otherwise it is clear that there is no solution which fulfills solution-condition-1, and we have to search for a solution fulfilling solution-condition-2. Hence we have to compute the other boundaries. The upper bound $\overline{D}$ (max-case) is computed according to Lemma 5.2.13 which also guarantees that $\overline{D}$ is defined and $\overline{D} \geq \underline{D}$ as there is no solution which fulfills solution-condition-1. This also yields that an optimal solution can be found for $D^* \in (\underline{D}, \overline{D}]$ (max-case) and that for the remaining algorithm $F^{-1}(\overline{D}) \leq m < F^{-1}(\underline{D})$ holds. The lower bound $\underline{D}$ (sum-case) is computed according to Lemma 5.2.14 which also guarantees that $\underline{D} \leq \overline{D}$ as there is no solution which fulfills solution-condition-1. This

Algorithm 5.1. Schedule Malleable Jobs, in case of a given final solution computation. (max-case): or (sum-case): in front of an instruction mean that this instruction is only executed in the max-case or sum-case respectively. Comments are marked in C-style.

```
Data: $n, m$, functions and inversions $\overline{f_j}, (\overline{f_j})^{-1}$ (max-case) or additionally
      derivatives and inversions $\overline{f_i}', (\overline{f_i}')^{-1}$ (sum-case), the domains of the
      inversions, a final solution computation when for each $x_j^*$ either the value or
      the neighboring bend points are known.
Result: Continuous domain optimal solution $x^* = (x_1^*, \dots, x_n^*)$ which can be placed
        as in Lemma 5.2.9
/* Initialize $\underline{x_i}, \overline{x_i}$ as lower and upper bounds for $x_i^*$ and $\underline{D}, \overline{D}$ as bounds for $D^*$ */
get or compute the minimum $\hat{x}_j$ for each job
(max-case): $\underline{D} := \max_j \overline{f_j}(\hat{x}_j)$,  (sum-case): $\overline{D} := 0$
for $i \in \{1 \dots n\}$ do (max-case): $\overline{x_i} := (\overline{f_i})^{-1}(\underline{D})$ ,   (sum-case): $\overline{x_i} := (\overline{f_i}')^{-1}(\overline{D})$
if $\sum_{i=1}^{n} \overline{x_i} \leq m$ then                         /* solution-condition-1 fulfilled */
    optimal solution $\overline{x}$ return $x^* := (\overline{x_1}, \dots, \overline{x_n})$
(max-case): $\overline{D} := \max_{j:\hat{x}_j \geq m/n} \overline{f_j}(m/n)$ ,  (sum-case): $\underline{D} := \min_j \overrightarrow{f_j}(m/n)$
for $i \in \{1 \dots n\}$ do (max-case): $\underline{x_i} := (\overline{f_i})^{-1}(\overline{D})$ ,  (sum-case): $\underline{x_i} := (\overline{f_i}')^{-1}(\underline{D})$
/* Initialization done                                                       */
if $\sum_{i=1}^{n} \underline{x_i} = m$ then                          /* solution-condition-2 fulfilled */
    optimal solution $\underline{x}$ return $x^* := (\underline{x_1}, \dots, \underline{x_n})$
while $(\underline{D}, \overline{D})$ contains D-values of bend points do
    select $\tilde{D} \in (\underline{D}, \overline{D})$
    for $i \in \{1 \dots n\}$ do (max-case): $\tilde{x_i} := (\overline{f_i})^{-1}(\tilde{D})$ ,  (sum-case): $\tilde{x_i} := (\overline{f_i}')^{-1}(\tilde{D})$
    if $\sum_{i=1}^{n} \tilde{x_i} = m$ then                       /* solution-condition-2 fulfilled */
        optimal solution $\tilde{x}$ return $x^* := (\tilde{x_1}, \dots, \tilde{x_n})$
    if $\sum_{i=1}^{n} \tilde{x_i} < m$ then                 /* new lower bounds for resource usage */
        for $i \in \{1 \dots n\}$ do $\underline{x_i} := \tilde{x_i}$
        (max-case): $\overline{D} := \tilde{D}$ ,  (sum-case): $\underline{D} := \tilde{D}$
    else                                                 /* new upper bounds for resource usage */
        for $i \in \{1 \dots n\}$ do $\overline{x_i} := \tilde{x_i}$ for all $i$
        (max-case): $\underline{D} := \tilde{D}$ ,  (sum-case): $\overline{D} := \tilde{D}$
end
final solution computation $x^*$ using $\underline{x_i}, \overline{x_i}$
return $x^*$
```

also yields that an optimal solution can be found for $D^* \in [\underline{D}, \overline{D})$ (sum-case) and that for the remaining algorithm $F^{-1}(\underline{D}) \leq m < F^{-1}(\overline{D})$ holds.

The initialization in the max-case as well as in the sum-case already finds and returns optimal solutions if solution-condition-1 holds. It remains to search for optimal solutions which fulfill solution-condition-2 for a $D^* \in (\underline{D}, \overline{D}]$ (max-case) or $D^* \in [\underline{D}, \overline{D})$ (sum-case). In particular, we have $\overline{x_j} \leq \hat{x}_j$ for all jobs. By testing the solution $\underline{x} = (\underline{x_1}, \ldots, \underline{x_n})$ on optimality, we make sure that afterwards there is no optimal solution for $D^* = \underline{D}$ or $D^* = \overline{D}$ in either case. Hence optimal solutions can only exist for $D^* \in (\underline{D}, \overline{D})$. This property that no optimal solution can exist on the boundaries of the interval $(\underline{D}, \overline{D})$ is maintained throughout the algorithm by always testing new boundaries on optimality before using them (lines 13-14).

The optimal D^* is then searched by a repeated selection of a new value $\tilde{D} \in (\underline{D}, \overline{D})$ which works as an interval split of the interval $(\underline{D}, \overline{D})$ bounded by the already known lower $\underline{D}$ and upper $\overline{D}$ limits for D^*. Similarly, throughout the algorithm the lower $\underline{x_i}$ and upper $\overline{x_i}$ limits are maintained for the resource amount given to each job i in an optimal solution. In order to update the limits for the resource amounts, we have to compute for each i the amount $\tilde{x}_i$ of resources needed for $\overline{f_i}(\tilde{x}_i) = \tilde{D}$ (max-case) or $\overline{f_i}'(\tilde{x}_i) = \tilde{D}$ (sum-case). The derivatives might contain jumps (at the bend points), but then there is an $\tilde{x}_i$ with $\overrightarrow{f_i}(\tilde{x}_i) \leq \tilde{D} \leq \overleftarrow{f_i}(\tilde{x}_i)$ which can be used as an inverse. Due to the initialization of $\underline{D}$ and $\overline{D}$ we have that the respective inverse image of $\tilde{D} \in (\underline{D}, \overline{D})$ is always contained in $[\underline{x_i}, \overline{x_i}]$.

It remains to be explained how we select the next $\tilde{D}$ from the interval $(\underline{D}, \overline{D})$ (line 11 in the Algorithm 5.1). Graphical examples are given in Figure 5.4. In the standard case there is a fast final solution computation available. Hence it is not important to select $\tilde{D}$ in a way that bounds the degradation against an optimal solution as fast as possible. Instead it is important to fulfill the preconditions for the fast final solution as fast as possible. A fast final solution can compute the optimal resource distribution if no $\overline{f_j}(b)$ (max-case) or $\overrightarrow{f_j}(b)$ or $\overleftarrow{f_j}(b)$ (sum-case) for any bend point b is contained within $(\underline{D}, \overline{D})$. At least we only look at fast final solutions with this property in this work (examples can be found in Section 5.2.5). For the max-case we call the values $\overline{f_j}(b)$ and for the sum-case $\overrightarrow{f_j}(b)$ and $\overleftarrow{f_j}(b)$ **D-values** for the bend point b. Especially there are two D-values for each bend point in the sum-case even if they have the same value.

The selection of $\tilde{D}$ should be done in a way that reduces the number of D-values of bend points within the interval $(\underline{D}, \overline{D})$ as fast as possible (we call these D-values active D-values). A simple way to do this is to just randomly pick a D-value for a random bend point b for a random j from the interval $(\underline{x_j}, \overline{x_j})$ (if there is still a bend point contained in the interval, otherwise a different j is selected).

In order to get a more predictable convergence, the D-value $\tilde{D}$ should be randomly selected from all D-values within the interval $(\underline{D}, \overline{D})$ for all jobs (at most

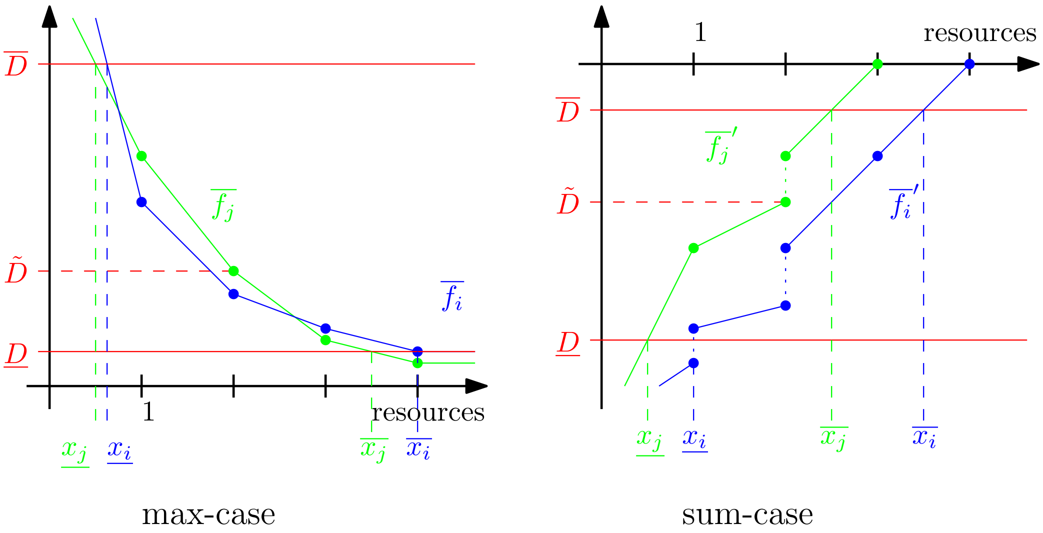


Figure 5.4. The selection of $\tilde{D}$ and its connection to the $\underline{x_j},\overline{x_j}$ for the max-case and the sum-case and two different jobs for each case.

$\mathcal{O}(m \cdot n)$ possibilities) with equal probability. The new $\tilde{D}$ either removes (by becoming the new $\underline{D}$ or $\overline{D}$) all D-values $\geq \tilde{D}$ or all D-values $\leq \tilde{D}$ from the set of active D-values. Due to the random selection of $\tilde{D}$ we only need a logarithmic number ($\mathcal{O}(\log(nm))$) of interval splitting steps to meet the preconditions for the final solution with high probability.

But it is even possible to do this deterministically in a logarithmic number of steps. Within all jobs the function/derivative values of the bend points are already sorted due to their monotonicity. We compute the median D-value D_j of all D-values of job j within $(\underline{D},\overline{D})$ for each job j. Then we sort the D_j according to their values. Each D_j is then weighted with the number of remaining D-values of job j within $(\underline{D},\overline{D})$. If we select the weighted median D_i of the D_j as the new $\tilde{D}$, we remove at least a quarter of all active D-values when $\tilde{D}$ becomes either the new $\underline{D}$ or $\overline{D}$. With a deterministic time sorting algorithm this approach becomes deterministic.

In case the problem does not provide a fast final solution, we have to change the condition for the while-loop (line 10) and the final solution computation (line 22). As the needed approximation guarantee of a solution is only known in the real application, we restrict ourselves to explain a step how an upper bound of the degradation against the optimal solution can be halved. This step then can be repeated as many times as necessary. In both cases we select $\tilde{D} = (\underline{D} + \overline{D})/2$ and execute the main loop once for each halving step. It only remains to show how the final solution is computed in this case.

$\tilde{D} = (\underline{D} + \overline{D})/2$ is a good option for the max-case as D for a feasible solution equals the value of its global objective. So in the max-case without a fast final solution we can do just some interval halving and then take $D^* = \overline{D}$ and $\underline{x} = (\underline{x_1}, \ldots, \underline{x_n})$ as final solution ($D^* = \underline{D}$ uses more than the available resources). This results in a maximal degradation of $\overline{D} - \underline{D}$ against the optimal solution.

For the sum-case without a fast final solution we have to do a little bit more. We split the unused resources in $\underline{x}$ proportional to the differences $\overline{x_j} - \underline{x_j}$ and add them to $\underline{x_j}$, we get a feasible final solution $\dot{x} = (\dot{x}_1, \ldots, \dot{x}_n)$ with $\dot{x}_j = \underline{x_j} + (\overline{x_j} - \underline{x_j}) \cdot (m - \sum_{j=1}^{n} \underline{x_j}) / \sum_{j=1}^{n} (\overline{x_j} - \underline{x_j})$ and $\sum_{j=1}^{n} \dot{x}_j = m$. This results in a maximal degradation of $(\overline{D} - \underline{D}) \cdot (m - \sum_{j=1}^{n} \underline{x_j})$ against the optimal solution.

Altogether Algorithm 5.1 produces an optimal solution as output when a fast final solution computation exists. The optimal solution is then placed as in Lemma 5.2.9. With the algorithm we can now prove Theorem 1:

Proof of Theorem 1. During the initialization (lines 1-7) Algorithm 5.1 checks if there is a feasible solution which fulfills solution-condition-1 from Lemma 5.2.13 or Lemma 5.2.14. If such a solution is found, it is an optimal solution and thus returned. The remaining part of the algorithm has to find a solution which fulfills solution-condition-2 from Lemma 5.2.13 or Lemma 5.2.14. Due to the lemmata such a solution must exist for $D^* \in [\underline{D}, \overline{D}]$ if there is no solution which fulfills solution-condition-1. An optimal solution is found if we have found a D^* with $F^{-1}(D^*) = m$ for the F^{-1} defined in Lemma 5.2.11 (max-case) or Lemma 5.2.12 (sum-case). Due to the monotonicity of F^{-1} as shown in Lemma 5.2.11 or Lemma 5.2.12 it is clear that $F^{-1}(\tilde{D}) < m$ or $F^{-1}(\tilde{D}) > m$ show that the respective $\tilde{D}$ is a new upper or lower limit for D^* (depending on the case). Hence the interval of possible D^* can be reduced by interval splitting. Due to the monotonicity of $(\overline{f_j})^{-1}$ and $(\overline{f_j}')^{-1}$ the interval $[\underline{x_j}, \overline{x_j}]$ of possibilities for the optimal resource assignment x_j^* for job j is also monotonically reduced. The optimal solution is always contained in the respective intervals. If the conditions for the final solution computation are met (see below that this will happen) the algorithm computes the optimal solution for the continuous domain problem.

We also have to show that the discrete solution (corresponding to the optimal continuous domain solution) found by using the placement according to Lemma 5.2.9 is an optimal one. Corresponding here means that the average used resource amount for each job is the same in the discrete solution as in the continuous domain solution. This is done by using the condition 1 of Definition 5.2.10. As the set of schedules in which a job uses only two neighboring resource amounts forms a dominant set of schedules, it is clear that an optimal solution can be found within that set. Also for each feasible schedule in the dominant set there exists a corresponding continuous domain solution. The last part of condition 1 requires that all solutions from the dominant set that correspond to the same continuous domain solution produce the same objective value. This objective value is also the same for the continuous domain solution due to the definition of the $\overline{f_j}$ (Definition 5.2.8). Hence no discrete solution can have a better objective value than an optimal continuous domain solution. Thus all feasible solutions from the dominant set which correspond to an optimal continuous domain solution produce the same value and are optimal. In particular this holds for the discrete solution produced by the placement according to Lemma 5.2.9 from an optimal continuous domain solution.

Now we consider the approximation steps if no final solution computation is available. The running time $\mathcal{O}(n \log m)$ of the halving steps comes from the running time of the body of the main while-loop which is calculated below (with $\tilde{D}$ selection in $\mathcal{O}(1)$). It remains to show that the bounds for the degradation against the optimal solution hold.

In the max-case we have $\overline{D} = \overline{f_j}(\underline{x_j}) \geq \overline{f_j}(x_j^*) = D^* \geq \overline{f_j}(\overline{x_j}) = \underline{D}$ for each job j. As $\overline{D} - D^* \leq \overline{D} - \underline{D}$, we have that $\overline{D} - \underline{D}$ is an upper bound for the degradation against the optimal result. By using $\tilde{D} = (\underline{D} + \overline{D})/2$ as selection for a new $\tilde{D}$ this bound is halved for each iteration of the main loop.

In the sum-case we have to do a bit more to bound the degradation against the optimal solution. We know that $\underline{x_j} \leq x_j^* \leq \overline{x_j}$ for all j. We also know the difference between the global objective values of the two solutions x^* and $\underline{x} = (\underline{x_1}, \ldots, \underline{x_n})$:

$$\sum_{j=1}^{n} \overline{f_j}(x_j^*) - \sum_{j=1}^{n} \overline{f_j}(\underline{x_j}) = \sum_{j=1}^{n} \int_{\underline{x_j}}^{x_j^*} \overline{f_j}'(x) \mathrm{d}x$$

We also know that $\underline{D} \leq \overline{f_j}'(x) \leq \overline{D}$ for $x \in [\underline{x_j}, \overline{x_j}] \supseteq [\underline{x_j}, x_j^*]$. If we use $\underline{x}$ as solution, an amount of $m - \sum_{j=1}^{n} \underline{x_j}$ resources remains unused. If we split the unused resources proportional to the differences $\overline{x_j} - \underline{x_j}$ and add them to $\underline{x_j}$, we get a feasible solution $\dot{x} = (\dot{x}_1, \ldots, \dot{x}_n)$ with $\dot{x}_j = \underline{x_j} + (\overline{x_j} - \underline{x_j}) \cdot (m - \sum_{j=1}^{n} \underline{x_j}) / \sum_{j=1}^{n} (\overline{x_j} - \underline{x_j})$ and $\sum_{j=1}^{n} \dot{x}_j = m$. We have $\underline{x_j} \leq \dot{x}_j \leq \overline{x_j}$ for all j and:

$$\sum_{j=1}^{n} \overline{f_j}(\dot{x}_j) - \sum_{j=1}^{n} \overline{f_j}(\underline{x_j}) = \sum_{j=1}^{n} \int_{\underline{x_j}}^{\dot{x}_j} \overline{f_j}'(x) \mathrm{d}x$$

And hence:

$$\begin{aligned}
\sum_{j=1}^{n}\overline{f_j}(\dot{x}_j)-\sum_{j=1}^{n}\overline{f_j}(x_j^*) \;&=\; \sum_{j=1}^{n}\int_{\underline{x_j}}^{\dot{x}_j}\overline{f_j}'(x)\mathrm{d}x-\sum_{j=1}^{n}\int_{\underline{x_j}}^{x_j^*}\overline{f_j}'(x)\mathrm{d}x\\
\leq\sum_{j=1}^{n}\overline{D}\cdot(\dot{x}_j-\underline{x_j})\;&-\;\sum_{j=1}^{n}\underline{D}\cdot(x_j^*-\underline{x_j})\\
=(\overline{D}-\underline{D})\cdot(m-\sum_{j=1}^{n}\underline{x_j})&
\end{aligned}$$

The value of $m-\sum_{j=1}^{n}\underline{x_j}$ can only decrease by an interval halving step. By using $\tilde{D}=(\underline{D}+\overline{D})/2$ as selection for a new $\tilde{D}$, this bound is at least halved for each iteration of the main loop.

The only thing left to prove is the running time. We go through the algorithm as given in the pseudo-code. The n minima $\hat{x}_j$ can each be computed in time $\mathcal{O}(\log m)$. The initialization of $\underline{D}$ and $\overline{D}$ can be done with n function/derivative evaluations and computing the min or max of n elements. The initialization of the $\underline{x_i}$ and $\overline{x_i}$ then needs another $2n$ inversions. The sum of n elements can also be computed in time $\mathcal{O}(n)$. Altogether the initialization can be done in time $\mathcal{O}(n\log m)$. The body of the main while-loop (lines 10-21) contains a selection of $\tilde{D}$ (which is analyzed below), n inversions, a sum of n summands and $n+1$ assignments. Excluding the selection of a new $\tilde{D}$ the body of the main loop can be done in time $\mathcal{O}(n\log m)$.

Let us now look at the time it takes to evaluate the condition of the while-loop (line 10) in case that a final solution computation exists. Because of condition 3 we can find the first bend point b_1 and compute the number of bend points in $(b_1,\underline{x_j})$ for each job j. Thus we can compute the number of bend points k_j and the numbers of the smallest b_ℓ and largest b_u bend points within $[\underline{x_j},\overline{x_j}]$ in time $\mathcal{O}(1)$. In the max-case we compute $\overline{f_j}(b_\ell)$ and $\overline{f_j}(b_u)$ in $\mathcal{O}(\log m)$ and compare these with $\overline{D}$ and $\underline{D}$ to check if these D-values are within $(\underline{D},\overline{D})$. Due to the strict monotonicity this is sufficient to compute the number of D-values of bend points of job j within $(\underline{D},\overline{D})$. In the sum-case it works analogously except that we have to compute $\overrightarrow{f_j}(b_\ell), \overleftarrow{f_j}(b_\ell), \overrightarrow{f_j}(b_u)$ and $\overleftarrow{f_j}(b_u)$ to compare these with $\overline{D}$ and $\underline{D}$. The bend points between b_ℓ and b_u are counted twice in the sum-case and once in the max-case. Altogether the while-condition can be evaluated in time $\mathcal{O}(n\log m)$.

If we use the deterministic selection of $\tilde{D}$ we have to compute the median D-value D_j for each job j, the number of D-values within $(\underline{D},\overline{D})$ for each job j, sort the D_j and find the weighted median. Finding the number of D-values within $(\underline{D},\overline{D})$ for each job j can be done like in the check of the while-condition.

After b_ℓ and b_u are known and their respective D-values are compared to $\underline{D}$ and $\overline{D}$, the bend point with the median D-value is known (in the sum-case we also know whether to use the left or right derivative) because of the strict monotonicity of the functions/derivatives of the job objective functions. Thus D_j can be computed in $\mathcal{O}(\log m)$. All D_j and their respective weights thus can be computed in time $\mathcal{O}(n \log m)$. The sorting can then be done in $\mathcal{O}(n \log n)$, and the weighted median can be found in time $\mathcal{O}(n)$. For each job j at least half of its active D-values are $\geq D_j$, and at least half of its active D-values are $\leq D_j$. For the weighted median $\tilde{D}$ of the D_j thus at least one quarter of all D-values of all jobs are $\geq \tilde{D}$, and at least one quarter are $\leq \tilde{D}$. Hence for the deterministic selection of $\tilde{D}$ the number of executions of the while-loop (line 10) is in $\mathcal{O}(\log(nm))$ because each time a constant fraction of the active D-values is eliminated. The selected D-value is always eliminated from the active D-values as it lies outside of $(\underline{D}, \overline{D})$.

If an active D-value is randomly selected as $\tilde{D}$, the number of active D-values is reduced at least by one. We can randomly select from all active D-values by computing their number k and then randomly select $\ell \in \{1, \ldots, k\}$ (we assume a rand-function which selects all possible elements with the same probability). Like in the check of the while-condition we compute the number k_j of active D-values of job j within $(\underline{D}, \overline{D})$ in time $\mathcal{O}(\log m)$. By computing the prefix sum $s_j = \sum_{i=1}^{j-1} k_j$ (can be done for all jobs together in time $\mathcal{O}(n)$), it is easy to find the job j with $s_j < \ell \leq s_{j+1}$, and thus the randomly selected ℓ-th D-value. Thus the random selection with equal probability for all D-values can be done in time $\mathcal{O}(n \log m)$.

It remains to show that this kind of random selection of $\tilde{D}$ leads to a logarithmic number of executions of the while-loop with high probability. Let us first look at the set of active D-values X before the next $\tilde{D}$ is selected. If we imagine the set of active D-values to be ordered with minimum D_0 and maximum D_4, there exist D-values D_1, D_2, D_3 within that set that split the set into quartiles. Hence each of the sets $X_1 = \{D \in X | D_0 \leq D \leq D_1\}$, $X_2 = \{D \in X | D_1 \leq D \leq D_2\}$, $X_3 = \{D \in X | D_2 \leq D \leq D_3\}$ and $X_4 = \{D \in X | D_3 \leq D \leq D_4\}$ contains at least one quarter of the active D-values (for example D-values which equal D_1 are in X_1 and X_2). For a split of $(\underline{D}, \overline{D})$ by $\tilde{D}$ all active D-values in $(\underline{D}, \tilde{D}]$ are eliminated if $D^* \in [\tilde{D}, \overline{D})$ and vice versa. We look at two different cases: case 1: $D^* \in (D_1, D_3)$ or case 2: $D^* \in (\underline{D}, D_1] \cup [D_3, \overline{D})$. In case 1 selecting a $\tilde{D} \in X_2 \cup X_3$ leads to the elimination of either all active D-values in X_1 or X_4 (depending on the relative position of $\tilde{D}$ and D^*). In case 2 selecting a $\tilde{D} \in X_2 \cup X_3$ leads to the elimination of either all active D-values in X_1 or X_4 (depending on the location of D^*). Hence we have in both cases with a probability of at least $1/2$ that at least $1/4$ of all active D-values are eliminated in the next step. Let us ignore the improvements due to steps which do not eliminate at least a quarter of active D-values and set r as the worst-case number of steps which eliminate at least a quarter of the active

D-values that are needed to eliminate all D-values. We know that $r \in \mathcal{O}(\log(nm))$. The total number t of steps needed to eliminate all D-values is at most the number of steps needed to get r steps done which occur with a probability of $1/2$. Hence t is a negative binomial distributed random variable (see Hesse [66, page 185,186]) with an expected value of $E(t) = 2r$. We have

$$P(t=k) = \binom{k-1}{r-1}(1/2)^r(1/2)^{k-r} = \binom{k-1}{r-1}(1/2)^k$$

and hence for $k \geq 4r$ we have:

$$\frac{P(t=k+1)}{P(t=k)} = \frac{k \cdot (1/2)}{k-r+1} \leq \frac{k}{2 \cdot (3/4) \cdot k} = \frac{2}{3}$$

Thus the probabilities for higher values of k are exponentially shrinking, and we have for $k \geq 4r$ that $P(t \geq k) \leq 3 \cdot P(t=k)$ and especially for $\ell \in \mathbb{N}$:

$$P(t \geq k+\ell) \leq 3 \cdot \left(\frac{2}{3}\right)^{\ell} \cdot P(t=k)$$

Hence there exist $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $nm \geq n_0$ we have that $P(t \geq c \cdot \log(nm)) \leq (nm)^{-1}$. The final solution has at most the same running time as the upper bound for the while-loop. Hence we have the given upper bound for the running time. □

In one of our articles [116] that use a specialized version of Algorithm 5.1 to the scheduling problem $P|var|C_{max}$ (with convex running time functions) we present a speedup idea that can also be used in order to speed up Algorithm 5.1. In the article we also use interval splitting to find the optimal C_{max}, and for each split we have to compute the inverse of the running time functions which leads to two logarithmic factors in the running time of the scheduling algorithm (one for the interval splitting and one for the inversions that have to be computed for each split). The idea presented in our article [116] is to use only an approximation of the inversion that can be computed in constant time and thus to get rid of one logarithmic factor. Also weighted selection instead of sorting is used to find the right job for the computation of D. A similar approach together with some additional assumptions on the computation of the functions/derivatives and their inverses would probably also work here and reduce the time complexity given in Theorem 1 to $\mathcal{O}(n\log(nm))$.

The most important property the solution described in this section relies on is that the solvable scheduling problems are convex problems. An important case where we do not always have a convex problem is $P|var|\sum \omega_j C_j$. A non-convex example for this problem is given in Section 5.2.5. In this case we cannot use the solution presented in this section because the first condition in Definition 5.2.10 is not met (and hence Theorem 1 is not applicable). In the max-case we only use the inversions of the $\overline{f_i}$, it looks as if the convexity (as required in condition 2 in Definition 5.2.10) of these functions is not important as long as they are monotonically decreasing. But also in this case the convexity (at least of the f_i) is important in order to fulfill the first condition in Definition 5.2.10. Using the average amount of assigned resources as input for a function $\overline{f_i}$ works with Definition 5.2.8. But it is not well-defined in general as there is no need to only use the integer resource amounts that are adjacent to the average. Condition 1 of Definition 5.2.10 at least makes sure that for different possibilities of reaching the same average resource usage the one that only uses the two adjacent integer resource amounts is among the optimal possibilities.

Another reason that justifies the restrictions of condition 2 in Definition 5.2.10 especially the convexity of $\overline{f_j}/f_j$ is that the scheduling problem becomes NP-hard for general functions f_j. This is easy to see by using the observation by Du and Leung [40] already mentioned in Section 3.1.3. They observed that for problems with a variable degree of parallelism one can fix the degree of parallelism k for a job by a speedup function that is $s(p) = 0$ for all $p < k$ and $s(p) = s(k)$ for all $p \geq k$. Together with the already mentioned result from Drozdowski [39] that $P|size_j, pmtn|C_{max}$ is NP-hard we can easily see that the max-case (which includes $P|var|C_{max}$) becomes NP-hard for general functions f_j. We also show the NP-hardness of the problem $P|var|C_{max}$ in one of our articles ([116], joint work with Peter Sanders) without the observation of Du and Leung by using a method similar to the one used by Drozdowski [39]. Here we present a short lemma based on our NP-hardness proof from [116] which even works for the more restricted case of strictly monotonically decreasing running time functions. If condition 1 in Definition 5.2.10 holds, each job with an average resource usage of at least 1 starts at the beginning of the schedule, and thus the running time equals the finishing time C_j.

Lemma 5.2.15. *$P|var|C_{max}$ is NP-hard even for jobs with strictly monotonically increasing speedup functions s_j for resource amounts between* 0 *and their respective maximum $\hat{x}_j$. The running time functions of the jobs in this case are strictly monotonically decreasing between* 0 *and $\hat{x}_j$.*

Proof. We prove this by solving a given instance of `PARTITION` ($a_1, \ldots, a_n \in \mathbb{N}$, Question: Is there a $g : \{1, \ldots, n\} \to \{0, 1\}$ with $\sum_{j:g(j)=1} a_j = \sum_{j:g(j)=0} a_j (= 1/2 \cdot \sum_{j=1}^{n} a_j =: B)$?, see Garey and Johnson [51, page 47] for the NP-completeness of this problem). We call instances of the `PARTITION` problem which allow a split into two sets with equal weight yes-instances, and the instances which do not allow such a split are called no-instances.

We now construct an instance of the scheduling problem with n jobs, each with an amount $w_j = 2a_j$ of work and a speedup function of $s_j(k) = k$ for $k < a_j$ and $s_j(k) = 2a_j$ for $k \geq a_j$. The discrete and the interpolated speedup function (interpolation: $s_j(x) = (1 - x + \lfloor x \rfloor)s_j(\lfloor x \rfloor) + (x - \lfloor x \rfloor)s_j(\lfloor x \rfloor + 1))$ are strictly monotonically increasing on $(0, a_j]$. Hence the running time function is strictly monotonically decreasing on $(0, a_j]$ but not convex for $a_j > 1$. The number of cores is set to $m = B$. The question to decide is whether there is a schedule with makespan 2.

If there is a yes-solution for the `PARTITION` instance, we get a yes-solution for our scheduling problem by running first (between time 0 and 1) the jobs with $g(i) = 1$ and then (between time 1 and 2) the jobs with $g(i) = 0$. Each job j is run on a_j cores in parallel, which leads to a running time of 1 for each job.

For a yes-solution of our scheduling problem it is required that each job always runs with the maximal efficiency. Otherwise the sum of used core time for all jobs would exceed $2B$. Thus each job j has to use exactly a_j cores in parallel during its complete running time. Also there can't be idle cores as $\sum_{j=1}^{n} a_j = 2B$. Hence there must exist a yes-solution of `PARTITION` ($g(j) = 1$ for the jobs starting at time 0, $g(j) = 0$ for the others).

It is remarkable that the scheduling problem built in this proof also violates condition 1 in Definition 5.2.10 as the optimal solution does not use only two neighboring resource amounts throughout the schedule for each job. Instead, each job uses the two resource amounts 0 and a_j which are no neighbors for $a_j > 1$. □

As mentioned during the introduction of the conditions for our solution approach (Definition 5.2.10), we now discuss our running time assumptions. For large m the running time $\mathcal{O}(n(\log n + \log m)\log(nm))$ of our core algorithm is far below the $n \cdot m$ possible objective function values of the n jobs. Hence we have to justify why it makes sense that not all of these values are part of the input of our algorithm. In our article [116] we gave two different reasons. The first one is that scheduling decisions are made frequently based on objective functions which change at most slightly (for example on a machine which frequently handles the same type of jobs). In this case the $n \cdot m$ values are loaded once and are only accessed during each scheduling decision which makes a scheduling decision in time sublinear in nm useful. The second reason is that there might be more compact representations of the objective function which can be evaluated in time $\mathcal{O}(1)$. The most important reason from our perspective is that the most likely application of the core algorithm is scheduling on the system level which means distribution of resources between different applications. The objective functions and their derivatives are then a part of the interface between application and operating system (see Chapter 4 for a discussion about decision distribution and Figure 4.4 for an example interface including such functions). In this case the evaluation of function, derivative and inverse are a call from the operating system to the application which returns the appropriate results for its behavior. Thus the operating system scheduler never gets all $n \cdot m$ values. Altogether there are many cases where the assumption of constant time access to each single value of the objective function makes sense without having to access all possible values.

As we do not assume a special representation for the objective functions we have to define how costly it is to access their properties. Although the running time assumptions in the conditions 3 and 4 in Definition 5.2.10 look quite arbitrary, there are some reasons why they might be fulfilled in many cases. In Definition 5.2.8 we defined $\overline{f_i}(x)$ as the result if job i runs on $\lfloor x \rfloor$ and $\lfloor x \rfloor + 1$ resources for some time such that it runs on x resources on average. Hence if we know $\lfloor x \rfloor$, we have to use the problem-specific interpolation which we assume to be some closed formula which can be evaluated/inverted in constant time. Thus the inversion of a function/derivative consists of finding $\lfloor x \rfloor$ which can be done by interval halving because of the monotonicity of the functions/derivatives and some constant time computation. The evaluation can be assumed as constant time computation in this case. Also computing the number of bend points and finding a special bend point is easy if the bend points are just the integers. If the interpolations are given as closed formula, it might be possible to use some algebra to get a closed formula for D^* once $\lfloor x_i^* \rfloor$ is known for each job i, and then each x_i^* can be computed in constant time. Hence it is plausible that a final solution fulfilling the condition 4 in Definition 5.2.10 can be computed in $\mathcal{O}(n)$. We present some applications with such inversions and final solutions in Section 5.2.5. In all examples the basic idea for the computation of the final solution is that for an optimal solution $x^* = (x_1^*, \ldots, x_n^*)$ we have $\sum_{i=1}^n x_i^* = m$ as long as there are not enough resources to provide each job with its optimal resource amount $\hat{x}_i$.

In the sum-case, the result of this section has an interesting connection to economic science. The common derivative $D^* = \overline{f_1}'(x_1^*) = \cdots = \overline{f_n}'(x_n^*) \leq 0$ can be considered as the *price* of the resource or as marginal profit (as D^* is negative the formulation might be more fitting for $-D^*$). The payment in this case is done in objective function value. If a job i has already a resource amount of x_i then adding a small amount δ of additional resources changes its contribution to the sum and thus to the objective function by an amount of $\approx \delta \cdot \overline{f_i}'(x_i)$ (as long as $\overline{f_i}'$ does not change too much). This is an improvement as $\overline{f_i}'(x_i) \leq 0$ and we want to minimize the sum. Hence if there is a possibility to get more resources in exchange to a penalty to the objective function, we are willing to pay a price up to $-\delta \cdot \overline{f_i}'(x_i)$ for an amount δ of the resource for job i. Thus the upper limit of the per-unit price we are willing to spend for job i is $-\overline{f_i}'(x_i)$. Another formulation of Lemma 5.2.14 is then: if the upper limit of the per-unit resource price we are willing to spend is the same for all jobs, then we have an optimal allocation of the resources.

Let us now take a look at the four main properties of a scheduling problem as defined at the beginning of Section 1.2. The decision space in this case is for each resource unit when it is used and for which job. This is simplified by the fact that all resource units are equal and by condition 1 of Theorem 1 to find a possibly non-integer average distribution of the m resource units among the n jobs. The most important properties of the problem are m and the job-specific functions $\overline{f_j}$. The objectives are $\max_j \overline{f_j}(x_j)$ or $\sum_{j=1}^{n} \overline{f_j}(x_j)$. For the information one can assume that all information is given to the scheduler. Alternatively and probably more realistically the functions $\overline{f_j}$ are aggregations of more complex job-internal properties.

The part $\overline{f_j}(x) \xrightarrow[x \to 0]{} \infty$ in condition 2 of Theorem 1 can probably be omitted, but this leads to a lot of special cases in the algorithm and in the proofs. The dominant set in condition 1 of Theorem 1 can also be more restricted as long as all solutions produced by the core algorithm and the subsequent placement are still contained within that set.

An algorithm slightly related to our Algorithm 5.1 was given by Ludwig and Tiwari [98] for an approximation of $P|any|C_{max}$. Their algorithm only deals with integer/discrete resource assignments and balances the core-time demand of all jobs with the running time of the largest instead of optimizing the running time of all jobs directly (in case we assume that the functions optimized in our max-case are running times). The common thing is, that they also use interval splitting in order to estimate a globally optimal scheduling property. In their case the algorithm is only a pre-computation for an approximation algorithm for $P|size_j|C_{max}$, and they optimize a lower bound for this approximation.

Even though Algorithm 5.1 is a fast algorithm, it can be further sped up by parallelization. The parallelization is presented in Section 5.2.4.

5.2.4 Parallelization

The parallelization of our malleable scheduling approach was initially developed in joint work with Peter Sanders [116] for a specialized version ($P|var|C_{max}$). Here we give a more detailed version that also fits our generalized approach. We use the notation from the previous Section 5.2.3. Although computing a schedule for parallel jobs in parallel itself seems to be quite natural, we have found little work about parallel schedule computation (see Section 3.1.7), especially nothing about scheduling malleable jobs in parallel.

In order to show that it is possible to get polylogarithmic time scheduling algorithms for the kind of scheduling problems treated in Section 5.2.3, we will first show how to parallelize the placement algorithm from Section 5.2.2 and then

Algorithm 5.1. After that we discuss different possibilities for the selection of $\tilde{D}$. Unlike in the other sections we use p as the number of processors/cores for the schedule computation in this section as m already describes the amount of available resources.

The parallelization will work with the PRAM (EREW) model as well as with the network model when the connection graph is complete (see Section 2.6.2 for details on these models). We assume that the job descriptions are distributed in a way such that each core has to take care of the same number of jobs. Also each core gets a consecutive set of jobs (when the jobs are numbered) such that the numbers of these jobs are larger than all job numbers on cores with a smaller core number (when the cores are numbered). When $p \geq n$, we have polylogarithmic execution time. In this case the algorithm only uses n cores, each handling one job. We begin with the introduction of the collective operations used for the algorithm: broadcast, reduction and prefix sum. These collective operations are commonly known and are only introduced for completeness and notation purposes.

If we want to distribute a value initially known to only one core to all other cores involved in the computation, we have to perform a **broadcast**. The first step is always that the knowing core sends the value to another core (network model) or stores it for another core (PRAM model). After that the set of participating cores is halved, and we repeat the first step until all cores know the information. The whole broadcast takes time in $\mathcal{O}(\log p)$. If we only look at the connections through which a core gets the broadcasted value, the cores form a tree with the initially knowing core as root (an example is given in Figure 5.5). Here we only have to deal with the case that a value known to core 0 must be broadcasted to the other cores.

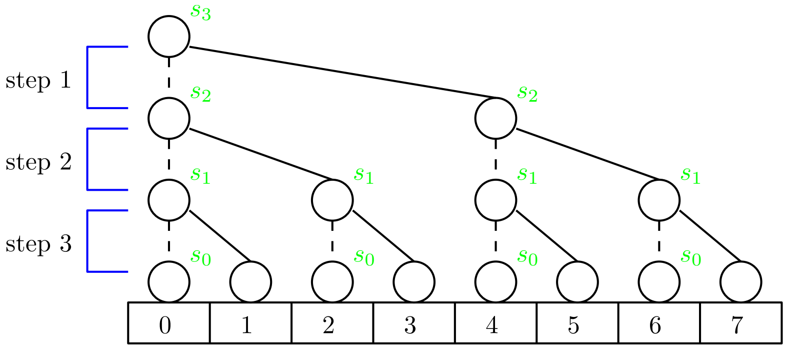


Figure 5.5. A sketch of the broadcast and reduce tree for $p = 8$. The dashed lines are no connections but indicate that the core keeps the value. The green values are the temporary variables used for the computation of the prefix sum which uses the same tree structure.

If we want to compute a sum or another associative and commutative (simple) operation (for example max or min) and each core has one input value, we have to perform a **reduction**. The idea here is to perform a broadcast backwards. If a core gets a value from another core during a certain time step in a broadcast, it sends its value to that core at the same time step in reversed order. The other core then performs the operation on the received value and its own value and sends the result whenever it is its turn. In the end we have a result for all input values of all cores at core 0. As we communicate like the broadcast and have to perform one operation per communication, the running time of the reduction is the same in the $\mathcal{O}$ notation: $\mathcal{O}(\log p)$.

If the cores are ordered and each core has one value and we want to compute for each core the sum of the values of the cores ahead of itself in the ordering, then we have to compute a **prefix sum**. Let for example each core have one value x_i for $i \in \{0,\dots,p-1\}$, then $\sum_{i=0}^{k-1} x_i$ is the prefix sum for core k. We assume that the ordering fits the tree construction as in Figure 5.5, and thus the cores ahead of one core in the ordering are left of it in the tree construction. The computation of the prefix sum is similar to performing a reduction followed by a broadcast. During the reduction phase each core sends one value and receives between 0 and $\log p$ values depending on the position in the tree. During the broadcast phase each core receives one value and sends the same number of values which it has received during the reduction phase. Each core stores its own value in s_0. The first received value r is added and the result is stored in $s_1 = r + s_0$. The same is done for the other received values which produces the stored values $s_2, \dots$ until the core sends its final (largest) sum to its parent node in the reduction tree. The needed storage amount is in $\mathcal{O}(\log p)$ per core. Core 0 has no upper core in the tree and thus does no upward communication and subsequently has a prefix sum of 0. Let a and b be two cores connected by an edge in the broadcast/reduction tree such that a is higher up in the tree than b. Let s_k and s_{k+1} be the sums of a before and after receiving the value from b and r be the value that a receives from its parent node during the broadcast phase. r is the prefix sum of a. In order to compute the prefix sum of b, one has to add the values of a and all cores between a and b to r. This is exactly $r + s_k$ which is sent to b during the broadcast phase. Altogether the prefix sum operation can be done in time $\mathcal{O}(\log p)$. More detailed descriptions for reduction and prefix sum (formulated for bit strings) can be found in the textbook of Leighton [91].

Parallel placement algorithm Let us now look at the parallelization of the computation of discrete schedules for malleable jobs when a non-integer solution for the resource distribution among these jobs is given. The sequential problem is dealt with in Section 5.2.2 and its results are summarized in Lemma 5.2.9. Let us assume we have given a non-integer solution $x^* = (x_1^*, \dots, x_n^*)$ for our scheduling problem. We then have to determine for each job when it starts and stops to

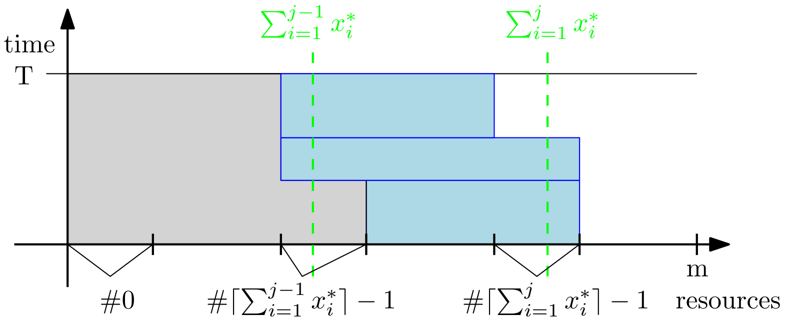

$\sum_{i=1}^{j-1} x_i^*$
$\sum_{i=1}^{j} x_i^*$
time
T
m
resources
#0
$\#\lceil \sum_{i=1}^{j-1} x_i^* \rceil - 1$
$\#\lceil \sum_{i=1}^{j} x_i^* \rceil - 1$

Figure 5.6. Parallel placement of malleable jobs. The three triangles are each placed below one resource unit. Below these triangles the number of the respective resource unit is denoted.

use which resource. We assume the discrete resources are numbered from 0 to $m-1$ and the time interval within which the jobs are executed is $[0,T]$. If we place the jobs one by one as in Section 5.2.2, job j starts at time 0 on resources $\lceil \sum_{i=1}^{j-1} x_i^* \rceil, \dots, \lceil \sum_{i=1}^{j} x_i^* \rceil - 1$. If $\lceil \sum_{i=1}^{j} x_i^* \rceil - 1 < \lceil \sum_{i=1}^{j-1} x_i^* \rceil$, job j gets no resources at time 0 and $x_j^* < 1$. At time $T \cdot (\sum_{i=1}^{j-1} x_i^* - \lfloor \sum_{i=1}^{j-1} x_i^* \rfloor)$ resource $\lfloor \sum_{i=1}^{j-1} x_i^* \rfloor$ is given to job j (if $\sum_{i=1}^{j-1} x_i^* - \lfloor \sum_{i=1}^{j-1} x_i^* \rfloor = 0$ nothing has to be done). At time $T \cdot (\sum_{i=1}^{j} x_i^* - \lfloor \sum_{i=1}^{j} x_i^* \rfloor)$ resource $\lfloor \sum_{i=1}^{j} x_i^* \rfloor$ is removed from job j (if $\sum_{i=1}^{j} x_i^* - \lfloor \sum_{i=1}^{j} x_i^* \rfloor = 0$ nothing has to be done). The job then finishes at time T with resources $\lfloor \sum_{i=1}^{j-1} x_i^* \rfloor, \dots, \lfloor \sum_{i=1}^{j} x_i^* \rfloor - 1$. If $\lfloor \sum_{i=1}^{j} x_i^* \rfloor - 1 < \lfloor \sum_{i=1}^{j-1} x_i^* \rfloor$, then job j has no resources left at time T and $x_j^* < 1$. In order to compute all these times and resource indices in $\mathcal{O}(1)$ on the core which takes care of the job, the core only has to know $\sum_{i=1}^{j-1} x_i^*$ and T and x_j^*. The placement of a single job is depicted in Figure 5.6. We will see below that each core knows T and the x_j^* for its jobs from the parallel computation of x^*. It remains to show how each core can compute $\sum_{i=1}^{j-1} x_i^*$ for its jobs j. Let $j_1 \dots j_k$ be the jobs on a core c. Then c computes $s = \sum_{i=1}^{k} x_{j_i}^*$ in time $\mathcal{O}(k) = \mathcal{O}(n/p)$ as each core gets the same amount of jobs (± 1). c then takes part in the computation of the prefix sum with s and receives its prefix sum $r = \sum_{i=1}^{j_1-1} x_i^*$ in time $\mathcal{O}(\log p)$. Then c can compute $\sum_{i=1}^{j_1-1} x_i^* = r$ and $\sum_{i=1}^{j_\ell-1} x_i^* = x_{j_{\ell-1}}^* + \sum_{i=1}^{j_{\ell-1}-1} x_i^*$ for all of its jobs j_ℓ, $\ell \in \{1 \dots k\}$ in time $\mathcal{O}(k) = \mathcal{O}(n/p)$. Hence the discretization of a non-integer solution x^* can be computed in time $\mathcal{O}(n/p + \log p)$ or even in time $\mathcal{O}(\log n)$ for $p \geq n$.

Parallelization of the main algorithm First we take a look at the parallelization of the initialization of Algorithm 5.1. Due to condition 3, the function values of $\overline{f_j}$ and $(\overline{f_j})^{-1}$ (max-case) or $\overline{f_j}'$, $(\overline{f_j}')^{-1}$ and $\overrightarrow{f_j}$, $\overleftarrow{f_j}$ (sum-case) can be computed in time $\mathcal{O}(\log m)$. First each core computes $\hat{x}_j$ for all of its jobs j. Then $\overline{f_j}(\hat{x}_j)$

and with one reduction $\underline{D} = \max_j \overline{f_j}(\hat{x}_j)$ are computed in the max-case and $\underline{D}$ is broadcasted. $\overline{D} = 0$ in the sum-case. After that each core computes $\overline{x_j} = (\overline{f_j})^{-1}(\underline{D})$ (max-case) or $\overline{x_j} = (\overline{f_j'})^{-1}(\overline{D})$ (sum-case) for each of its jobs j. With a further reduction $\sum_{j=1}^{n} \overline{x_j}$ is computed on core 0 which decides if $\sum_{j=1}^{n} \overline{x_j} \leq m$. If yes, then core 0 knows that solution-condition-1 is fulfilled and initiates the placement algorithm and also computes and broadcasts T if needed (the computation and broadcast of T is always assumed when a solution is computed and T is still unknown). If no, then the computation of the other boundaries is initiated. If $\hat{x}_j \geq m/n$, also $\overline{f_j}(m/n)$ (max-case) or $\overrightarrow{f_j}(m/n)$ (sum-case) are computed. After that the core computes the local maximum (max-case) and minimum (sum-case) for all of its jobs j. With these preparations all cores take part in one reduction to compute $\overline{D} = \max_{j:\hat{x}_j \geq m/n} \overline{f_j}(m/n)$ (max-case) or $\underline{D} = \min_j \overrightarrow{f_j}(m/n)$. $\overline{D}$ (max-case) and $\underline{D}$ (sum-case) are then broadcasted to all cores. Each core computes $\underline{x_j} = (\overline{f_j})^{-1}(\overline{D})$ (max-case) or $\underline{x_i} = (\overline{f_i'})^{-1}(\underline{D})$ (sum-case) for its jobs. After the initialization we also check $\sum_{j=1}^{n} \underline{x_j} = m$ which can be computed with one reduction.

Hence the initialization/part before the main loop can be done with a constant number of local computations with running time $\mathcal{O}(\log m)$ for each job and three or four reductions and one to three broadcasts. Thus the parallel running time for the initialization is $\mathcal{O}(n/p \cdot \log m + \log p)$ or even $\mathcal{O}(\log m + \log n)$ in case of $p \geq n$.

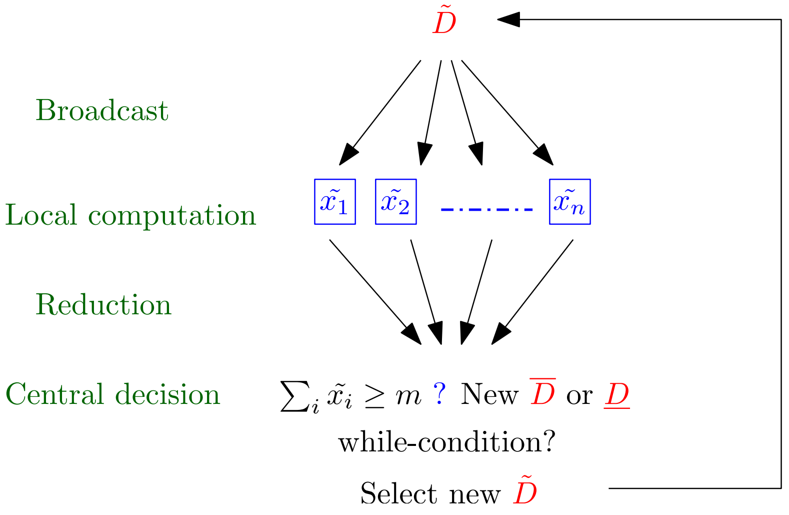


Figure 5.7. A sketch of the parallel execution of the while-loop from Algorithm 5.1.

The main part of the work within Algorithm 5.1 is done in the main while-loop. The while-loop is parallelized by parallelizing the body of the loop. The different possibilities to parallelize the selection of $\tilde{D}$ are described below, but we can assume that core 0 knows the current $\tilde{D}$, and it is distributed by a broadcast to all other cores. The computation of the resource demands $\tilde{x_j} = (\overline{f_j})^{-1}(\tilde{D})$ or $\tilde{x_j} = (\overline{f_j}')^{-1}(\tilde{D})$ for $\tilde{D}$ is done like the initialization of the $\underline{x_j}$ and $\overline{x_j}$ by local computations. Together with the broadcast of $\tilde{D}$ this can be done in time $\mathcal{O}(\log p + n/p \cdot \log m)$. Afterwards $\sum_{i=1}^{n} \tilde{x_i}$ is computed by local summations and a reduction afterwards which can be done in time $\mathcal{O}(\log p + n/p)$. With this result core 0 decides if there is a new $\underline{D}$, $\overline{D}$ or a final solution (time in $\mathcal{O}(1)$). Core 0 broadcasts its result (in time $\mathcal{O}(\log p)$) such that the other cores can set $\underline{x_j}$ or $\overline{x_j}$ (in time $\mathcal{O}(n/p)$) or finish their computation and proceed with the placement.

The while-condition (line 10 in Algorithm 5.1) can be checked similarly as in the proof of Theorem 1. We describe how to compute the number of active D-values because we also need this number later. For the while-condition only the information if this number is larger than zero is needed. We can compute the number of bend points k_j and the numbers of the smallest b_ℓ and largest b_u bend points within $[\underline{x_j},\overline{x_j}]$ in time $\mathcal{O}(1)$. In the max-case we compute $\overline{f_j}(b_\ell)$ and $\overline{f_j}(b_u)$ (in $\mathcal{O}(\log m)$) and compare these with $\overline{D}$ and $\underline{D}$. Due to the strict monotonicity this is sufficient to compute the number of D-values of bend points of job j within $(\underline{D},\overline{D})$. In the sum-case it works analogously except that we have to compute $\overrightarrow{f_j}(b_\ell), \overleftarrow{f_j}(b_\ell), \overrightarrow{f_j}(b_u)$ and $\overleftarrow{f_j}(b_u)$ to compare these with $\overline{D}$ and $\underline{D}$. The bend points between b_ℓ and b_u are counted twice in the sum-case and once in the max-case. Hence the number of active D-values of one job can be computed in time $\mathcal{O}(\log m)$ by its core. As each core has to do this computation for all of its jobs and as we also have to perform a reduction afterwards to get the sum over all jobs, the checking of the while-condition can be done in $\mathcal{O}(\log p + n/p \cdot \log m)$ (or $\mathcal{O}(\log p + \log m)$ in case of $p \geq n$). Altogether one iteration of the while-loop can be done in time $\mathcal{O}(\log p + n/p \cdot \log m)$ plus the time used for the selection of a new D. In case $p \geq n$ the while-loop only takes time in $\mathcal{O}(\log p + \log m)$ plus the time used for the selection of a new D. A graphical illustration of the parallelization of the while-loop is given in Figure 5.7.

The only part of the algorithm whose parallelization remains to be shown is the selection of $\tilde{D}$.

When no final solution computation exists, we use interval halving. Core 0 knows $\underline{D}$ and $\overline{D}$ and thus can just compute $\tilde{D} = (\underline{D} + \overline{D})/2$ and broadcast the result. The computation in this case only takes constant time and the broadcast of the new $\tilde{D}$ is already included in our description of the while-loop above.

The first option in case of the existence of a final solution computation is the selection of a random D-value with equal probability for all active D-values. We do this by summing up the total number k of active D-values like in the check of the while-condition; core 0 then picks a random number r between 1 and k with equal probability. As part of the computation of k we have computed the number of active D-values k_j for each job j. By computing the prefix sum $s_j = \sum_{i=1}^{j-1} k_i$ (can be done in time $\mathcal{O}(\log p + n/p)$)and broadcasting r, each core can check if it has the job j with $s_j < r \leq s_{j+1}$ and thus the randomly selected r-th D-value. The $r - s_j$-th D-value of a job can be found by computing the number of active D-values of bend point b_ℓ, and with this information one can compute in $\mathcal{O}(1)$ the bend point of the $r - s_j$-th D-value (and if the left or right derivative is needed in the sum-case). Then the responsible core computes the D-value $\tilde{D}$ of the bend point in $\mathcal{O}(\log m)$. Core 0 gets $\tilde{D}$ either by a direct send or a special reduction and can afterwards use it as normal. Thus the random selection with equal probability for all D-values can be done in time $\mathcal{O}(\log p + n/p \cdot \log m)$ (or $\mathcal{O}(\log p + \log m)$ in case of $p \geq n$). With this random selection we end up with a feasible input for the fast final solution with high probability after $\mathcal{O}(\log(nm))$ iterations of the while-loop (look into the proof of Theorem 1 for more details regarding the probability).

It is also possible to perform the deterministic selection of $\tilde{D}$ in parallel similarly to the deterministic sequential selection. For the deterministic selection of $\tilde{D}$ we restrict ourselves to the case of $p \geq n$ and an EREW PRAM. Each core c ($c = j$) computes the median D-value D_j for its job j. Then the values D_j are sorted by all cores using Cole's merge sort [33]. Also each D_j gets as weight k_j the number of active D-values of job j. Afterwards the sum s_j of all weights for all $D_i < D_j$ (in the sorted order) is computed via a prefix sum for each D_j. Then each core checks if $s_j < k/2 \leq s_j + k_j$ for its D_j. The D_j for which this is the case (the weighted median of the D_j) becomes $\tilde{D}$ and is sent directly or via reduction to core 0. The parallel deterministic selection consists of a local computation of the median for each job ($\mathcal{O}(\log m)$), Cole's merge sort ($\mathcal{O}(\log n)$), a prefix sum ($\mathcal{O}(\log n)$), a reduction and a broadcast ($\mathcal{O}(\log n)$, computation of k), a check on each core ($\mathcal{O}(1)$) and possibly another reduction ($\mathcal{O}(\log n)$). Altogether the deterministic selection can be done in time $\mathcal{O}(\log n + \log m)$.

With the deterministic selection the number of steps of the while-loop is in $\mathcal{O}(\log nm)$, for the random selection of a D-value this is the case with high probability. Altogether we get this result:

Theorem 2. *The scheduling algorithm for the class of scheduling problems described in Theorem 1 can be parallelized on p cores on an EREW PRAM or a machine fitting the fully connected network model. As an additional condition the final solution must also allow a parallel computation in time* $\mathcal{O}((n/p \cdot \log m + \log p) \cdot \log(nm))$. *Then the running time of the parallel algorithm is in* $\mathcal{O}((n/p \cdot \log m + \log p) \cdot \log(nm))$ *with high probability. In case of* $p \geq n$ *and when running on an EREW PRAM, there is even a deterministic parallel algorithm with running time in* $\mathcal{O}((\log m + \log n) \cdot \log(nm))$.

The parallelization adds another parallel scheduling algorithm to the small set of existing parallel scheduling algorithms (see Section 3.1.7) and is, as far as we know, the first parallel scheduling algorithm for parallel jobs. The parallelization even falls into Nick's Class as it has polylogarithmic running time (see Section 1.2.2 for a short introduction into parallel complexity).

The parallelization is also a solution for the problem of overloaded decision makers in scheduling (see Chapter 4). With a parallel schedule computation scheduling decisions can be made much faster. In case of $p = n$ and when running on an EREW PRAM, we even have the optimal parallel speedup compared to the sequential solution (disregarding constant factors).

The parallelized approach where one core takes care of one job makes the already mentioned economic interpretation for the sum-case more fitting. The actors (the cores each acting on behalf of its job) try to find a resource distribution where the smallest marginal cost for losing resources for an actor is at least the same as the biggest marginal gain of getting resources for all actors. If such a distribution is found, this is an optimal solution.

5.2.5 Applications and Examples

In this Section we show how to use the core algorithm and its parallelization for more concrete scheduling problems. The first three examples (minimize the maximal running time, minimize the used energy and minimize the optimization gap) are applications of the results from Section 5.2.2 and Section 5.2.3 (and also Section 5.2.4). Together with the examples the necessary techniques are presented to make them fit the approach. These techniques might also be helpful for other problems.

We also present an example that looks similar but does not lead to a convex problem and thus does not allow our core algorithm to be used.

Minimize the Maximal Running Time

In this example we are given n jobs which are malleable and have concave speedup functions (see Section 2.4 for a definition and discussion of speedups). The goal is to distribute m cores among these jobs such that the finishing time of the last finishing job is minimized. The formal problem description is $P|var|C_{max}$ with concave speedup functions. As the speedup functions are only defined for positive integer numbers of cores we use a concavity definition similar to the convexity definition from Section 5.2.2. The problem can easily be motivated. Having different malleable jobs it is quite natural to ask which distribution of cores leads to the minimal finishing time of the set of all jobs. It is also quite natural to assume that the efficiency decreases as more cores are added to a job, thus the speedup increase for an additional core is monotonously decreasing for each additional core. Hence concave speedup functions are quite natural.

The problem was investigated in the already mentioned article from Błażewicz et al. [24] from which the here presented idea of the discretization of a continuous domain result is taken. Together with a previous article by the same authors (Błażewicz et al. [26]) they solved the problem in time $\mathcal{O}(n \max\{m, n \log^2 m\})$. In joint work with Peter Sanders [116] we were able to improve this result to $\mathcal{O}(n + \min\{m,n\} \log m)$ if the schedule is computed on one core and to even $\mathcal{O}(n/m + \log^2 m)$ if the schedule is computed on the m cores that are to be scheduled. This example is clearly based on our article [116] although we show here that the problem fulfills the conditions of our general solution instead of solving the problem directly.

We will now show that this problem can also be solved by our approach summarized in Theorem 1 and Theorem 2. In order to do this, we have to show that the problem meets the four conditions in Theorem 1 and the additional one in Theorem 2. The problem obviously fits the max-case with the finishing times C_j of the jobs.

The first thing we have to do in order to show that the problem fits our global approach is to take a closer look at the properties of the finishing time functions $C_j(k)$ for an integer amount of cores k and at $\overline{C_j}(x)$ for an average possibly non-integer amount of cores x.

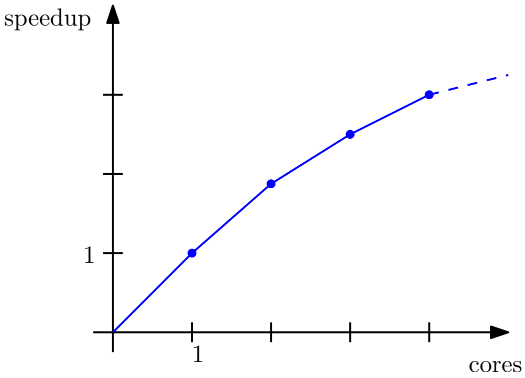


Figure 5.8. The original discrete speedup function (dots) is made continuous by linear interpolation.

We start by making the speedup function s continuous. Let w be the amount of work of the job that is done during time T on a single core. Then running on $k \in \mathbb{N}_0$ cores does $w \cdot s(k)$ work during time T ($s(0) = 0$) or an amount w of work can be done in time $T/s(k)$. If a job runs on k cores for a time αT and on $k+1$ cores for a time βT with $\alpha + \beta = 1$ and speedups $s(k)$ and $s(k+1)$, the speedup for this configuration during the time T is $\alpha s(k) + \beta s(k+1)$. This is the case as the work done during time T is $w\cdot(\alpha s(k)+\beta s(k+1))$, and parallel work amount during a time T divided by the sequential work amount is always the speedup. Hence if running on $x\in\mathbb{R}_{>0}$ cores for time T means running on $\lfloor x\rfloor$ cores for time $(1-x+\lfloor x\rfloor)T$ and on $\lfloor x\rfloor+1$ cores for time $(x-\lfloor x\rfloor)T$, then the speedup $\overline{s}$ for non-integer core amounts is just the linear interpolation between its neighboring integers. An example of a speedup function made continuous is drawn in Figure 5.8. If we name the sequential running time/work of job j as w_j, we get $C_j(k)=w_j/s_j(k)$ and $\overline{C_j}(x)=w_j/\overline{s_j}(x)$ and also $s_j(1)=\overline{s_j}(1)=1$. As $\overline{s_j}$ is the linear interpolation of a discrete concave function, we know that it is concave itself. Hence we know from Lemma 5.2.3 that $\overline{C_j}$ is convex and continuous and even strictly convex on the interval where $\overline{s_j}$ is strictly monotonically increasing (the strictness is not needed in the max-case). As $\overline{s_j}(0)=0$ we have that $\overline{C_j}(x)\xrightarrow[x\to 0]{}\infty$. The number of bend points is m. Between $k,k+1\in\mathbb{N}$ the functions $\overline{s_j}$ and $\overline{C_j}$ are always given through a closed algebraic form only dependent on $s_j(k),s_j(k+1)$ and w_j. With this we have already fulfilled **condition 2** from Theorem 1.

Now we check if this problem fulfills **condition 1** from Theorem 1. We assume we have an optimal solution with $\max_i C_i=T$. Then we look at a single job j. As $C_j\leq T$, there exist $k_i\in\mathbb{N}_0$ and $\alpha_i\in(0,1]$ for $i\in\{1,\dots,\ell\}$ such that j runs on k_i cores for a time $\alpha_i T$ during the time interval $[0,T]$ with $\sum_{i=1}^{\ell}\alpha_i=1$ and $k_1<k_2<\cdots<k_\ell$. If $k_\ell\leq k_1+1$, then job j fits into condition 1. Hence we assume $k_\ell>k_1+1$. Let $\hat{k_j}$ be the minimal amount of cores for which job j reaches its highest speedup. Then we can also assume that $k_\ell\leq\hat{k_j}$ as otherwise we can replace the running time on k_ℓ cores by an appropriate combination of running time on 0 and $\hat{k_j}$ cores such that the amount of work done is kept the same and the average number of used cores is decreased. Let $\omega_1=\alpha_1 T s_j(k_1)$ be the work done during running on k_1 cores and $\omega_\ell=\alpha_\ell T s_j(k_\ell)$ the work done during running on k_ℓ cores. We have $s_j(k_1)<(\omega_1+\omega_\ell)/(T(\alpha_1+\alpha_\ell))<s_j(k_\ell)$ as s_j is strictly monotonically increasing between 0 and $\hat{k_j}$. As $\overline{s_j}$ is continuous and strictly monotonically increasing, there exists an $x\in\mathbb{R}_{>0}$ with $k_1<x<k_\ell$ and $\overline{s_j}(x)=(\omega_1+\omega_\ell)/(T(\alpha_1+\alpha_\ell))$. Due to the concavity of $\overline{s_j}$ we also have $x\leq(\alpha_1 k_1+\alpha_\ell k_\ell)/(\alpha_1+\alpha_\ell)$. Altogether we have that running on $\lfloor x\rfloor$ cores for time $(1-x+\lfloor x\rfloor)T(\alpha_1+\alpha_\ell)$ and on $\lfloor x\rfloor+1$ cores for time $(x-\lfloor x\rfloor)T(\alpha_1+\alpha_\ell)$ does the same amount of work as running on k_1 cores for a time $\alpha_1 T$ and on k_ℓ cores for a time $\alpha_\ell T$. By this exchange the total work amount of the job and the finishing time T of the schedule stay the same. The average core usage of the job stays the same or is reduced, and the difference between the lowest number of cores and the largest number of cores j runs on is decreased by at least one through the replacement.

The above argument can be used repeatedly until $k_\ell\leq k_1+1$. If we use the argument given for j on all jobs, we get a solution with the same running time in which all jobs use the same or a smaller average amount of resources and each job runs on only two adjacent core numbers. As the jobs with their original average core usage fit into the $[0,T] \times [0,m]$ time resource rectangle in the original solution, Lemma 5.2.9 shows that the new solution in which each job runs on only two adjacent core numbers can also be placed within that rectangle. Hence for each optimal schedule there exists a schedule with the same running time in which each job only uses two adjacent core amounts throughout the whole running time of the schedule. Thus the schedules in which each job only uses two adjacent core amounts throughout the whole running time form a dominant set for the investigated problem.

Let us now look at the definition of $\overline{C_j}(x) = w_j/\overline{s_j}(x)$. The $\overline{C_j}$ are well-defined as they are only dependent on the average used core amount. In case of $x \geq 1$ the definition coincides with the finishing time of job j (for a solution in the dominant set, when only two adjacent core numbers are used throughout the schedule). For $x < 1$ the job can finish earlier (for example starting it at time 0 on one core and letting it run uninterruptedly). Let us now look at an optimal solution and a job j with $x_j < 1$ which finishes before $T = C_{max}$. Setting the finishing time of j to T (for example by moving the last instruction of j to the end of the schedule) does not change the objective value of the schedule ($C_{max} = T$). Hence using $\overline{C_j}(x)$ (with x the average resource usage of j during $[0,T]$) instead of the real finishing times does not change the objective value of an optimal schedule. If an optimal discrete schedule with $C_{max} = T$ belongs to the dominant set (each job uses only neighboring resource amounts), it is clear that a feasible continuous domain schedule with the same objective value can be computed by using the $\overline{C_j}$ (just compute the average resource usage x_j of each job j during $[0,T]$, then $\overline{C_j}(x_j) = T$ for all jobs). An optimal continuous domain schedule (length T) can be transformed to a discrete schedule by using the placement from Lemma 5.2.9 (all jobs are placed within the $[0,T] \times [0,m]$ time resource rectangle). As we know from Lemma 5.2.13, it either holds solution-condition-1 (the minimal running time for at least one job is T) or solution-condition-2 ($\sum_j x_j = m$ which means that the $[0,T] \times [0,m]$ time resource rectangle is fully occupied after the placement). This means that at least one job has to run until time T. Thus the objective value has to stay the same after the placement. Hence for each optimal continuous domain solution (computed with the $\overline{C_j}$) there exists a corresponding discrete solution with the same objective value and for each optimal discrete solution (in the dominant set) there exists a corresponding continuous domain solution (computed with the $\overline{C_j}$) with the same objective value. Altogether the usage of $\overline{C_j}$ does not change anything regarding the optimal solutions. Especially Algorithm 5.1 computes an optimal continuous domain solution (computed with the $\overline{C_j}$) and thus an optimal discrete solution with the placement.

Hence we have shown that for each solution there exists an at least equally good solution in which each job only uses two neighboring core numbers. Hence such solutions form a dominant set, and it does not matter when a job uses the one or the other core amount as required in condition 1 from Theorem 1. Also the $\overline{C_j}$ are well-defined by changing the finishing times in the discrete schedules accordingly, and using them instead of C_j does not change the objective value of an optimal solution.

The fast computation for the function and its inverse (we are in the max-case here) as required by **condition 3** from Theorem 1 can be done as follows. We assume that the speedups for each job j for each possible $(0,\dots,m)$ integer amount of cores are given $(s_j(0) = 0, s_j(1) = 1, \dots)$ or can be computed in time $\mathscr{O}(1)$. Also we assume the sequential running time $C_j(1)$ and the minimal core amount of the maximal speedup $\hat{k}_j \leq m$ are given. Due to the concavity of s_j it is also possible to compute $\hat{k}_j$ in $\mathscr{O}(\log m)$. For a given (possibly non-integer) resource amount x for a job j we can compute $\overline{C_j}(x)$ by getting $s_j(\lfloor x \rfloor)$ and $s_j(\lfloor x \rfloor + 1)$ (can be done in $\mathscr{O}(1)$) and then computing $\overline{s_j}(x) = (x - \lfloor x \rfloor) s_j(\lfloor x \rfloor + 1) + (1 - x + \lfloor x \rfloor) s_j(\lfloor x \rfloor)$ (linear interpolation) and finally $\overline{C_j}(x) = C_j(1)/\overline{s_j}(x)$. For the inversion we want to compute the x which fits for a given D the equation $D = \overline{C_j}(x)$. This can be done by first computing the required average speedup $\overline{s_j}(x) = C_j(1)/D$ and then searching a k with $s_j(k) \leq \overline{s_j}(x) \leq s_j(k+1)$ by interval halving in time $\mathscr{O}(\log m)$ (between 0 and $\hat{k}_j$ the speedup s_j is strictly monotonically increasing due to the concavity). With that k we can then compute x by $x = k + (\overline{s_j}(x) - s_j(k))/(s_j(k+1) - s_j(k))$ (due to the linear interpolation) or we can return $x = m+1$ if $\overline{s_j}(x) > s_j(\hat{k}_j)$. Altogether $\overline{C_j}$ and its inverse can be computed both in time $\mathscr{O}(\log m)$. The bend points for this problem are the integers in $(0, m]$. Thus it is clear that the ℓ-th bend point can be found in $\mathscr{O}(1)$ and the number of bend points within $[\underline{x_j}, \overline{x_j}]$ is just $\lfloor \overline{x_j} \rfloor - \lceil \underline{x_j} \rceil + 1$ (for the open interval one has to deduct the boundary points if necessary).

The final condition we have to look for is **condition 4** from Theorem 1 and its parallelization if we want to use Theorem 2. We know that there are no D-values of bend points left within $(\underline{D}, \overline{D})$ when the final solution computation starts. As D^* is not found yet we have $\lfloor \underline{x_j} \rfloor + 1 = \lceil \overline{x_j} \rceil$ for all jobs j. Thus we know $k_j = \lfloor \underline{x_j} \rfloor$ with $k_j \leq x_j^* < k_j + 1$ for each job j and a (still unknown) optimal solution $x^* = (x_1^*, \dots, x_j^*)$ with $\sum_{j=1}^{n} x_j^* = m$. Solutions with $x_i^* = \hat{k}_i$ (solution-condition-1) are already found and cannot occur in this part of the algorithm. For these solutions we can easily compute an optimal resource distribution by assigning each job j an amount of $\overline{C_j}^{-1}(C_i(\hat{k}_i))$ cores. Due to the concavity and $x_j^* < \hat{k}_j$ we have $s_j(k_j + 1) > s_j(k_j)$. We know $\overline{s_j}(x_j^*) = C_j(1)/D^*$ for all j in the optimal solution. This can be rearranged by:

$$
\begin{aligned}
\overline{s_j}(x_j^*) &= C_j(1)/D^* \\
\Leftrightarrow (1 - x_j^* + k_j)s_j(k_j) + (x_j^* - k_j)s_j(k_j+1) &= C_j(1)/D^* \quad \text{def. of } \overline{s_j} \\
\Leftrightarrow x_j^*(s_j(k_j+1) - s_j(k_j)) + (k_j+1)s_j(k_j) & \\
- k_j s_j(k_j+1) &= C_j(1)/D^* \\
\Leftrightarrow D^* x_j^* + D^* \frac{(k_j+1)s_j(k_j) - k_j s_j(k_j+1)}{s_j(k_j+1) - s_j(k_j)} &= \frac{C_j(1)}{s_j(k_j+1) - s_j(k_j)}
\end{aligned}
$$

As this holds for each j

$$
\begin{aligned}
\Rightarrow D^* \sum_{j=1}^{n} x_j^* + D^* \sum_{j=1}^{n} \frac{(k_j+1)s_j(k_j) - k_j s_j(k_j+1)}{s_j(k_j+1) - s_j(k_j)} &= \sum_{j=1}^{n} \frac{C_j(1)}{s_j(k_j+1) - s_j(k_j)} \\
\Leftrightarrow D^* \left(m + \sum_{j=1}^{n} \frac{(k_j+1)s_j(k_j) - k_j s_j(k_j+1)}{s_j(k_j+1) - s_j(k_j)} \right) &= \sum_{j=1}^{n} \frac{C_j(1)}{s_j(k_j+1) - s_j(k_j)}
\end{aligned}
$$

Altogether we have:

$$
\Rightarrow \quad \frac{\sum_{j=1}^{n} \frac{C_j(1)}{s_j(k_j+1) - s_j(k_j)}}{m + \sum_{j=1}^{n} \frac{(k_j+1)s_j(k_j) - k_j s_j(k_j+1)}{s_j(k_j+1) - s_j(k_j)}} = D^* \tag{5.1}
$$

$$
\Rightarrow \quad \frac{C_j(1)/D^* - (k_j+1)s_j(k_j) + k_j s_j(k_j+1)}{s_j(k_j+1) - s_j(k_j)} = x_j^* \tag{5.2}
$$

In Equation (5.1) the only thing still unknown is D^*. The sums can be computed in time $\mathcal{O}(n)$ or in time $\mathcal{O}(n/p + \log p)$ if we compute the solution in parallel on p cores (for $p \geq n$ in $\mathcal{O}(\log n)$). After the sums have been computed, the remainder can be computed in $\mathcal{O}(1)$ (on core 0 in the parallel solution). With a known D^* it is possible to compute x_j^* (by Equation (5.2)) in time $\mathcal{O}(1)$. Thus we can compute all x_j^* in time $\mathcal{O}(n)$ in the sequential solution computation. For the parallel computation it is necessary to broadcast D^* first and then compute the x_j^* locally. This can be done in parallel on p cores in time $\mathcal{O}(n/p + \log p)$ and for $p \geq n$ in $\mathcal{O}(\log n)$. Altogether this shows the following corollary:

Corollary 3. *The scheduling problem $P|var|C_{max}$ (minimize the maximal running time of a set of malleable jobs) with concave speedup functions can be optimally solved in time $\mathcal{O}(n(\log n + \log m)\log(nm))$. If the solution is computed on $p \geq n$ cores of an EREW PRAM, it can be optimally solved in time $\mathcal{O}((\log n + \log m)\log(nm))$.*

In the joint work with Peter Sanders [116] there are some small improvements of the running time ($\mathcal{O}(n + \min\{n,m\} \log m)$ instead of $\mathcal{O}(n(\log n + \log m) \log(nm)))$ as there exists an approximate constant time inversion of the $\overline{C_j}$ and a possibility to reduce the number of relevant jobs to m.

Minimize the Used Energy

Another problem we investigated in a joint work with Peter Sanders [117] was the problem of the minimal possible energy consumption for a set of jobs. This example is clearly based on our article [117] although we show here that the problem fulfills the conditions of our general solution instead of solving the problem directly. We are given a set of malleable jobs with concave speedup functions s_j (with $s_j(0) = 0$ and $s_j(1) = 1$) which have a common release time 0 and deadline T and a machine with identical parallel cores. For the speedup functions we can alternatively assume that they are constant after reaching their maximum. The cores have the additional capability of changing their operation frequency. Of course higher frequency costs more energy. We use the following energy model: If a job runs on p cores with frequency f for a time t the amount of used energy is $E = p \cdot f^{\alpha} \cdot t$ with $\alpha > 2$ (the increase in used energy is clearly super-linear in the operating frequency, see discussion at the beginning of Section 7.1 and the example from Hennessy and Patterson [64, page 23]). The operating frequency of all cores working on the same job at the same time is the same. Each job has to do an amount w_j of work. The work done during a time t is proportional to the operating frequency and the parallel speedup. We use the formula $w = s(p) \cdot f \cdot t$. Besides the concavity of the speedup function we also need a further technical restriction for the speedup function: A function h with $h(0) = 0$ and further defined as $h : p \mapsto \sqrt[\alpha-1]{s^{\alpha}(p)/p}$ must be strictly monotonically increasing for all $p < \bar{p}$ (for a $\bar{p} \in \mathbb{N}$) and monotonically decreasing for all $p > \bar{p}$ and additionally concave on $(0, \bar{p}]$.

The restrictions for the speedup function are met for many classes of speedup functions of parallelizable jobs:

- Linear speedup functions: $s(p) = p$
- Jobs which fulfill Amdahl's Law: $s(p) = \frac{p}{sp+1-s}$ with $s \in (0,1)$ being the sequential fraction.
- Jobs with linear speedup and parallelization overhead g for the following overheads: p, $\log p$, $p \log p$, $\sqrt{p}$, p^2, $\log^2 p$ and speedup function $s(p) = \frac{w+g(1)}{w/p+g(p)}$

For jobs with these restrictions in place it is now possible to define an energy function $\overline{E}(x)$ for non-integer core numbers x. In order to do this, it needs to be shown that using the optimal energy consumption of a job for a fixed (possibly non-integer) number of cores fixes the used frequencies of the job in a way such that we do not need to consider them any more. From Theorem 7 (see Section 7.1.1) we know that there exists an energy optimal schedule for an average core usage of $x \leq \bar{p}$ (including frequency selection) in which the job runs on $\lfloor x \rfloor$ cores for a time $(1-x+\lfloor x \rfloor)T$ and on $\lfloor x \rfloor + 1$ cores for a time $(x - \lfloor x \rfloor)T$ and uses energy

$$\overline{E}(x) = \frac{w^{\alpha}}{T^{\alpha-1}} \cdot ((x-\lfloor x \rfloor) \cdot h(\lfloor x \rfloor + 1) + (1-x+\lfloor x \rfloor) h(\lfloor x \rfloor))^{-\alpha+1}$$

If more than $\bar{p}$ cores are available for the job, then it uses $\bar{p}$ cores throughout T. It is also known from Theorem 7 that the function $\overline{E}(p+\tau)$ is strictly convex and strictly decreasing on $(0,\bar{p}]$ and has its minimum at $\bar{p}$ and $\overline{E}(0+\tau) \xrightarrow[\tau \to 0]{} \infty$. Also $\overline{E}(p+\tau)$ is continuously differentiable on $\mathbb{R}_{>0} \setminus \mathbb{N}$ and the left derivative $\overrightarrow{E}$ is continuous from the left and the right derivative $\overleftarrow{E}$ is continuous from the right. By defining the $x \in \mathbb{N} \cap (0,m]$ as bend points this directly fulfills **condition 2** from Theorem 1.

Given an average core usage x_j for job j we select the energy optimal way of scheduling from Theorem 7 which means that the job runs on $\lfloor x_j \rfloor$ cores for a time $(1-x_j+\lfloor x_j \rfloor)T$ and on $\lfloor x_j \rfloor + 1$ cores for a time $(x_j - \lfloor x_j \rfloor)T$. For every solution $x = (x_1, \ldots, x_n)$ with $\sum_{i=1}^{n} x_i \leq m$ we can use Lemma 5.2.9 to show that the solution also fits into the $[0,T] \times [0,m]$ time resource rectangle. Hence a global schedule consisting of these optimal schedules for each job is feasible. It is not important for the energy usage of a single job and the global schedule objective when a job runs on one core amount or the other during $[0,T]$ (this makes the energy function well-defined for the average core usage). Thus the schedules in which a job only uses two neighboring core numbers form a dominant set of schedules as required by **condition 1** from Theorem 1.

We assume that the values $s_j(p)$ for each job j and an integer $p \in \{0,\ldots,m\}$ are given in a way such that they can be looked up or computed in time $\mathcal{O}(1)$. Thus we can also compute $h_j(p) = \sqrt[\alpha-1]{s_j^{\alpha}(p)/p}$ in time $\mathcal{O}(1)$. We also assume that the minimal core amount $\bar{p}_j$ is given for which the energy function E_j reaches its minimum for job j, otherwise we can compute $\bar{p}_j$ by computing the maximum for h_j by interval halving in $\mathcal{O}(\log m)$ (condition $h_j(p+1) > h_j(p)$ if yes $\bar{p}_j \geq p+1$, if no $\bar{p}_j \leq p$). Given the energy function of a job j according to Theorem 7 we can compute the derivative for an x with $p < x < p+1 \leq \bar{p}_j$ for an integer p especially if we set $p = \lfloor x \rfloor$:

$$\begin{aligned}
\overline{E_j}(x) = &\quad \frac{w_j^\alpha}{T^{\alpha-1}}\cdot((x-p)\cdot h_j(p+1)+(1-x+p)h_j(p))^{-\alpha+1} \\
\overline{E_j}'(x) = &\quad -\frac{w_j^\alpha}{T^{\alpha-1}}\cdot(\alpha-1)(h_j(p+1)-h_j(p)) \\
&\qquad \cdot((x-p)\cdot h_j(p+1)+(p+1-x)h_j(p))^{-\alpha}
\end{aligned}$$

For an integer $p \in \{1,\dots,\bar{p}_j\}$ the left and right derivatives of $\overline{E_j}$ are:

$$\begin{aligned}
\overleftarrow{E_j}(p) = &\quad -\frac{w_j^\alpha}{T^{\alpha-1}}\cdot(\alpha-1)(h_j(p+1)-h_j(p))\cdot h_j^{-\alpha}(p) \\
\overrightarrow{E_j}(p) = &\quad -\frac{w_j^\alpha}{T^{\alpha-1}}\cdot(\alpha-1)(h_j(p)-h_j(p-1))\cdot h_j^{-\alpha}(p)
\end{aligned}$$

Hence we can compute the derivative or the left and right derivative for integers where the derivative does not exist in time $\mathcal{O}(1)$. Due to the strict convexity of $\overline{E_j}$ the derivatives $\overline{E_j}'$ or $\overleftarrow{E_j}$ and $\overrightarrow{E_j}$ are strictly monotonically increasing on $(0,\bar{p}_j]$ and due to the concavity of h_j we have $\overleftarrow{E_j}(p) \geq \overrightarrow{E_j}(p)$ for all $p \in \{1,\dots,\bar{p}_j\}$. Hence we can compute the inversion of the derivative. Let $D \leq 0$ be the given value to be inverted. The first thing to do is to find an integer for which either holds (1): $p_j \in \{1,\dots,\bar{p}_j\}$ with $\overrightarrow{E_j}(p_j) \leq D \leq \overleftarrow{E_j}(p_j)$ or (2): $p_j \in \{0,\dots,\bar{p}_j - 1\}$ with $\overleftarrow{E_j}(p_j) < D < \overrightarrow{E_j}(p_j+1)$ (by using $\overleftarrow{E_j}(0)$ as symbol for $-\infty$). In this case we have $h_j(p_j+1)-h_j(p_j) > 0$. In case (1) the inverse $(\overline{E_j}')^{-1}(D)$ equals p_j in the second case we have

$$
\begin{aligned}
D =& \overline{E_j}'(x_j) \\
=& -\frac{w_j^\alpha}{T^{\alpha-1}} \cdot (\alpha-1)(h_j(p_j+1)-h_j(p_j)) \\
& \cdot ((x_j-p_j)\cdot h_j(p_j+1)+(p_j+1-x_j)h_j(p_j))^{-\alpha} \\
& \Leftrightarrow \\
& ((x_j-p_j)\cdot h_j(p_j+1)+(p_j+1-x_j)h_j(p_j))^{\alpha} \\
=& (-D)^{-1}\cdot \frac{w_j^\alpha}{T^{\alpha-1}} \cdot (\alpha-1)(h_j(p_j+1)-h_j(p_j)) \\
& \Leftrightarrow \\
& (x_j-p_j)\cdot (h_j(p_j+1)-h_j(p_j))+h_j(p_j) \\
=& (-D)^{\frac{-1}{\alpha}} \cdot \sqrt[\alpha]{\frac{w_j^\alpha}{T^{\alpha-1}} \cdot (\alpha-1)(h_j(p_j+1)-h_j(p_j))} \\
& \Leftrightarrow \\
x_j - p_j =& \frac{(-D)^{\frac{-1}{\alpha}} \cdot \sqrt[\alpha]{\frac{w_j^\alpha}{T^{\alpha-1}} \cdot (\alpha-1)(h_j(p_j+1)-h_j(p_j))} - h_j(p_j)}{h_j(p_j+1)-h_j(p_j)} \qquad (5.3)
\end{aligned}
$$

There are at most $2m$ possibilities for D either being in one of the intervals from case (1) or (2). Due to the strict convexity of $\overline{E_j}$ the possibilities are ordered and thus the correct interval can be found in time $\mathcal{O}(\log m)$. In case (1) we then already know $(\overline{E_j}')^{-1}(D) = p_j$ and in case (2) we can compute $(\overline{E_j}')^{-1}(D) = x_j$ by using Equation (5.3). The bend points are the integers which makes addressing them and counting their number within an interval possible in $\mathcal{O}(1)$. This fulfills **condition 3** from Theorem 1 as we can compute the inversion of the derivative (and also the derivative itself) in time $\mathcal{O}(\log m)$.

For the fast final solution as required by **condition 4** from Theorem 1 we know that solution-condition-2 from Lemma 5.2.14 holds ($\sum_{i=1}^n x_i = m$). Also there are no D-values within $(\underline{D},\overline{D})$ and we thus know for each job the $p_j = \lfloor x_j \rfloor$ for which either $\overrightarrow{E_j}(p_j) \leq D^* \leq \overleftarrow{E_j}(p_j)$ (case 1) or $\overleftarrow{E_j}(p_j) < D^* < \overrightarrow{E_j}(p_j+1)$ (case 2) holds.

We compute the sums

$$S_2 = \sum_{j=1}^{n} p_j$$

$$S_3 = \sum_{j \in \text{Case(2)}} \frac{h_j(p_j)}{h_j(p_j+1) - h_j(p_j)}$$

$$S_4 = \sum_{j \in \text{Case(2)}} \frac{\sqrt[\alpha]{\frac{w_j^\alpha}{T^{\alpha-1}} \cdot (\alpha-1)(h_j(p_j+1) - h_j(p_j))}}{h_j(p_j+1) - h_j(p_j)}$$

For the still unknown optimal solution $x^* = (x_1^*, \ldots, x_n^*)$ and the already known p_j we also know the sum $S_1 = \sum_{j \in \text{Case(2)}} (x_j^* - p_j)$ because of $S_1 = m - S_2$. As Equation (5.3) holds for all jobs j in case (2) we have $S_1 = (-D^*)^{\frac{-1}{\alpha}} S_4 - S_3$ where D^* is the common derivative value of the optimal solution. Hence we can compute

$$-D^* = \left(\frac{S_4}{S_3 + S_1}\right)^\alpha$$

And with the known D^* we can compute the x_j^* by Equation (5.3). If we compute the final solution sequentially, we compute the sums S_2, S_3, S_4 in time $\mathcal{O}(n)$, then D^* is computed in time $\mathcal{O}(1)$, and after that Equation (5.3) is evaluated for all jobs in time $\mathcal{O}(n)$. Altogether the sequential final solution can be computed in time $\mathcal{O}(n)$. For the parallel solution on p cores the sums can be computed in time $\mathcal{O}(n/p + \log p)$ by reductions, then D^* is broadcasted (time $\mathcal{O}(\log p)$) after it is computed in constant time on core 0. The local applications of Equation (5.3) can be done in time $\mathcal{O}(n/p)$. This also fulfills the condition for the fast parallel scheduling from Theorem 2.

Altogether this shows the following corollary:

Corollary 4. *Given a set of malleable jobs with concave speedup functions (plus the additional restriction) and a common release time and deadline and a machine whose cores can change their frequency (energy consumption $E = f^\alpha \cdot t$), we can compute the schedule for the minimal energy usage in time $\mathcal{O}(n(\log n + \log m)\log(nm))$. If the solution is computed on $p \geq n$ cores of an EREW PRAM, it can be optimally solved in time $\mathcal{O}((\log n + \log m)\log(nm))$.*

The proof for Theorem 7 can be found in Section 7.1.1 and some further enhancements in Section 7.1.2. It is also important to note that cores not used by any job are not considered in the energy usage (they are considered to be switched off). Here we also have a nice example for the already mentioned resource price. If a job gets more core time when it uses less than $\bar{p}_j \cdot T$, then it can reduce its energy consumption. Hence each quantity of core time is worth an energy amount depending on the job and the core time the job already has. This energy amount is the price of the resource amount (core time) we are willing to pay for the job. If that price is the same for all jobs when we have distributed all resources ($m \cdot T$), we have an optimal solution.

Minimize the Sum of Expected Optimization Values

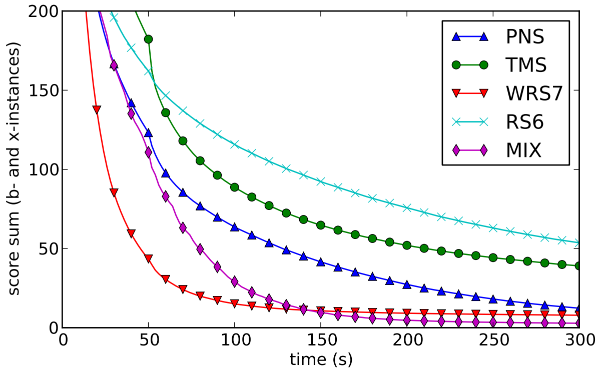


Figure 5.9. A graph taken from *Constraint-Based Large Neighborhood Search for Machine Reassignment* (joint work with Felix Brandt and Markus Völker [18]) which shows the improvements of different strategies for a complex optimization problem. The work is described in Section 5.3.

This example is about the problem of having n different jobs working on optimizations running on the same machine with a common deadline and release time. A possible real world example might be a robot which regularly optimizes some parameters for different tasks in order to minimize the used energy for its electric motors. Only the sum of the different optimizations matters, not the result of a single optimization. For many such optimizations the optimal value might be too difficult to compute because of NP-hardness and time limits, but some approximated result might be enough. For many optimization problems the heuristics behave like the ones in Figure 5.9: the more work is put into an optimization the better the result gets, but the improvements are larger in the beginning and diminish after some work has been done. Especially the objective function looks like a strictly convex function of the used work/time. We want to minimize the sum of these optimization values by distributing the available cores between the different optimization jobs.

We assume that for each optimization j there is a function q_j mapping the invested work w_j (assumed as continuous variable) to the expected value of the optimization and further that this function is strictly convex and strictly monotonically decreasing and continuously differentiable on $\mathbb{R}_{>0}$. Also we assume that the quality of the solution becomes very bad if almost no work is invested ($q_j(w) \xrightarrow[w\to 0]{} \infty$). The objective can then be described as: Minimize $\sum_{j=1}^{n} q_j(w_j)$. Now we take a look at the definition of the amount of work w_j invested into an optimization j. The optimizations are assumed to be parallelizable (in a malleable way) such that the equivalent amount of sequential work done can be computed by $w_j = d_j \cdot s_j(p) \cdot t$ for an integer number of cores p used during a time t. The speedup functions s_j are assumed to first reach their maximum for $\bar{p}$ and to be concave on $[0, \bar{p}]$. Also they are assumed to be strictly monotonically increasing on $[0, \bar{p}]$. As only the total invested amount of work counts for the expected value of the optimization, it is not important at which time within the period the work is done. All optimizers have the same deadline and release time and thus run within the same time interval $[0, T]$.

Now we build a continuous domain function from the average amount of used cores in $[0, T]$ to the expected solution quality for a single optimization job j. We start by constructing a continuous speedup function similar to the example of minimizing the maximal running time. If an optimizer j runs for a time t on p_1 cores and on p_2 cores for the remaining time $T - t$ it does the work $w_j = d_j \cdot (t \cdot s_j(p_1) + (T - t) \cdot s_j(p_2))$. If we only use adjacent core amounts, the speedup is just a linear interpolation like in Figure 5.8. Thus we have the continuous concave speedup functions $\overline{s_j}(x) = (1 - x + \lfloor x \rfloor) s_j(\lfloor x \rfloor) + (x - \lfloor x \rfloor) s_j(\lfloor x \rfloor + 1)$ for each optimizer j with $\overline{s_j}(0) = 0$ and $\overline{s_j}(1) = 1$. The expected quality when using an average core amount x in the time interval $[0, T]$ can then be computed by $\overline{q_j}(x) = q_j(d_j \cdot \overline{s_j}(x) \cdot T)$. For non-integer x we can compute the first and second derivative of $\overline{q_j}$:

$$\begin{aligned}
\overline{q_j}(x) &= q_j(d_j \cdot \overline{s_j}(x) \cdot T) \\
\overline{q_j}'(x) &= q_j'(d_j \cdot \overline{s_j}(x) \cdot T) \cdot d_j T \cdot \overline{s_j}'(x) \\
\overline{q_j}''(x) &= q_j''(d_j \cdot \overline{s_j}(x) \cdot T) \cdot d_j^2 T^2 \cdot (\overline{s_j}'(x))^2 \quad + q_j'(d_j \cdot \overline{s_j}(x) \cdot T) \cdot d_j T \cdot \overline{s_j}''(x) \\
&= q_j''(d_j \cdot \overline{s_j}(x) \cdot T) \cdot d_j^2 T^2 \cdot (\overline{s_j}'(x))^2
\end{aligned}$$

As $\overline{s_j}$ is a piecewise linear function, $\overline{s_j}''(x) = 0$ for all non-integer x. For integer x these derivatives do not need to exist as $\overline{s_j}'$ does not necessarily exist for integer x. For non-integer x we have $\overline{s_j}'(x) = s_j(\lfloor x \rfloor + 1) - s_j(\lfloor x \rfloor)$. Let $\bar{p}_j$ be the core amount at which the optimizer j reaches its highest speedup. Then we have for all integer $p \in \{1, \ldots, \bar{p}_j - 1\}$ that $s_j(p+1) - s_j(p) \leq s_j(p) - s_j(p-1)$ due to the concavity of s_j. For integer $p \in \{1, \ldots, \bar{p}_j - 1\}$ we also can compute the left and right derivatives of $\overline{q_j}$ by $\overrightarrow{\overline{q_j}}(p) = q_j'(d_j \cdot \overline{s_j}(p) \cdot T) \cdot d_j T \cdot (s_j(p) - s_j(p-1))$ and $\overleftarrow{\overline{q_j}}(p) = q_j'(d_j \cdot \overline{s_j}(p) \cdot T) \cdot d_j T \cdot (s_j(p+1) - s_j(p))$. The left derivative is continuous from the left and the right derivative from the right. As q_j is strictly monotonically decreasing ($q'_j(d_j \cdot \overline{s_j}(p) \cdot T) < 0$) and s_j concave, we have $\overrightarrow{\overline{q_j}}(p) \leq \overleftarrow{\overline{q_j}}(p)$. We have $\overline{q_j}''(x) > 0 \; \forall x \in (0, \bar{p}_j)$ as $q''_j(d_j \cdot \overline{s_j}(x) \cdot T) > 0$ due to the strict convexity and $\overline{s_j}'(x) > 0 \; \forall x \in (0, \bar{p}_j)$. Altogether this shows that the (left and right) derivatives of $\overline{q_j}$ are strictly monotonically increasing and that $\overline{q_j}$ is strictly convex between 0 and its minimum $\bar{p}_j$. We also set $\overline{q_j}(x) = \overline{q_j}(\bar{p}) \; \forall x > \bar{p}$ as we can always use less than the available cores. Also $\overline{q_j}$ is continuously differentiable on $(0, m]$ except for the bend points which are the positive integers. With $\overline{q_j}(x) \xrightarrow[x \to 0]{} \infty$ this fulfills **condition 2** from Theorem 1.

For the expected quality q_j of the optimizer j it does not matter when or how the work is done during $[0, T]$, only the total amount of work computed by $\sum_{i=1}^{k} t_i s_j(p_i)$ matters (assumed job j runs on k different amounts of cores, and t_i is the time it runs on p_i cores). Hence only the speedup through the usage of the average core amount x_j matters. We can show that the solutions in which each job j with the average core usage x_j only uses the core amounts $\lfloor x_j \rfloor$ and $\lfloor x_j \rfloor + 1$ form a dominant set of the solutions. This can be done analogously to the example of minimizing the maximal running time due to the concavity of s_j. From Lemma 5.2.9 we know that solutions with only adjacent core amounts can be placed within the $[0, T] \times [0, m]$ time resource rectangle. For the value of $\overline{q_j}$ it does not matter when the work is done within $[0, T]$ (this makes $\overline{q_j}$ well-defined). Thus the overall solution value cannot be changed by reordering the times during which the optimizer j uses $\lfloor x_j \rfloor$ or $\lfloor x_j \rfloor + 1$ cores. Hence we have fulfilled **condition 1** from Theorem 1.

In order to investigate the validity of condition 3 and condition 4 from Theorem 1, we have to look more closely at the functions q_j. Up to now we have only assumed that the functions are strictly convex and strictly monotonically decreasing and continuously differentiable on $\mathbb{R}_{>0}$. Also we assumed $q_j(w) \xrightarrow[w \to 0]{} \infty$. If we start with the simple model that the expected improvements are somehow proportional to the logarithm of the invested work (the needed work grows exponentially with the improvement), we can give a function class for the q_j. We use the following function class for the q_j for the rest of this example:

$$q_j(w_j) = -b_j \cdot \ln(a_j w_j) + c_j$$

The functions of this class are strictly monotonically decreasing and strictly convex for $w > 0$ if $a_j, b_j > 0$. By using $w_j = d_j \cdot \overline{s_j}(x) \cdot T$ we can compute $\overline{q_j}$ and its derivative:

$$\begin{aligned}
\overline{q_j}(x) =& -b_j \cdot \ln(a_j d_j \overline{s_j}(x) T) + c_j \\
& \text{thus we have:} \\
\overline{q_j}'(x) =& \frac{-b_j}{\overline{s_j}(x)} \cdot (s_j(\lfloor x \rfloor + 1) - s_j(\lfloor x \rfloor)) \\
& \text{for non-integer } x \in (0, \bar{p}_j) \\
\overleftarrow{q_j}(x) =& \frac{-b_j}{\overline{s_j}(x)} \cdot (s_j(x+1) - s_j(x)) \\
& \text{for integer } x \in \{1, \ldots, \bar{p}_j - 1\} \\
\overrightarrow{q_j}(x) =& \frac{-b_j}{\overline{s_j}(x)} \cdot (s_j(x) - s_j(x-1)) \\
& \text{for integer } x \in \{1, \ldots, \bar{p}_j\}
\end{aligned}$$

The functions and derivatives can be computed in $\mathcal{O}(1)$. The derivatives can be inverted by first finding an appropriate $\lfloor x \rfloor$ with $\overrightarrow{q_j}(\lfloor x \rfloor) \leq D \leq \overleftarrow{q_j}(\lfloor x \rfloor)$ (case 1) or $\overleftarrow{q_j}(\lfloor x \rfloor) < D < \overrightarrow{q_j}(\lfloor x \rfloor + 1)$ (case 2) and then returning $\lfloor x \rfloor$ in the first case and computing $(\overline{q_j}')^{-1}(D)$ in the second case by computing an x for $\overline{q_j}'(x) = D$:

$$\begin{aligned}
D =& \frac{-b_j}{\overline{s_j}(x)} \cdot (s_j(\lfloor x \rfloor + 1) - s_j(\lfloor x \rfloor)) \\
\Leftrightarrow \overline{s_j}(x) =& \frac{-b_j}{D} \cdot (s_j(\lfloor x \rfloor + 1) - s_j(\lfloor x \rfloor)) \\
=& (1 - x + \lfloor x \rfloor) s_j(\lfloor x \rfloor) + (x - \lfloor x \rfloor) s_j(\lfloor x \rfloor + 1) \\
\Leftrightarrow x - \lfloor x \rfloor =& \frac{-b_j}{D} - \frac{s_j(\lfloor x \rfloor)}{s_j(\lfloor x \rfloor + 1) - s_j(\lfloor x \rfloor)} \qquad (5.4)
\end{aligned}$$

As the left and right derivatives are strictly monotonically increasing, the fitting $\lfloor x \rfloor$ can be found in time $\mathcal{O}(\log m)$, and if needed, x can be computed by Equation (5.4) in time $\mathcal{O}(1)$. The bend points are the integers which makes addressing them and counting their number within an interval possible in $\mathcal{O}(1)$. This fulfills **condition 3** from Theorem 1.

Similarly to the example of minimizing the used energy we show the existence of a fast final solution as required by **condition 4** from Theorem 1. We know that solution-condition-2 from Lemma 5.2.14 holds ($\sum_{i=1}^{n} x_i = m$). Also there are no D-values within $(\underline{D}, \overline{D})$, and we thus know for each job the $p_j = \lfloor x_j \rfloor$ for which either $\overrightarrow{q_j}(p_j) \leq D^* \leq \overleftarrow{q_j}(p_j)$ (case 1) or $\overleftarrow{q_j}(p_j) < D^* < \overrightarrow{q_j}(p_j + 1)$ (case 2) holds.

We compute the sums

$$S_2 = \sum_{j=1}^{n} p_j$$
$$S_3 = \sum_{j \in \text{Case}(2)} -b_j$$
$$S_4 = \sum_{j \in \text{Case}(2)} \frac{s_j(p_j)}{s_j(p_j+1) - s_j(p_j)}$$

For the still unknown optimal solution $x^* = (x_1^*, \dots, x_n^*)$ and the already known p_j we also know the sum $S_1 = \sum_{j \in \text{Case}(2)} (x_j^* - p_j)$ because of $S_1 = m - S_2$. As Equation (5.4) holds for all jobs j in case (2) we have $S_1 = S_3/D^* - S_4$ where D^* is the common derivative value of the optimal solution. Hence we can compute

$$D^* = \frac{S_3}{S_4 + S_1}$$

And with the known D^* we can compute the x_j^* by Equation (5.4). If we compute the final solution sequentially, we compute the sums S_2, S_3, S_4 in time $\mathcal{O}(n)$, then D^* is computed in time $\mathcal{O}(1)$, and after that Equation (5.4) is evaluated for all jobs in time $\mathcal{O}(n)$. Altogether the sequential final solution can be computed in time $\mathcal{O}(n)$. For the parallel solution on p cores the sums can be computed in time $\mathcal{O}(n/p + \log p)$ by reductions, then D^* is broadcasted (time $\mathcal{O}(\log p)$) after it is computed in constant time on core 0. The local applications of Equation (5.4) can be done in time $\mathcal{O}(n/p)$. This also fulfills the condition for the fast parallel scheduling from Theorem 2. This shows the following corollary:

Corollary 5. *Given a set of malleable optimization jobs with concave speedup functions and an expected optimization value $q_j(w_j) = -b_j \cdot \ln(a_j w_j) + c_j$ depending on the invested work w_j, we want to minimize the sum $\sum_{j=1}^{n} q_j(w_j)$. The work amount w_j is defined by $w_j = d_j \cdot s_j(p) \cdot t$ with the speedup function s_j, the running time t and a job-specific constant d_j and additivity for disjoint time intervals. If all jobs have a common release time and deadline, we can compute the optimal core distribution for the minimal expected sum of the optimization values in time $\mathcal{O}(n(\log n + \log m)\log(nm))$. If the solution is computed on $p \geq n$ cores of an EREW PRAM, the problem can be optimally solved in time $\mathcal{O}((\log n + \log m)\log(nm))$.*

Minimize the Weighted Sum of Running Times

This example considers the problem of scheduling malleable jobs on a machine with parallel identical cores in order to minimize the weighted sum of the finishing times ($P|var|\sum \omega_j C_j$) with concave speedup functions for the jobs. As the speedup functions are concave, the C_j are convex functions for jobs starting at time 0, and thus the problem looks similar to the three previous examples. But this problem is different. We will show that it is no convex problem, and the presented techniques do not work in this case. We will also give an illustrating example and show that the problem is NP-hard.

Example: We have a parallel machine with 4 identical cores, two identical malleable jobs each with work amount 6 (the sequential running time), weight $\omega_j = 1$ and the speedup function:

# cores	1	2	3	4
speedup	1	2	3	3

So the speedup is linear until 3 cores for each job and an additional 4th core brings no benefit.

In all previous examples identical jobs got the same amount of resources in the optimal solution. If we divide the resources evenly between the two jobs, each job will get 2 cores in this case which leads to a running time of 3 for each job and thus to a value of 6 for the objective function $\sum \omega_j C_j$. But if we start with giving a job the maximal useful amount of cores (3) and then hand over the cores to the other job when the first is finished, we get a better solution. The first job finishes after running for 2 time units and the second after $2+4/3$ time units which leads to $\sum \omega_j C_j = 4+4/3 \approx 5.33$ and thus to a better solution. The possible schedules are depicted in Figure 5.10.

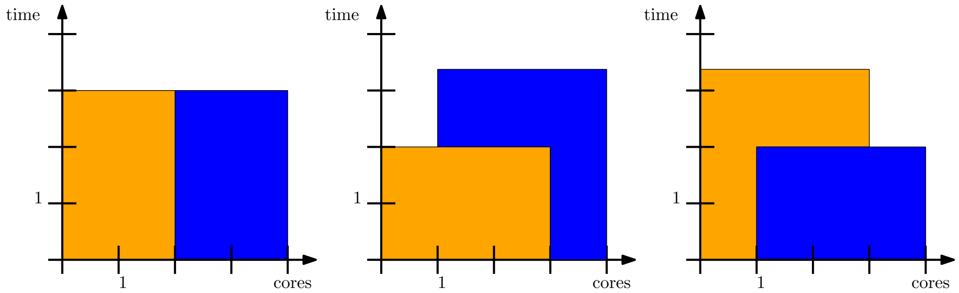


Figure 5.10. Possible different schedules for $P|var|\sum \omega_j C_j$.

The running time of the whole schedule is a little bit longer in case of the better solution $2+4/3 \approx 3.33$, and the average amount of used cores is 1.8 for both jobs. Hence in this case the average amount of resources is not enough to compute the optimal schedule/placement of a job (the jobs are scheduled differently even though their average resource amount is the same). Also the problem is no convex problem any more. If we look at the initial distribution of the cores two extreme cases (1 : 3 and 3 : 1) lead to better results than the even distribution.

For this example **condition 1** from Theorem 1 is violated. Hence we cannot use the developed fast algorithm for this problem.

NP-hardness We can even show that finding the optimal schedule for the problem $P|var|\sum \omega_j C_j$ with concave speedup functions is NP-hard. We prove this by solving a given instance of PARTITION ($a_1, \dots, a_n \in \mathbb{N}$, Question: Is there an $g : \{1, \dots, n\} \rightarrow \{0, 1\}$ with $\sum_{i:g(i)=1} a_i = \sum_{i:g(i)=0} a_i (= 1/2 \cdot \sum_{i=1}^{n} a_i)$?, see Garey and Johnson [51, page 47] for the NP-Completeness of this problem). We call instances of the PARTITION problem which allow a split in two sets with equal weight yes-instances, and the instances which do not allow such a split are called no-instances.

Let us now construct a scheduling problem from the given PARTITION problem $a_1, \dots, a_n \in \mathbb{N}$. For each a_i we construct one malleable job with a concave speedup function $s_i(x) = x$ for $x \leq a_i$ and $s(x) = a_i$ for $x > a_i$. Concave speedup functions lead to convex running time functions (see Lemma 5.2.3). Each job has an amount of work a_i to complete and has a weight $\omega_i = a_i$. Hence the fastest way to complete the job is to run it on a_i cores which leads to a running time of 1. The machine used in the constructed scheduling problem consists of $p = 1/2 \cdot \sum_{i=1}^{n} a_i$ identical cores. If p is non-integer, an g according to the question cannot exist and thus the original PARTITION problem can be answered in time $\mathcal{O}(n)$. Hence we assume this is not the case here.

In order to compare the objective function values of different schedules, we introduce some additional notation. Let t_i be the finishing time of job i and 0 be the common release time of all jobs. As each job has a work amount of a_i and a maximum speedup (when running on a_i or more cores) of a_i, we have $t_i \geq 1$ for all i. If we have a yes-instance, we can construct the basic schedule B for this instance. For B all jobs with $g(i) = 0$ start at time 0 and run on a_i cores, then all jobs with $g(i) = 1$ start at time 1 and run on a_i cores. The objective value for B is then $\gamma(B) = \sum_{i=1}^{n} \omega_i C_i = \sum_{i=1}^{n} a_i t_i = \sum_{i:f(i)=0} a_i + \sum_{i:f(i)=1} a_i \cdot 2 = 3/2 \cdot \sum_{i=1}^{n} a_i$. We set $W = 1/2 \cdot \sum_{i=1}^{n} a_i$. The basic schedule B is the same as in the yes-instance of the NP-hardness proof of $P|size_j, pmtn|C_{max}$ by Drozdowski [39], but the major part of the following proof is entirely different.

Lemma 5.2.16. *The basic schedule B is an optimal schedule for all problems constructed from yes-instances of the original PARTITION problem and has an objective value of $\gamma(B) = 3/2 \cdot \sum_{i=1}^{n} a_i = 3 \cdot W$. Any schedule S for a problem constructed from a no-instance of the original PARTITION problem has an objective value of $\gamma(S) > 3/2 \cdot \sum_{i=1}^{n} a_i$.*

Proof. We start this proof by introducing a new way to describe feasible schedules for the given problem. Given a feasible schedule S for a set of jobs $j_1,\dots,j_n$ with weights $\omega_1,\dots,\omega_n$ and finishing times $t_1,\dots,t_n$. We assume that the jobs are ordered such that the sequence of finishing time is non-decreasing. The sum of all weights of the jobs is $2W$. We now define a function $f_S : [0,2W) \to \mathbb{R}_{>0}$ depending on the schedule by $f_S(x) = t_i$ for $x \in [\sum_{k=1}^{i-1}\omega_k, \omega_i + \sum_{k=1}^{i-1}\omega_k)$. Thus f_S is a monotonically increasing step function on $[0,2W)$. The area between function and x-axis $\int_0^{2W} f_S(x)\mathrm{d}x = \sum_{i=1}^{n} t_i \cdot \omega_i = \gamma(S)$ is the objective value of the schedule. An example for the description is given in Figure 5.11.

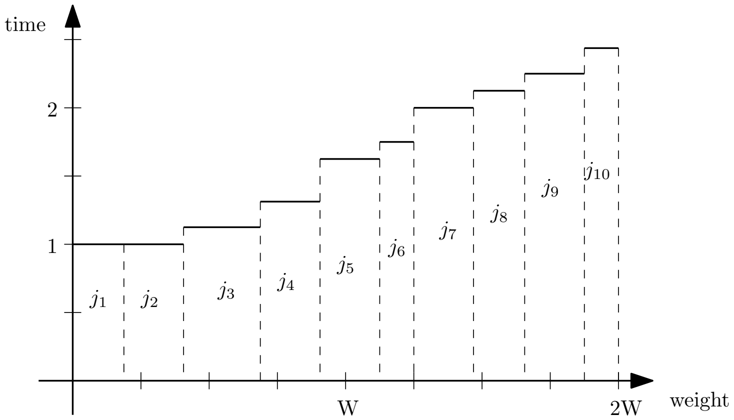


Figure 5.11. Representation of a schedule for the problem $P|var|\sum\omega_jC_j$. The area between the graph and the axis is the value of the objective function.

For a yes-instance the basic schedule B leads to a quite simple function f_B with $f_B(x) = 1$ for $x \in [0,W)$ and $f_B(x) = 2$ for $x \in [W,2W)$ (see Figure 5.12 for an example). $\gamma(B) = 3 \cdot W$ as calculated above.

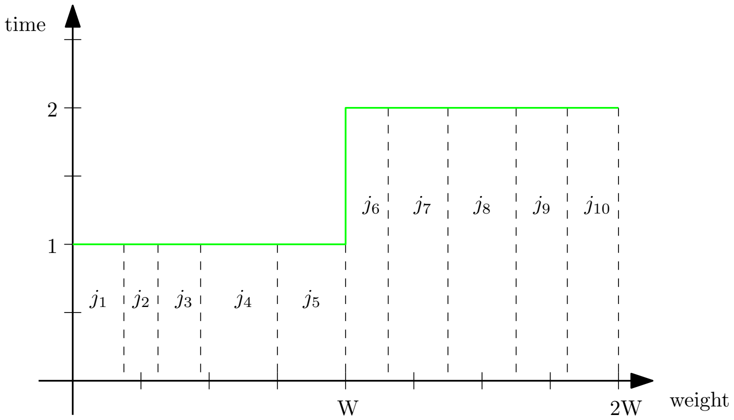


Figure 5.12. Representation of the basic schedule B in case when a partition exists.

Now we look at the differences between an arbitrary schedule S and the basic schedule B and what these differences mean for the resulting objective value of the schedule. From the definition of the speedup functions s_i we know that a job i reaches its maximal speedup on a_i cores and runs with the same speed for all larger core amounts. Hence if a job i runs on more than a_i cores during any time interval, we can reduce the amount of used cores to a_i during this time interval without slowing the job (or any other job) down. Thus we can restrict ourselves w. l. o. g. to the cases in which each job i runs at most on a_i cores.

The speedup functions are linear for all $x \leq a_i$. Thus the efficiency stays the same for all core amounts $x \leq a_i$. Hence the amount of core-time (all time intervals of the usage of all cores summed up) a job needs to finish is the same for all cases we restrict ourselves to.

As $s_i(x) \leq a_i$ and a job i has an amount of work a_i to do, it follows immediately that no job can be finished before time 1, as the minimal execution time for all jobs is 1. The work done by the whole machine between time 0 and 1 is at most $1 \cdot p = W$ and the weight of each job equals its work amount. Thus the total weight of jobs completed until time 1 is at most W. Now we take a look at the jobs j_i with finishing times $1 < t_i < 2$ and $\sum_{k=1}^{i} \omega_k > W$. If there are no such jobs in schedule S, then all jobs j_i with $\sum_{k=1}^{i} \omega_k \leq W$ finish at time $t_i \geq 1$ and all jobs j_i with $\sum_{k=1}^{i} \omega_k > W$ finish at time $t_i \geq 2$, which leads to an objective value $\gamma(S) \geq 3W = \gamma(B)$. The equality $\gamma(S) = \gamma(B)$ can in this case only be satisfied if all jobs j_i with $\sum_{k=1}^{i} \omega_k \leq W$ finish at time $t_i = 1$ and all jobs j_i with $\sum_{k=1}^{i} \omega_k > W$ finish at time $t_i = 2$ and if there exists an i with $\sum_{k=1}^{i} \omega_k = W$ (which is only possible for a yes-instance).

We distinguish two cases (with respect to the ordered set of jobs):

- **Case 1:** There exists $\ell \in \{1, \dots, n\}$ such that $\sum_{k=1}^{\ell} \omega_k = W$ (only possible for yes-instances).
- **Case 2:** There exists $\ell \in \{1, \dots, n\}$ such that $\sum_{k=1}^{\ell} \omega_k > W > \sum_{k=1}^{\ell-1} \omega_k$.

In **case 1** we define the following three job sets: I_1 is the set of jobs j_i with $\sum_{k=1}^{i} \omega_k \leq W$; I_b is the set of jobs j_i with finishing times $1 < t_i < 2$ and $\sum_{k=1}^{i} \omega_k > W$; and I_2 is the set of jobs j_i with finishing times $t_i \geq 2$ and $\sum_{k=1}^{i} \omega_k > W$. For the basic schedule B the set I_b is empty. In a slight misuse of notation we will also use the job sets I_1, I_b and I_2 to denote the respective sets of indices.

In **case 2** we split the job j_ℓ into two jobs $j_{\ell'}, j_{\ell''}$ with the same finishing time but each with an appropriate share of the work and the weight (weight and work is the same for all jobs) such that $\sum_{k=1}^{\ell-1} a_k + a_{\ell'} = W$. This is still a feasible schedule due to the malleability and the properties of the speedup function, also the objective function is not changed by that operation. The speedup functions are irrelevant for the new jobs as we do the split only for the purpose of computing the objective value and not for rescheduling. We also change the indices and n

such that $1, \ldots, n$ are the indices for all jobs in the used order. Then we define the same three sets as in case 1.

The jobs in I_b are the only ones that can contribute to the improvement of the objective value of S compared to B. Figure 5.13 shows the improvement against B in blue and the degradation against B (jobs of the sets I_1 and I_2) in red.

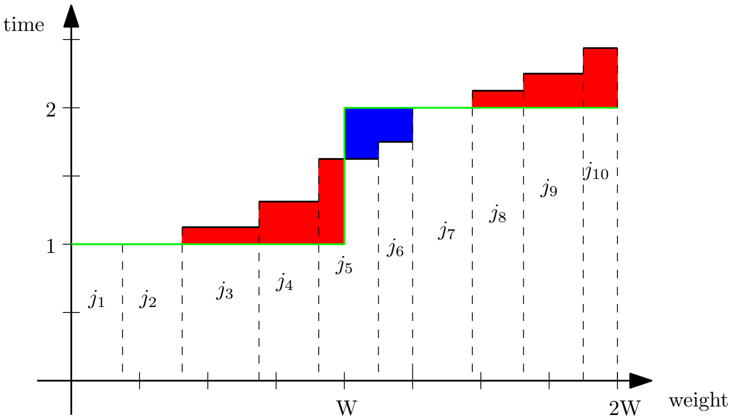


Figure 5.13. Comparison of the basic schedule with a different schedule.

A job $j_i \in I_b$ can perform at most an amount of $(t_i - 1) \cdot a_i$ work after time 1. Thus it has to perform an amount of at least $(2 - t_i) \cdot a_i$ work *before* time 1. Altogether the jobs in I_b have to perform an amount of $\sum_{i \in I_b}(2 - t_i) \cdot a_i$ work before time 1. Hence the jobs $j_i \in I_1$ cannot all finish at time 1 as they have to perform at least an amount W of work. Thus the jobs $j_i \in I_1$ have to do an amount of $\sum_{i \in I_b}(2 - t_i) \cdot a_i$ work after time 1. If a job $j_k \in I_1$ has to do an amount of work w, it takes at least time w/a_k as a_k is the maximal speedup of job j_k. Job j_k also has a weight of a_k in the objective function. Hence we get a lower bound of the contribution of the jobs in I_1 as it does not matter which jobs have remaining work after time 1. This leads to a lower bound of $W + \sum_{i \in I_b}(2 - t_i) \cdot a_i$ for the contribution of the jobs in I_1 to the objective function (the W part comes from $\sum_{i \in I_1} a_i = W$ and the fact that no job in I_1 can finish before time 1). The contribution to the objective function of the jobs in I_b is $\sum_{i \in I_b} t_i \cdot a_i = 2 \cdot \sum_{i \in I_b} a_i - \sum_{i \in I_b}(2 - t_i) \cdot a_i$.

Now we take a look at the situation when a job $j_i \in I_b$ finishes. Due to the definition of I_b we have $\sum_{k=i+1}^{n} a_k < W = p$. As the maximal degree of parallelism for a job j_k is a_k, there can be at most $\sum_{k=i+1}^{n} a_k$ cores in use after job j_i has finished (all jobs of I_1 have finished before j_i due to the ordering of the jobs). Hence the idle core time between the times 1 and 2 is at least $\sum_{i \in I_b}(2 - t_i) \cdot a_i$. As the total amount of work of all jobs is $2W$ and the work that can be done before time 2 by the whole machine is also $2W$, an amount of $\sum_{i \in I_b}(2 - t_i) \cdot a_i$ work has to be done after time 2 due to the idleness induced by the jobs in I_b. Similar to the argument above it holds that if a job $j_k \in I_2$ has to do an amount of work w it takes at least time w/a_k as a_k is the maximal speedup of job j_k. Job j_k also has a weight of a_k in the objective function. Hence for the lower bound of the contribution of the jobs in I_2 it does not matter which jobs have remaining work after time 2. This leads to a lower bound of $2 \cdot \sum_{i \in I_2} a_i + \sum_{i \in I_b} (2 - t_i) \cdot a_i$ for the contribution of the jobs in I_2 to the objective function (the $2 \cdot \sum_{i \in I_2} a_i$ part comes from the fact that no job in I_2 finishes before time 2 by definition).

Altogether we have for the objective function of schedule S:

$$\begin{aligned}
\gamma(S) =& \sum_{i=1}^{n} t_i \cdot a_i = \sum_{i \in I_1} t_i \cdot a_i + \sum_{i \in I_b} t_i \cdot a_i + \sum_{i \in I_2} t_i \cdot a_i \\
\geq& W + \sum_{i \in I_b} (2 - t_i) \cdot a_i + \sum_{i \in I_b} t_i \cdot a_i + 2 \cdot \sum_{i \in I_2} a_i + \sum_{i \in I_b} (2 - t_i) \cdot a_i \\
=& W + \sum_{i \in I_b} (2 - t_i) \cdot a_i + 2 \cdot \sum_{i \in I_q \cup I_2} a_i \\
=& W + \sum_{i \in I_b} (2 - t_i) \cdot a_i + 2 \cdot W \\
=& \gamma(B) + \sum_{i \in I_b} (2 - t_i) \cdot a_i
\end{aligned}$$

Hence $\gamma(S) > \gamma(B)$ when $I_b \neq \emptyset$.

If the given scheduling problem stems from a yes-instance of the PARTITION problem, an optimal solution must fulfill $I_b = \emptyset$. The minimal finishing time for jobs in I_1 is 1 and 2 for jobs in I_2. The total weight of jobs contained in set I_1 can be at most W. The basic schedule B reaches the optimum in every respect and is thus an optimal schedule for the problems stemming from yes-instances.

When the given scheduling problem stems from a no-instance, building a basic schedule B with $\sum_{i \in I_1} a_i = \sum_{i \in I_2} a_i = W$ and $t_i = 1 \ \forall i \in I_1$ and $t_i = 2 \ \forall i \in I_2$ is impossible as such a partition does not exist without splitting a job, and the two parts of the split job (one in I_1, one in I_2) have to have the same finishing time. If a schedule S for a no-instance has a non-empty set I_b, we know that $\gamma(S) > 3W$. As $\sum_{i:t_i=1} a_i < W$, we have in case of $I_b = \emptyset$ that $\sum_{i:t_i \geq 2} a_i > W$ and thus $\gamma(S) \geq 1 \cdot \sum_{i:t_i=1} a_i + 2 \cdot \sum_{i:t_i \geq 2} a_i > 3W$. Hence the resulting objective function value is always larger than $3W$ in case of a no-instance. □

Lemma 5.2.16 shows that if we can compute the optimal schedule for the constructed scheduling problem, we can decide the partition problem which the respective instance stems from. Thus we get the following theorem:

Theorem 6. *The scheduling problem of minimizing the weighted sum of running times of malleable jobs ($P|var|\sum \omega_j C_j$) with concave speedup functions is NP-hard and thus unlikely to have a polynomial-time scheduling algorithm that always finds the optimal solution.*

The NP-hardness of the minimization of the weighted sum of running times indicate that the conditions formulated in Theorem 1 are not only the result of the selected solution approach. The minimization of the weighted sum of running times is a problem which looks similar to problems that can be optimally solved in polynomial time by our approach, but it violates the conditions that are necessary for our approach. Thus the conditions somehow separate polynomially solvable problems from NP-hard ones and are thus (at least partially) justified by properties of the problem and not only properties of the solution approach.

The NP-hardness of the problem with sequential jobs ($P||\sum \omega_j C_j$) is already known from the work of Bruno et al. [19]. They do not discuss preemption. Hence their work does not directly imply the NP-hardness of $P|var|\sum \omega_j C_j$ with concave speedup functions (*var* implies *pmtn*). Also a proof with parallel jobs seems to be more fitting for this work.

5.2.6 Techniques for Enhanced Problems

In this section we present some possibilities to extend the results from Section 5.2.2 and Section 5.2.3 (and also Section 5.2.4) to different kinds of scheduling problems.

Approximations for Problems with Moldable Jobs

In Section 3.1.3 we presented some fast approximation algorithms for the scheduling problem of minimizing the maximum running time of a set of independent moldable jobs ($P|any|C_{max}$ and $P|any,pmtn|C_{max}$). Here we show how we can use our scheduling results for malleable jobs (as presented in Section 5.2.3) to get approximation solutions for scheduling problems with moldable jobs. We look at the problems $P|any|C_{max}$ and $P|any,pmtn|C_{max}$ with concave speedup functions for each job (speedup with $s_i(0) = 0, s_i(1) = 1$). For malleable jobs instead of moldable jobs this problem can be solved optimally in time $\mathcal{O}(n(\log n + \log m)\log(nm))$. If the solution is computed on $p \geq n$ cores of an EREW PRAM, it can even be solved optimally in time $\mathcal{O}((\log n + \log m)\log(nm))$ (see Corollary 3 in Section 5.2.5). As each feasible schedule for moldable jobs is also a feasible schedule for malleable jobs, the optimal solution for malleable jobs is at least as good as the optimal solution for moldable jobs. Hence the comparison of any solution for moldable jobs with the optimal solution for the analogous problem for malleable jobs delivers an upper bound for the approximation ratio compared to the optimal solution for moldable jobs.

Approximation algorithm:

1. Treat the moldable jobs as malleable jobs and compute the optimal solution for that problem with Algorithm 5.1 (linear interpolation of the speedups) or

its parallelization as described in Section 5.2.4. This leads to $C_{max} = T_{var}$ and an average core usage x_i of job i.

2. Let B be the set of *big* jobs (jobs with an average core usage of $x_i > 1$) and let S be the set of *small* jobs (jobs with an average core usage of $x_i \leq 1$) in this solution. Each job in B is set to a parallelism degree of $\lfloor x_i \rfloor$ and gets its own $\lfloor x_i \rfloor$ cores. The big jobs can be placed in parallel by the usage of a prefix sum (similarly to the method in Section 5.2.4). The jobs in S are scheduled on the remaining (after the placement of B) cores in the next step.
3. The small jobs are running sequentially and each has a duration of at most T_{var}. At least $\lceil \sum_{i \in S} x_i \rceil$ cores are available to work on the small jobs. The total duration of all small jobs together is at most $T_{var} \cdot \sum_{i \in S} x_i$ as our malleable scheduling algorithm assigned them a total core time of $T_{var} \cdot \sum_{i \in S} x_i$. In order to split the jobs among the available cores, we compute the starting time and the finishing time of each job if they were computed on one core one after another (no special ordering required). This can also be done in parallel by using a prefix-sum. We assume that the cores participating in the work on the small jobs are numbered from 1 to m'. We assign each core k to the interval $[T_{var}(k-1), T_{var}(k+1)]$ which overlaps the intervals assigned to the cores $k-1$ and $k+1$ if these exist. An example for 8 cores is presented in Figure 5.14. If we split the jobs between the cores k and $k+1$ in the interval $[T_{var}k, T_{var}(k+1)]$ (the overlap of the assigned intervals), then we get at most m' job groups ($m'-1$ splits) all with a total length of at most $2T_{var}$. To find a splitting point without splitting a job is always possible as the maximal job length is bounded by T_{var}. We always choose the smallest possible value in the overlapping interval $[T_{var}k, T_{var}(k+1)]$ which coincides with the finishing time of a job started at a time before kT_{var}. This job and the splitting time can be easily determined by the prefix sum by detecting the job with starting time $< kT_{var}$ and finishing time $\geq kT_{var}$. A solution (similar to the scheduling of small jobs) for scheduling sequential jobs in parallel is also given in Peter Sanders's lecture "Parallele Algorithmen".

This approach does not use preemption and thus works for both problems ($P|any|C_{max}$ and $P|any, pmtn|C_{max}$ with concave speedup functions for each job). It has an approximation ratio of 2:

Lemma 5.2.17. *The approach described above (computing the optimal malleable solution and then round down the big jobs and schedule the small jobs by job split) is a factor* 2 *approximation for the problems $P|any|C_{max}$ and $P|any, pmtn|C_{max}$ with concave speedup functions (speedup with $s_i(0) = 0, s_i(1) = 1$).*

Proof. As each feasible schedule for moldable jobs is also a feasible schedule for malleable jobs, the optimal solution for malleable jobs is at least as good as the

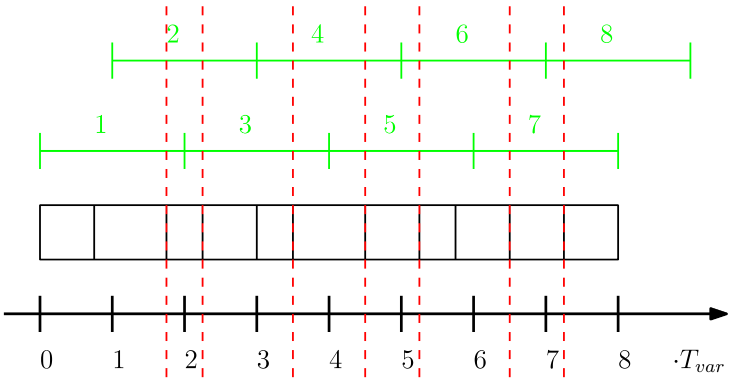


Figure 5.14. Example of the job split on 8 cores.

optimal solution for moldable jobs. Hence it is sufficient to prove that the running time of the schedule produced in step 2 cannot be larger than $2 \cdot T_{var}$.

Let us first look at the big jobs B. The speedup functions are concave with $s_i(0) = 0$ and $s_i(1) = 1$. Also the speedups are only defined for integer core amounts. For core amounts in between we use linear interpolation, like in Corollary 3. We can also assume that $x_i \leq \hat{x}_i$. Thus we have due to the concavity of the speedup functions:

$$\frac{s_i(x_i)}{s_i(\lfloor x_i \rfloor)} \leq \frac{s_i(\lfloor x_i \rfloor + 1)}{s_i(\lfloor x_i \rfloor)} \leq \frac{\lfloor x_i \rfloor + 1}{\lfloor x_i \rfloor}$$

The last inequality comes from the fact that the growth of a concave function during an integer step can be at most the average growth between 0 and the starting point of the step. Hence the duration of a big job can be at most doubled when changing from x_i to $\lfloor x_i \rfloor$ cores. The total core usage of all big jobs is $\sum_{i \in B} \lfloor x_i \rfloor \leq \lfloor \sum_{i \in B} x_i \rfloor$.

The small jobs S can be computed in time T_{var} on an average core amount of $\sum_{i \in S} x_i$ in the malleable solution. The splitting algorithm guarantees that no core working on the small jobs has to work more than a time amount of $2T_{var}$ and that all small jobs are done. □

In the $\mathcal{O}$-notation the running times of step 2 and step 3 are smaller than the running time of step 1 in the sequential case and in the parallel case with $p \geq n$ cores working on the schedule in an EREW PRAM. Hence the running times of Theorem 1 ($\mathcal{O}(n(\log n + \log m) \log(nm))$) and Theorem 2 ($\mathcal{O}((\log m + \log n) \cdot \log(nm))$) are also the running times for the sequential and the parallel version of the approximation algorithm. Thus we get a fast, easy and even parallelizable factor-2 approximation algorithm for $P|any|C_{max}$ and $P|any, pmtn|C_{max}$ with concave speedup functions for each job. The approximation can also be used for a mixture of moldable and malleable jobs. If all jobs are large (with a minimal degree of parallelism of k in the malleable solution), then we even get an approximation ratio of $(k+1)/k$.

Usage for Online Problems

Malleable job schedules enable easy adaptions to online problems, at least for the ones where all job properties become known together with the job. Every time a new job arrives we can just compute the new optimal solution (can be done fast and even in parallel see Theorem 2) and adapt all running jobs to it. Unfortunately the solutions computed in this way do not have to be optimal. If we assume a very simple instance of the problem "Minimize the used Energy" from Section 5.2.5, we already get a case with a suboptimal result which is described now. We have $m = 1$ and two jobs with a common deadline T and work amount w. The only difference between the jobs is that job 1 becomes known at time 0 and job 2 at time $T/2$. Thus the optimal solution at time 0 is to run job 1 with frequency $f = w/T$. At time $T/2$ when job 2 arrives, the optimal solution changes to running job 2 and the remaining part of job 1 with frequency $f = 3w/T = 3/2 \cdot w/(T/2)$. This clearly consumes more energy than the optimal solution which is running both jobs with frequency $f = 2w/T$ ($E_{opt} = T \cdot (w/T)^{\alpha} \cdot 2^{\alpha}$ instead of $E = T \cdot (w/T)^{\alpha} \cdot (1+3^{\alpha})/2$) because $g : x \mapsto x^{\alpha}$ is a strictly convex function for $x > 0$. Hence online problems are not optimally solved in all cases by the usage of malleable jobs.

5.3 General Optimization Methods in Scheduling

There are a lot of applications of general algorithmic design techniques in scheduling. Many of these applications are heuristics. There are complete books which are collections of such applications both within Xhafa and Abraham [145] and outside Xhafa and Abraham [146] computer science. Dealing with heuristics is especially interesting when different methods are compared regarding their solution quality for the same problems. Together with Felix Brandt and Markus Völker, the author of this work took part in the ROADEF/EURO challenge 2012 which had machine reassignment problems as topic (Afsar et al. [1] wrote an article about the challenge, results and the used techniques). We used general heuristic methods in order to solve the scheduling problems presented in the challenge. Hence this is an example of the application of general optimization methods to scheduling.

5.3.1 ROADEF/EURO Challenge 2012

We now take a deeper look into the problems to be solved for the challenge using the challenge description [113]. Given a set of machines $\mathscr{M}$ (up to 5 000 for the given problems) and a set of processes $\mathscr{P}$ (up to 50 000 for the given problems), a machine assignment is a map $M : \mathscr{P} \to \mathscr{M}$ assigning each process to a machine. An initial feasible assignment M_0 was given for each problem. The goal was to compute a new assignment regarding some hard constraints and some objectives. The hard constraints were:

- Capacity constraints: For each problem a number of resources is given (ranging from 3 to 12 for the given instances of the main round). For each kind of resource each machine has a capacity and each process a demand. The demand of all processes assigned to one machine can be at most the capacity of this machine for each resource.
- Conflict constraints: The set of processes is partitioned into services. All processes of a service must run on different machines.
- Spread constraints: The machines are distributed over several locations. For each service there is a minimum number of locations where at least one process of that service should run.
- Dependency constraints: A neighborhood is a set of machines, all neighborhoods are disjoint. If a service a depends on a service b, then all processes of a should run in a neighborhood in which also a process of b runs.
- Transient usage constraints: Some resources can be transient resources (depending on the problem). The process uses this resource on its initial and its new machine if it is moved.

The objective of the problem (cost function) is an (instance-specific) weighted sum of five objectives:

- Load cost: For each machine there is a safety capacity for each resource. If the assignment produces a usage of a resource which is higher than the safety capacity, then there is a cost proportional to the usage amount above the safety capacity. For each problem instance the different resources can be weighted differently.
- Balance cost: For some 2-tuples of resources there can be a balance cost defined as a 3-tuple (r_1, r_2, c). If u_1 is the usage of r_1 and u_2 that of r_2 on a certain machine and $c \cdot u_1 - u_2 > 0$, then the usage cost for the given tuple on the machine is $c \cdot u_1 - u_2$.
- Process move cost: The cost of moving a process (process-specific).
- Service move cost: The number of processes moved within a service. This

cost is only payed for the service with the most moved processes.

- Machine move cost: The cost of moving a process from one machine to another (specific to the two involved machines).

The challenge was organized in two rounds: a qualification round and the main round. In the qualification round ten problems of smaller sizes (up to 1000 processes) had to be optimized. The description here is focused on the main round, as we put the most effort into the program for the main round and as the final ranking is only based on the results of the main round. In both rounds the participating teams had to hand in a program that optimizes problems as described above given in a defined data format. The program for the main round was tested on 10 problem instances already known to the participants and 10 unknown new instances with similar properties. For each instance an initial feasible solution was given. All programs were run with a time limit of 5 minutes on each problem instance on a core2duo E8500 3.16MHz with 4GB RAM on Debian 64 or Win7 64 bits (see the submission page [114]). All programs were evaluated through the sum of $Score()$ over all instances. Given an instance and computed solution I, the best solution among competitors is B and the original reference solution R, the score is $Score(I) = 100 \cdot (C(I) - C(B))/C(R)$ where C is the cost function.

5.3.2 Our Approach to the Challenge

The description given here is an adapted version (large parts are directly copied) of the article to our solution approach [18] (joint work with Felix Brandt and Markus Völker, for the individual contributions to the articles used in this work see Section 3.4.4). Compared to the article, the focus here is more directed to parts of the solution with contributions of the author of this work. Let us first take a look at the basic properties of the given 10 instances (called b-instances in the challenge) for the main round:

By looking at Table 5.1 we can see that there are 10 to 500 times more processes than machines and that the set of processes is partitioned in a large number of disjoint subsets called services. These services are included in three hard constraints: the conflict constraint, the spread constraint and the dependency constraint. For most instances the average number of dependencies between services ranges between 0.3 and 3, but for instance 9 the average is 9.4, for instance 10 the average is 9.7 and for instance 4 the average is even 23.4. Altogether the constraints for a feasible solution are difficult to meet (at least for some instances). Thus building a feasible solution from scratch seems to be difficult (especially within a time limit of 5 minutes). Improving the given feasible solution step by step looks much more promising and possible. Hence we decided to use an approach that improves the given initial solution instead of computing a new solution

	Processes	Machines	Services	Resources (transient)	Balance costs
Instance 1	5000	100	2512	12 (4)	0
Instance 2	5000	100	2462	12 (0)	1
Instance 3	20000	100	15025	6 (2)	0
Instance 4	20000	500	1732	6 (0)	1
Instance 5	40000	100	35082	6 (2)	0
Instance 6	40000	200	14680	6 (0)	1
Instance 7	40000	4000	15050	6 (0)	1
Instance 8	50000	100	45030	3 (1)	0
Instance 9	50000	1000	4609	3 (0)	1
Instance 10	50000	5000	4896	3 (0)	1

Table 5.1. Overview of the given instances of the ROADEF/EURO Challenge 2012. Giving the number of processes (jobs), machines, services and the number of resources and the number of those which are transient. Additionally the number of tuples for balance cost is given.

from scratch. As basic method we used a heuristic similar to the neighborhood search (called local search by Hromkovič, [67, page 189]) or large neighborhood search (Ahuja et al. [3]). The idea of neighborhood search is to compute for a given feasible solution some feasible neighboring solutions and to choose an improving solution among them. The improved neighboring solution is then the basis for the next step. The neighborhood search has the additional benefit that there is always an available feasible solution which can be returned if the time limit comes up. In order to find an improving solution within our neighborhoods and given the complicated constraints, we decided to use a constraint programming library (Gecode [130]) to search for an improving solution within the selected neighborhood. Hence our general solution method is the repeated application of the following three steps until the given time limit is reached:

1. Neighborhood selection. Choose a subset of all processes. Only these processes are considered for reassignment in the current iteration.
2. Constraint search. Build a constraint program (CP), which contains only the selected subset of processes and calculates their costs when reassigned to other machines. Start a CP search for improving solutions and stop at the first improving solution or when the time limit is reached.
3. Update solution. Take the found solution as the starting solution of the next iteration. If no improving solution is found in the given search space or within

the iteration's time limit, the current solution is kept for the next iteration.

In preliminary experiments we observed that a high number of iterations is a key to good results. This led to two design decisions: First, we keep the subsets small (most of the time less than 10 processes are selected), because we can run many iterations on small process subsets in the same time as one on a big subset. Second, we stop each iteration on the first improving solution. This way the number of iterations can be dramatically increased, compared to fully exploiting the CP search space in each iteration. For the constraint search our model does not contain the the full problem for performance reasons but rather represents the difference to the current solution.

Data analysis The knowledge of the properties of the inputs is the basis for developing good heuristics. Hence we took a deeper look into the given instances in order to discover some maybe helpful properties. Of course we were careful not to over-analyze the data and not to build assumptions that might be false for the unknown data sets.

The first thing we observed was that the **service move cost** is not able to contribute more than a tiny fraction of the total cost. For all given instances, the weight assigned to the service move cost was 10. As the service move cost is defined by the maximum number of processes moved in any service multiplied by this weight, it is bounded by 10 times the number of processes. As the initial costs for all instances were far more than 10000 times the number of processes, these costs seemed negligible. After we had computed the first optimized solutions with our search process, we could see that the service move cost was still limited to about 1% of the solution cost (in most cases far less). Thus we considered the service move cost as insignificant and ignored it during the optimization process. With that we saved computing time and programming effort.

The second observation is that all instances have some **large resource demanding processes**. We order the processes for each b-instance and each resource according to their need of this resource from processes that need little of that resource up to the processes that need much of this resource. If a process belongs to the *r%* processes that use the least of a certain resource, we will say that this process belongs to the lower *r%* process quantile of that resource. Analogously for a process that belongs to the *r%* processes that use most of a certain resource, we will say that this process belongs to the upper *r%* process quantile of that resource. If a process belongs to the lower *r%* quantile for ALL resources we will say that it is in the lower *r%* set of all processes. If a process belongs to the upper *r%* quantile for at least ONE resource we will say that it is in the upper *r%* set of all processes.

Now we take a look at the very similar 3-resource instances 8, 9 and 10: For

	Processes upper 5%	Resources upper 5%	Processes lower 60%	Resources lower 60%	Resources
Instance 1	30%	51-84%	11%	0.1-2.0%	12
Instance 2	30%	53-84%	11%	0.1-2.1%	12
Instance 3	19%	41-63%	26%	1.1-5.2%	6
Instance 4	18%	42-64%	26%	1.1-5.2%	6
Instance 5	19%	42-64%	26%	1.1-5.3%	6
Instance 6	19%	42-64%	26%	1.1-5.4%	6
Instance 7	19%	41-63%	26%	1.1-5.3%	6
Instance 8	10%	46-60%	51%	2.1-3.9%	3
Instance 9	10%	45-60%	51%	2.2-3.9%	3
Instance 10	10%	45-60%	51%	2.2-3.9%	3

Table 5.2. Fraction and resource demand fraction of the processes in the upper 5% set (process belongs to the largest 5% for ONE resource) and the lower 60% set (process belongs to the smallest 60% for ALL resources). The range in the resource demand is between the resource with the smallest and the largest demand for the respective set. For instances with the same number of resources the values are very similar which might be a generation artifact.

these instances 51% of all processes are in the lower 60% set of all processes and what is much more interesting the resource usage of all jobs in the lower 60% set is between 2% and 4% of the total usage of the respective resource usage in the respective instance. Hence we have a large number of processes which contribute very little to the overall load. On the other side for all these three instances the fraction of processes belonging to the upper 5% set is between 9% and 10% but the resource usage of the upper 5% set is between 45% and 60% of the total usage of the respective resource usage in the respective instance. Hence we have a relatively small number of processes which contribute an important fraction to the total load. More than 90% of the processes in the upper 5% set belong to the upper 30% process quantile for all resources. Similar but weaker properties like this also occur in the other instances. The properties of all instances are shown in Table 5.2. Another interesting observation is that the instances with the same number of resources seem to be very similar with respect to large and small processes.

Hence there is a large fraction of processes that have little importance for the total load or the excess load cost, but there are some very important processes which might need a special handling. This is even more important as for 6 instances there are machines with all safety capacities set to 0 which implies that all processes on these machines should be moved elsewhere. These findings inspired the creation of the Target Move Search (see below), which works by enabling the movement of large processes (for example from the upper 5% set) by making room for them on another machine.

	Demand	Safety capacity
Instance 1	86-88%	83-85%
Instance 2	85-89%	87-88%
Instance 3	83-84%	83-84%
Instance 4	83-84%	87-88%
Instance 5	83-84%	83-85%
Instance 6	83-84%	87-88%
Instance 7	76-77%	83%
Instance 8	84%	83-84%
Instance 9	91%	87-88%
Instance 10	77%	84%

Table 5.3. Total demand (all processes) and safety capacity (of all machines) of the given instances as a fraction of the total capacity (of all machines). The range is between the resource with the smallest and the highest value.

We also took a look at the **total load** and the **total safety capacity** of the given instances. We give the total resource demand of all processes and the total safety capacity of all machines as fraction of the total capacity of all machines in Table 5.3. The fractions are computed for each resource separately and give the range of results. We can see from Table 5.3 that the values for the demand and the safety capacity are usually close together and that there is little variance between different resources.

Correlations between demands for different resources might also be interesting. If there are a lot of processes with complementary resource demands, it might be especially beneficial to optimize these processes by packing them together. In all instances we took the processes as observations and the different resources as random variables. Then we computed the correlation coefficients between the different resources. In all instances we only found positive correlation coefficients and coefficients close to zero. Especially for the three instances with only three resources, where it might have been possible to find fitting complementary resource demands, all coefficients were at least 0.5. Thus we did not develop a special treatment for processes which use some resources heavily and others only lightly.

Neighborhoods Given our general approach and the knowledge about the given instances, we developed four different neighborhood types for the optimization. The neighborhood selection is the heuristic part of our solution and thus crucial to balance the quality improvement and the computation effort. In our case different neighborhoods were used, but for all of them there is a small number of processes working on different machines for two neighboring solutions. Different neighborhoods can be beneficial for different situations of the optimization process, and they can provide different optimization possibilities. An experimental result that the mixture of different neighborhoods is better than each single neighborhood (in this case for our neighborhoods) is presented in Section 5.3.3.

For our search we used these four kinds of neighborhoods:

- Random search neighborhood: k processes are selected randomly from the set of all processes with equal probability or in the weighted case (weighted random search WRS_k) with a probability proportional to their contribution to the cost function (the possible load cost improvement if this process is deleted plus its move costs if the process was already moved). These processes then can be moved to any other machine to find neighboring solutions. The idea of the weighted random search is that neighborhoods with costly processes have a higher potential to improve the overall cost function.
- Process neighborhood search neighborhood (PNS): The processes are sorted with respect to their cost contribution (the possible load cost improvement if this process is deleted plus its move costs). Starting with the processes with the highest cost, we select 4 neighboring processes in the sorted order plus three randomly chosen processes. These processes then can be moved to any other machine to find neighboring solutions. The idea here is that heavy processes can switch machines for improvement, for example through a better fit for the demand of different resources.
- Target move search neighborhood (TMS): By looking at the given instances, we learned that there are big and costly processes that are difficult to move to other machines because these machines are already occupied with other processes. Thus we select up to seven processes on a designated target machine of the big process. Neighboring solutions are the solutions in which the large process is moved to the target machine, and the selected processes of the target machine are placed on any machine. The control flow of the TMS is illustrated in Figure 5.15. For the target move search neighborhood we first compute the load cost for each process, which is the load cost improvement on its current machine if this process is moved away, and build a process list (plist) according to this value. Starting with the process with the highest cost, we perform the following operations for all processes in order of decreasing load cost until we either find an overall solution improvement or

the time assigned to the target move search is up. For each machine which has enough resources for the big process (the list of these machines is computed in (mlist)) we randomly select 7 processes on this machine (or all processes if there are at most 7 on this machine) and put them into the neighborhood together with the big process. We also make sure that only solutions are accepted where the big process is placed on the selected machine (the number of fitting machines is usually small). Then we run the CP solver (cp). If no improving solution is found for a process, we continue with the process with the highest load cost among the remaining processes. If no improving solution is found for any process in plist and the time assigned to the target move search is not up yet, the target move search is just restarted. As the neighborhood selection contains some randomness (selection of the 7 processes) it is still possible to find an improving solution in the second walk through plist. The same holds for the process neighborhood search.

- Undo move search neighborhood (UMS): Especially in problem instances with transient resources it can be beneficial to move processes back to their initial machine. By doing this, it is possible to free transient resources and save movement costs. We select one moved process and five random processes that have been moved to its initial machine. These processes then can be moved to any other machine to find neighboring solutions.

Solution setup The different search neighborhoods are combined by running searches on each kind of neighborhood for some time. As the given machines have two cores we use two threads that combine the different strategies in a different way and exchange their best solutions from time to time. For our participation we used the following neighborhood scheme: Thread 1: TMS 5 seconds at a time, not started later than 45 seconds after program start; PNS 4 seconds at a time; WRS_7 4 seconds at a time, not started before 60 seconds after program start; UMS 1 second at a time; Thread 2: PNS 5 seconds at a time; TMS 5 seconds at a time, not started later than 60 seconds after program start; WRS_9 4 seconds at a time, not started before 60 seconds after program start; UMS 1 second at a time. The threads work through their list until the time is over. Every time when a new kind of search is started a thread compares its local solution with the best global solution and uses the better one to continue. After running one kind of search the local best solution is again compared with the global best solution and the global best solution is updated if necessary. We intentionally used slightly different strategy combinations and durations in order to gain benefits from the parallel optimization.

For instances with transient resources we apply a special process fixing in order to lower the burden on the transient resources. The fixing of the processes

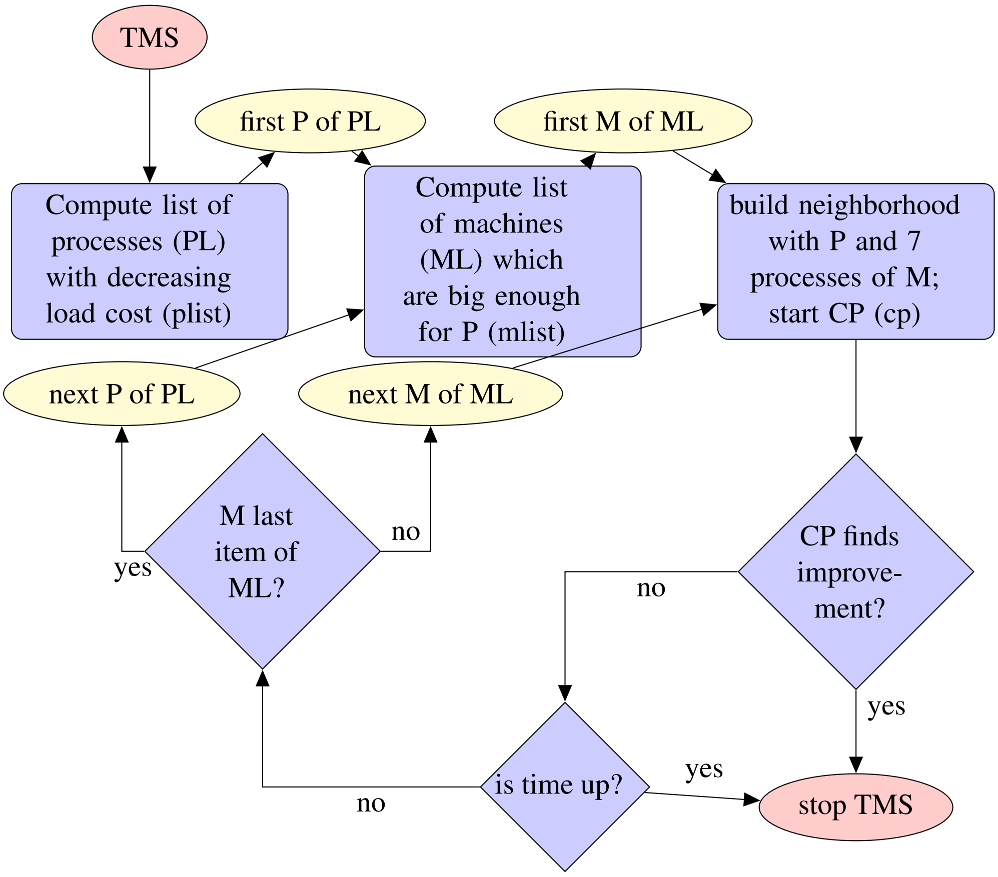

TMS
first P of PL
first M of ML
Compute list of processes (PL) with decreasing load cost (plist)
Compute list of machines (ML) which are big enough for P (mlist)
build neighborhood with P and 7 processes of M; start CP (cp)
next P of PL
next M of ML
M last item of ML?
yes
no
CP finds improvement?
no
yes
is time up?
no
yes
stop TMS

Figure 5.15. Control flow of the Target Move Search (TMS). The picture is taken from our article (joint work with Felix Brandt and Markus Völker [18]).

is done in two steps. First, the processes are ordered decreasingly with respect to their usage of the transient resource. If there is more than one transient resource, the demands for the transient resources are weighted by the inverse initial total free capacity of the respective resource. Second, starting from the beginning of the ordered list processes are fixed to their initial machine if the resource demand of fixed processes (including the process in question) is smaller than a fraction γ of the safety capacity for all resources on the respective initial machine. We used $\gamma = 0.9$ and released the fixed processes after 60 seconds from the beginning of the optimization process.

5.3.3 Results of the Challenge

After the presentation of our approach to the ROADEF/EURO Challenge 2012 in the previous section we now present some experimental results. First we present the results and descriptions of our own experiments which are mostly copied from the article about our solution approach [18] (joint work with Felix Brandt and Markus Völker). Compared to the article some experimental results are omitted. Then we give a brief description of our results in the competition and a short description of the approach of the solution of the competition winners.

Our experiments We implemented our CP-based solution approach in C++, using Gecode [130] in version 3.7.1 as CP-solver (our code is published, see https://github.com/fbrandt/ROADEF2012-J25). All experiments that are presented in the following have been performed on a computer with four 12-core AMD Opteron(tm) 6172 processors with 2.1 GHz and 256 GB of RAM. The underlying operating system was OpenSUSE 12.2 (64 bit). For each run of the multi-threaded experiments we used two of the 48 cores. For the single-threaded experiments only a single core was used.

For our experiments we used 20 test instances provided by the ROADEF/EURO challenge 2012. The instances are grouped into 2 sets, the b-instances already known during the development of our solver and the x-instances. The x-instances share the same basic characteristics as the b-instances but only became public after the contest had ended. Accordingly, our approach is not tuned towards the x-instances.

The score of the solution of an instance is defined as follows: all approaches were evaluated through the sum of $Score()$ over all instances. Given an instance and computed solution I, the best solution among competitors is B and the original reference solution R, the score is $Score(I) = 100 \cdot (C(I) - C(B))/C(R)$ where C is the cost function (also see above). All solution scores presented here are based on the best solutions submitted by any contestant of the ROADEF/EURO challenge 2012. Our approach did not find the best solution for any given instance, but we were often quite close.

Each experiment was repeated 30 times and all values presented in the following show the average of those 30 runs. In our evaluation, we mainly focus on solution scores. In comparison to the underlying solution costs the solution scores make it easier for the reader to see how our approach performs in comparison to the approaches of other contestants. In order to give an impression of the variance of the solutions, we look at the relative difference between the smallest and the largest solution cost of the 30 solutions produced by our approach which was handed in to the challenge. For most of the 20 instances the costliest solution produces a less than 1% higher cost than the best solution of our 30 runs. For the instances b_2, b_5, x_1, x_2 and x_7 our approach produces a difference of less than 3% between the largest and smallest solution cost of the 30 runs. For the instances b_1 and b_3 our solution produces a difference of about 6%. There are three instances which lead to a quite high relative difference between the best and worst solution value: x_3: 168%, x_5: 188% and x_8: 28%, but in these cases the cost of the reference solution is so much larger than the cost of our solution that the score differences are very small (x_3: < 0.08, x_5: < 0.0041 and x_8: < 0.00021). Altogether we computed an average solution score of 2.825. By using only the best solutions of our 30 runs, the score would have been 0.972, by using only the worst, it would have been 7.280.

Figure 5.16 gives an impression how our different neighborhoods improve the solution quality during the optimization time. Using only one neighborhood can be beneficial in the beginning, but in the end (after 5 minutes) our chosen combination of neighborhoods finds the largest improvements. Table 5.4 compares the different neighborhoods and our chosen mixture on all instances. The weighted random search strategies are the best single strategies, but our combination is clearly better than any used single strategy.

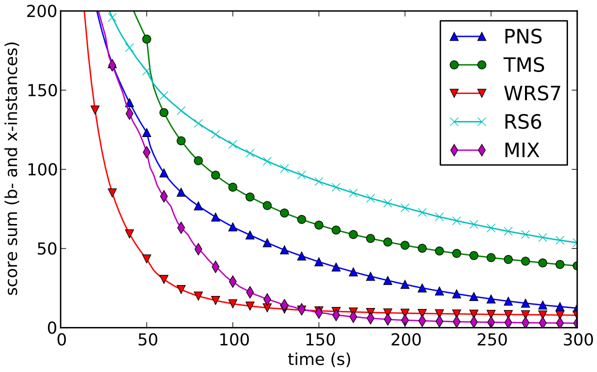


Figure 5.16. Behavior of different search strategies (neighborhoods) over time. MIX is our selected strategy combination. Graph taken from from our article (joint work with Felix Brandt and Markus Völker [18]).

We also take a look at the improvements through parallelization in Table 5.5. The results of thread 1 and thread 2 alone and their combination are given for a running time of five minutes. The parallelization results in a small improvement on nearly all instances.

Altogether our experiments show that strategy combination and parallelization improved the solution.

Comparison to other teams 82 teams registered for the qualification round (same restrictions and conditions but only some smaller problem instances). 48 teams handed in a program for the qualification round. 30 teams qualified for the final round, 28 of them handed in a solution program. Our team finished *second*

Instance	Score after 5 min						
	PNS	TMS	WRS$_7$	WRS$_9$	WRS$_{12}$	RS$_6$	Mix
b_1	1.559	5.423	2.088	2.015	1.962	3.877	1.082
b_2	0.620	0.203	0.593	0.566	0.601	1.208	0.306
b_3	0.176	3.308	0.181	0.207	0.233	0.152	0.106
b_4	0.001	0.125	0.001	0.001	0.000	0.002	0.001
b_5	0.216	6.541	0.206	0.231	0.262	0.105	0.139
b_6	0.000	0.057	0.000	0.000	0.000	0.000	0.000
b_7	0.470	1.705	0.119	0.116	0.112	7.399	0.088
b_8	0.228	4.865	0.129	0.150	0.265	0.011	0.034
b_9	0.011	0.669	0.009	0.028	0.238	0.016	0.008
b_10	1.702	2.266	0.276	0.321	0.458	12.410	0.105
x_1	2.167	6.012	2.550	2.435	1.840	6.437	0.248
x_2	0.641	0.314	0.681	0.560	0.523	1.115	0.307
x_3	0.120	0.738	0.259	0.255	0.234	0.160	0.075
x_4	0.001	0.038	0.001	0.001	0.001	0.002	0.001
x_5	0.003	0.249	0.139	0.144	0.159	0.010	0.002
x_6	0.000	0.050	0.000	0.000	0.000	0.000	0.000
x_7	2.356	2.608	0.360	0.323	0.347	8.431	0.205
x_8	0.000	0.008	0.002	0.002	0.001	0.003	0.000
x_9	0.015	0.818	0.013	0.032	0.328	0.017	0.012
x_10	2.093	3.063	0.280	0.301	0.398	12.296	0.107
Sum	12.380	39.058	7.887	7.687	7.962	53.650	2.825

Table 5.4. Comparison of the achieved solution scores after 5 minutes of computation for several neighborhood strategies. Table taken from from our article (joint work with Felix Brandt and Markus Völker [18]).

Instance	**Score after 5 min**		
	Thread 1	Thread 2	Thr. 1+2
b_1	1.795	0.933	1.082
b_2	0.347	0.373	0.306
b_3	0.128	0.134	0.106
b_4	0.001	0.001	0.001
b_5	0.150	0.181	0.139
b_6	0.000	0.000	0.000
b_7	0.155	0.130	0.088
b_8	0.075	0.142	0.034
b_9	0.025	0.024	0.008
b_10	0.206	0.162	0.105
x_1	0.439	0.663	0.248
x_2	0.389	0.363	0.307
x_3	0.087	0.087	0.075
x_4	0.001	0.001	0.001
x_5	0.003	0.003	0.002
x_6	0.000	0.000	0.000
x_7	0.315	0.267	0.205
x_8	0.001	0.000	0.000
x_9	0.038	0.031	0.012
x_10	0.219	0.194	0.107
Sum	4.374	3.688	2.825

Table 5.5. Influence of the parallelization on the achieved scores after 5 minutes running-time. Table taken from from our article (joint work with Felix Brandt and Markus Völker [18]).

in the final round among all teams in the junior category. 11 participating teams in the final round were junior teams. Teams were placed in the junior category if no team member had a Ph. D. yet. Our score of the challenge evaluation was 2.60, the best junior team scored 1.72, the scores of the places 3, 4 and 5 were 4.66, 4.95 and 10.66 [1]. Among all teams our team won the *fourth* place [1].

We also take a brief look at the methods used by the winning team (the senior team: Gavranović, Buljubašić and Demirović) as described by them [52]. They also took a neighborhood approach but used different neighborhoods (also 4). Two neighborhoods are pretty easy, the first is defined by just reassigning one process and the second by switching two processes. The third neighborhood is shifting a chain of processes such that each process is moved to the machine of the next process in the chain and the last one to the machine of the first process in the chain. The fourth neighborhood is very similar to our target move search. The set of processes that are allowed to move is increased during the running time starting with the largest process and adding smaller processes until all processes are considered. In contrast to our approach they seem not to use constraint programming or multithreading (at least nothing about this is described in their article [52]).

6

Scheduling in the Memory Hierarchy

Every application needs some memory to store its working set and of course it also needs to access this memory. As the computing cores became faster over time, their speed increases (in latency and bandwidth) were higher than the speed increases of memory (see Figure 2.1 in Section 2.2.1). Thus waiting for memory slowed down computation, and in order to mitigate this problem caches were introduced. But caches could not solve all problems regarding slow memory. The high latency (and low bandwidth) of memory was also a main reason for the movement towards parallel computing systems as performance improvements of sequential systems were hindered by the so called Memory-Wall (see Section 2.2.1). Today we have computing systems with many parallel cores which use a hierarchy of caches some shared, some exclusive. In case of multi-socket machines we additionally have even different local memories (see Section 2.2.2). As the reasons for parallelization and the complex memory hierarchy will remain in the future, it is likely that we will also have to deal with such systems in the future. The faster performance gain of computation units compared to the memory system will also lead to a higher importance of the latter regarding the overall system performance.

A lot of effort is put into the cache/memory hierarchy in order to build it and in order to use it efficiently. Borkar et al. [15] estimated in 2011 that 40% of the die size of current chips is used for caches. Also in software development there is much effort to use the memory hierarchy as efficiently as possible. There is a whole field of cache-oblivious algorithms (introduced by Friego et al. [48]) which use a given cache hierarchy optimally without parameter tuning. But of course scheduling is also important for the efficiency of the memory hierarchy. Memory bandwidth and shared caches are common resources which are important for a good schedule. Scheduling regarding those two common resources is important for application-internal schedulers and system-wide schedulers as well. Even applications like the LU-decomposition of dense matrices which are usually considered as computation-intensive can profit significantly from an application-internal scheduler which optimizes the memory access patterns (see our joint work with Peter Sanders and Tobias Maier [100] which is described in Section 6.2.2). In order to present our work about scheduling in the memory hierarchy, we first provide some basic measurements on recent machines in Section 6.1 to give an impression of the relevant performance values. Then we describe some approaches how to improve the efficiency of the memory system through scheduling in Section 6.2.

6.1 Properties of Memory Hierarchies

In order to give an overview of the properties of the memory system of current machines, we conduct experiments on the most recent server machines available at the group of Peter Sanders:

- *32-Core-Sandy-Bridge*: This four-socket machine uses 4 Intel Xeon CPU E5-4640 (Sandy Bridge microarchitecture) each running with 2.40 GHz. On each socket there are 8 cores which share 20 MB L3 cache. The 512 GB RAM consists of 32 16 GB DDR3-1600 DIMMs which are evenly distributed among the sockets (NUMA-architecture). The operating system is Ubuntu 14.04.4 LTS using the Linux kernel 3.13.0.
- *16-Core-Ivy-Bridge*: This two-socket machine uses 2 Intel Xeon CPU E5-2650 v2 (Ivy Bridge microarchitecture) each running with 2.60 GHz. On each socket there are 8 cores which share 20 MB L3 cache. The 128 GB RAM consists of 8 16 GB DDR3L-1600 DIMMs which are evenly distributed among the sockets (NUMA-architecture). The operating system is Ubuntu 14.04.4 LTS using the Linux kernel 3.13.0.
- *24-Core-Haswell*: This two-socket machine uses 2 Intel Xeon CPU E5-2670 v3 (Haswell microarchitecture) each running with 2.30 GHz. On each socket there are 12 cores which share 30 MB L3 cache. The 128 GB RAM consists of 8 16 GB DDR4-2133 DIMMs which are evenly distributed among the sockets (NUMA-architecture). The operating system is Ubuntu 14.04.4 LTS using the Linux kernel 3.13.0.
- *32-Core-Broadwell*: This two-socket machine uses 2 Intel Xeon CPU E5-2683 v4 (Broadwell microarchitecture) each running with 2.10 GHz. On each socket there are 16 cores which share 40 MB L3 cache. The 512 GB RAM consists of 16 32 GB DDR4-2400 DIMMs which are evenly distributed among the sockets (NUMA-architecture). The operating system is Ubuntu 14.04.4 LTS using the Linux kernel 3.13.0.

These machines represent the latest 4 Intel microarchitectures for server machines. All machines operate with enabled Intel Turbo Boost (this feature is usually turned on) which might lead to the effect that single-core operations are a little bit faster than multi-core operations as they run with a higher clock speed. The L1 and L2 caches are not shared among the cores and are of the same size for all cores of all machines (each core has an L2 cache of 256 KB and an L1 cache consisting of a 32 KB data- and 32 KB instruction-cache). The L3 cache is always shared among all cores on one socket and for each machine we have 2.5 MB L3 cache per core. These machines fit the description of typical parallel machines given in Section 2.2.2.

When we describe the performance development of memory (for example in Section 2.2.1), we look at two parameters: bandwidth (throughput) and latency. Accordingly we use two tests for our experiments, one focused on throughput and one on latency:

- *linear-test (lin)* This test reads through an array. In real applications this can happen when an application sums up the elements of an array or scans an array for a special element. The important thing is that the next element to be read is adjacent to the current element and its address can be computed without having to look at the content of previously read elements. This test should be able to reach the possible memory bandwidth of the core/socket.
- *chain-test (ket)* This test follows an index chain in an array. A real application can produce a similar access pattern if it iterates through a linked list or an index chain or permutation within an array. The important point of this test is that in order to compute the address of the next element to be read the value of the current element is needed. Hence for each step the test has to wait for the full latency of one memory access. This test should give a good impression of the access latency of memory.

We restrict ourselves to reading tests as reading is a more common operation than writing and we expect no major additional insights by writing tests as we only test on independent memory locations for each core.

Each test consists of two parts: preparation and execution. The tests are given two parameters, the size of the array to work on and the number of times the array has to be read. For the *linear-test* the preparation is allocating an array of the given size and writing a long integer (8 byte) at each cell. The execution of the linear-test iterates over the array (the number of times is given) and sums the integers up. The loop computes 8 additions per iteration. The *chain-test* allocates an array of long integers (8 byte each) of the given size and prepares a pseudo-random cyclic permutation within the array. Each cell contains the index of the next cell in the permutation. The permutation is generated by Sattolo's algorithm [118] and a linear congruential generator with $x_{n+1} =$

$x_n \cdot 6364136223846793005 + 1442695040888963407 \mod 2^{64}$ (the numbers are often attributed to Donald Knuth). The chain-test then iterates over the array by $k = \text{array}[k]$ the number of times given. The loop contains 8 such steps. We developed the chain-test by using an idea (follow a closed permutation in an integer array) of how to measure memory latency introduced by Timo Bingmann (another Ph.D. student at the chair of Peter Sanders) in a discussion about that topic. The threads that execute these tests are pinned to their respective cores and perform the memory allocation after they have been pinned. This usually leads to a placement of the allocated array on the same NUMA node where the core is located. In order to ease synchronization, all participating threads are part of the same process. The threads perform the reading tests in a synchronized way that makes sure that each measured test always runs in parallel to the other tests given at the same time. This is done by synchronization and eventually running some unmeasured tests before and after the measured tests. The test procedure of a thread/core is done as follows: first the memory is allocated and prepared, and then an unmeasured test run is started; after the unmeasured test run the thread checks if all other participating cores have finished their preparation, and if this is the case, then the measured test runs are started, and if not, unmeasured test runs are done until the condition is fulfilled; after the measured test runs the core performs unmeasured test runs until all participating cores have finished their measured test runs. The test program is written in C++ and compiled by gcc 4.8.4 using the option -O3.

Timo Bingmann also built a parallel memory benchmark pmbw [10]. The focus of his benchmark is application development rather than scheduling. He provides a wide range of reading and writing benchmarks of memory and different cache levels on different machines. There are also many different low level optimizations compared to each other. One of the differences is the way how the memory is allocated. This leads to some different results, but the differences are too small to change the main implications of this section. Also pmbw offers no possibility to run different kinds of memory access patterns in parallel. We also do some measurements with the STREAM benchmark by McCalpin [103]. We compare these measurements with our basic experiment below.

Basic experiment: The first check tests the behavior of the machines when only one core is used (seq) or when all cores are used. We test different array sizes: 1 MiB, 2 MiB, 10 MiB, 20 MiB, 100 MiB, 200 MiB, 1000 MiB in order to get a feeling of the influence of different sizes especially regarding the cache. We read the same data amount from each array size by adjusting the number of times the array is read. For the chain-test we read 1000 MiB for each measurement and for the linear-test we read 100 000 MiB per measurement as it is much faster. When the measurement program is started, three measurements are performed on the same array. Additionally we start the program 5 times with the same measurement which leads eventually to different memory allocations. Hence we have 15 measurements for each participating core for each experiment. Between the termination of an instance of the measurement program and the next start we let 60 seconds pass to allow the machine to cool down. For the sequential experiments we always use core 0. For the parallel experiments we use all cores, but we also look at the running times/bandwidth of core 0. All of our own tests only use one thread per core, ignoring the available virtual HT-cores (Hyper-Threading Technology). We compute the bandwidth by dividing the amount of read data in MiB (1 000 or 100 000) through the measured running time of the experiment and 1 024 (in order to get GiB/s). The Table 6.1 reports the reached bandwidth of core 0 computed with the average running time (15 measurements). All tests are done on otherwise idle machines. The test abbreviations are ket seq for the chain-test only running on core 0, lin seq for the linear-test only running on core 0, and ket all and lin all for the respective tests running on all cores. The column headers are the array sizes in MiB. We omit to report the measurement errors because the difference between the largest and smallest measurement are quite low. On all 4 machines, the relative difference between the smallest and largest of the 15 measurements (for core 0) is less than 2% except for three experiments on the 24-Core-Haswell where the relative difference is bounded by 4%. In case of the experiments on all cores (all-experiments) we also look at the variance of the averages (of the 15 measurements) between different cores. On 32-Core-Sandy-Bridge and 16-Core-Ivy-Bridge we can bound the relative difference between the averages of different cores by 3.5%. On 24-Core-Haswell and 32-Core-Broadwell the cores seem to vary more, only 9 of 28 experiments have relative differences between the cores of less than 3.5%. In 15 of 28 all-experiments on these machines the variance between the cores was more than 4% but less than 9%. On the 32-Core-Broadwell we even have four experiments (lin all and ket all on 1 MiB and 2 MiB) which produce relative differences between 13% and 16% between the different cores. Hence we observe larger differences between the cores on the newer machines, and the differences between different cores are usually larger than the differences between different measurements on the same core.

First we take a look at the results of the sequential chain-test (ket seq) in Table 6.1. On all machines we can see a performance drop of about a factor of 4 between the test on the 20 MiB array and the 100 MiB array and only smaller changes between the other array sizes. This is very likely an effect of the cache, as the L3 cache sizes per socket are between 20 MB and 40 MB for all tested machines. Hence in case of the 20 MiB array the major part will fit into the cache as opposed to the 100 MiB array where only a minor part fits into the cache. If we assume that the running time of the chain-test completely comes from the memory latency (the time between giving a load command and receiving the value), we can

	1	2	10	20	100	200	1 000
ket seq	0.542	0.573	0.536	0.493	0.098	0.088	0.082
ket all	0.479	0.506	0.098	0.088	0.079	0.077	0.075
lin seq	22.2	22.5	22.5	16.4	13.0	13.0	13.0
lin all	19.3	19.5	3.68	3.68	3.68	3.68	3.68

(a) 32-Core-Sandy-Bridge

	1	2	10	20	100	200	1 000
ket seq	0.650	0.686	0.641	0.556	0.113	0.102	0.095
ket all	0.571	0.602	0.113	0.102	0.092	0.090	0.088
lin seq	28.5	28.9	28.9	23.0	15.7	15.7	15.7
lin all	24.7	25.1	5.62	5.61	5.61	5.61	5.61

(b) 16-Core-Ivy-Bridge

	1	2	10	20	100	200	1 000
ket seq	0.489	0.505	0.469	0.465	0.117	0.102	0.093
ket all	0.444	0.466	0.109	0.099	0.090	0.089	0.087
lin seq	28.4	28.6	28.7	28.7	12.4	12.5	12.4
lin all	24.4	24.4	4.69	4.68	4.67	4.67	4.67

(c) 24-Core-Haswell

	1	2	10	20	100	200	1 000
ket seq	0.420	0.425	0.394	0.390	0.119	0.098	0.087
ket all	0.393	0.404	0.101	0.091	0.084	0.082	0.080
lin seq	29.2	29.3	29.4	29.4	11.9	11.9	11.8
lin all	23.6	23.7	3.69	3.70	3.69	3.70	3.70

(d) 32-Core-Broadwell

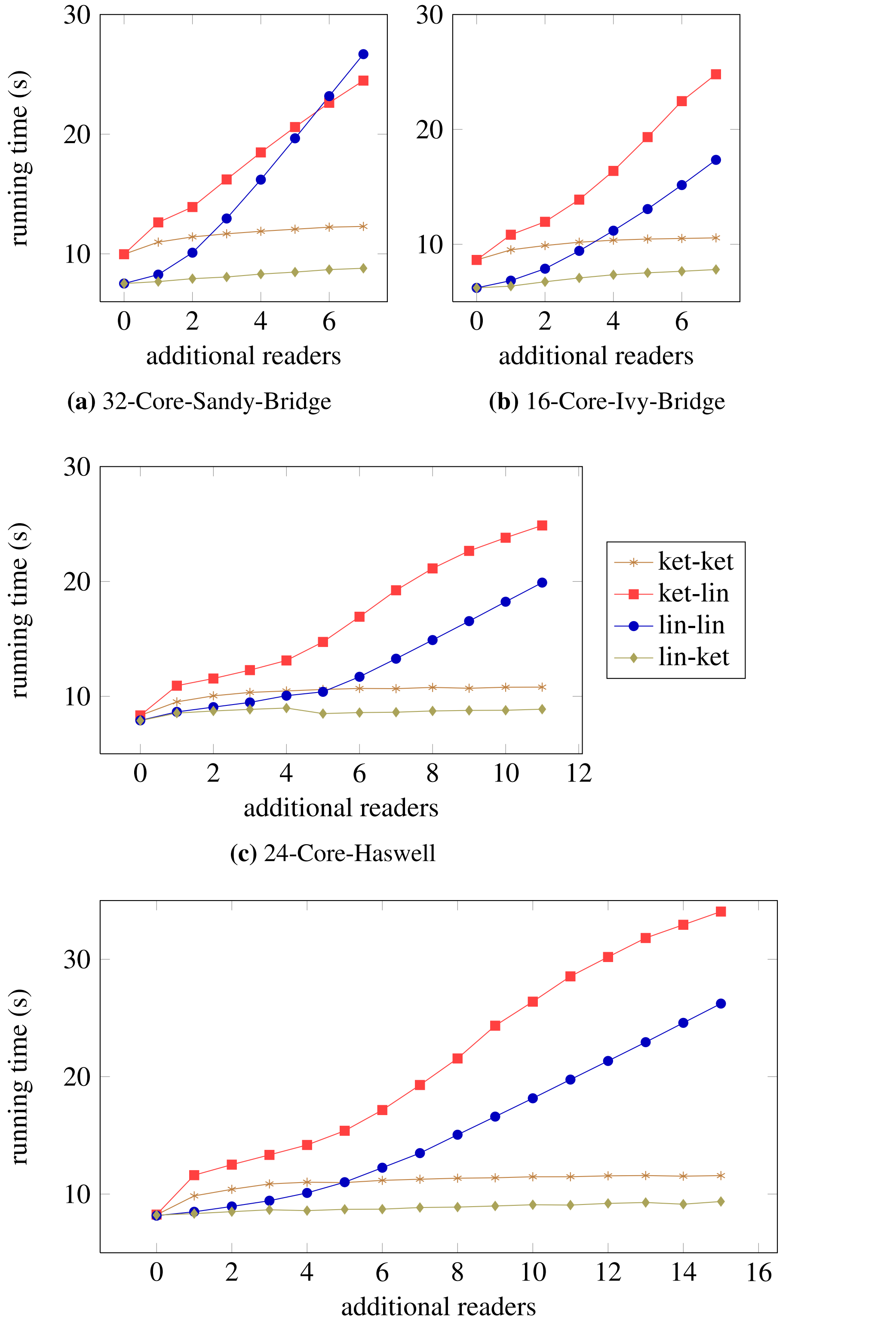
Table 6.1. The chain-test on a single core or all cores (ket seq, ket all) and the linear-test on a single core or all cores (lin seq, lin all). The reported values are bandwidths in GiB/s. The column headers are the array sizes in MiB.

compute the latency. For a bandwidth of 0.5 GiB/s in the chain test (similar to the measured values when reading from the cache), the latency is $\approx$ 14.9 ns as we load 8-byte values. For a bandwidth of 0.1 GiB/s (similar to the values when reading from arrays which do not fit into the cache), the latency is $\approx$ 74.5 ns. This corresponds to about 30 or 150 clock cycles on a 2 GHz core. In case of all cores performing the chain-test the situation changes a bit. The bandwidths for the small array sizes are a bit reduced, in parts probably due to an automatically reduced clock speed as all cores are working. We also see the sharp drop of bandwidths now between the 2 MiB and the 10 MiB column. This is plausible as all machines have 2.5 MB of L3-cache per core. Hence all the 2 MiB arrays fit into the last level cache, but the 10 MiB arrays do not.

Now we look at the results of the linear-test. In the sequential case we also (like in the chain-test) have a performance drop between the 20 MiB array and the 100 MiB array. The performance drop is probably also due to the fact that the 20 MiB array fits into the L3-cache and the 100 MiB array does not. The penalty of reading data from outside of the cache is smaller than in the chain-test, we see only about a factor of 2. The bandwidths of the linear-test are high, the common value of 28 GiB/s means that $\approx$ 3.8 billion $(3.8 \cdot 10^9)$ 8-byte elements are read per second which is significantly more than one element per clock cycle of the used cores. Even when reading from RAM, a core on average gets more than one array element every two clock cycles. This shows how well optimized the cache prefetching and other systems are in case that a program reads linearly through an array. In the parallel case the cache is only sufficient for the 1 MiB and 2 MiB arrays for the same reasons as described with the chain-test. The linear-test is also a little bit slower when running on all cores than when running on only one core even when all accesses can be satisfied from the cache. This is probably also due to the clock speed reduction when all cores work at the same time. A rather sharp performance drop occurs between the 2 MiB and the 10 MiB column for the linear-test on all cores. The measured bandwidth for the array sizes of 10 MiB and above drops to about one fifth of the bandwidth for the 2 MiB array size on all machines. The drop is much more severe than in case of the sequential linear-test. Hence the reason cannot only come from the fact that the data is not read from the cache any more, but there must be something within the connection to the memory that reaches its maximal capacity. Thus it is likely that the linear-test on all cores reaches the maximal memory bandwidth (at least for the kind of accesses used in our tests). The slowdown experiment (see below) looks at the behavior of memory accesses when the memory bandwidth is pushed towards its maximum. Another interesting observation of the basic experiment described in Table 6.1 is that there is no major improvement of neither memory bandwidth (per core) nor memory latency over the last four generations of server processors from Intel, although the total memory bandwidth of a socket is increased as the newer machines use more cores per socket. This even holds for the sequential tests. Of course this observation is only rough as the four machines included in this test set are different when regarding clock speed, cores per socket, and even the number of sockets differs.

In order to support our measurements, we also look at other benchmarks which test similar things at least in parts. One often used memory benchmark is the STREAM benchmark by McCalpin [103]. In contrast to our tests STREAM uses more complicated operations (copy, scale, add, triad) that include writing to memory, reads an array only once per measurement, does not use loop unrolling and is parallelized via OpenMP. The OpenMP overhead is part of the time measured section opposed to our tests where parallelization and synchronization are always done outside of the time measurement. We run STREAM with *OMP_NUM_THREADS=1* and a number of threads which is the number of cores including the virtual HT-cores as otherwise some cores remain unused. On 32-Core-Sandy-Bridge STREAM copy on one core measures a bandwidth of 10.6 GiB/s, and on all cores the copy bandwidth is measured as 76,1 GiB/s. Especially for the parallel case it is likely that the measurement includes memory accesses across different NUMA nodes. Our measurements on this machine are 13.0 GiB/s in the sequential case and 3.68 GiB/s on core 0 in the parallel case ($3.68 \cdot 32 = 117.67$). Hence we see some differences of our measurements compared to STREAM, but these differences seem to be explainable by the differences of the tests. The other tested machines show similar differences, STREAM is about 20% slower in the sequential case and about 40% slower in the parallel case.

Slowdown experiment: As we can see in Table 6.1, the memory accesses are slowed down when other cores also run bandwidth-demanding tests. In this experiment we take a closer look at this phenomenon. As we know that each socket has its own memory connection on all tested machines, we can restrict ourselves to tests on one socket (this assumption is also documented by experiments, see below). Figure 6.1 shows the running times of a test running on core 0 (linear-test or chain-test) when different numbers of other tests are run on the same socket at the same time. All the tests in this experiment are run with an array size of 100 MiB as this size is sufficient to force the usage of main memory because the caches are always smaller. Hence this experiment is a measurement of the behavior of the connection between the processor and the memory. In the legend we always denote first which test runs on core 0 (first test) and second which tests run on the other cores in order to slow down the test on core 0 (second tests). The data amount read per core is the same as in the basic experiment, 1 000 MiB for the chain-test and 100 000 MiB for the linear test. Like in the basic experiment we



(a) 32-Core-Sandy-Bridge

(b) 16-Core-Ivy-Bridge

(c) 24-Core-Haswell

(d) 32-Core-Broadwell

Figure 6.1. Running time of the tests on 100 MiB arrays depending on the number of parallel tests on the same socket. The first abbreviation in the legend denotes the measured test, the second abbreviation denotes the kind of tests on the other cores.

perform three measurements per one program run (and one array allocation). The program is started five times with the same combination of tests. This results in 15 measurements of the running time of the test on core 0 for each combination of tests. The plots in Figure 6.1 report the average of these 15 measurements. Also like in the basic experiment we let 60 seconds pass between two program runs.

The numbers used for Figure 6.1 are produced by test combinations which leave all but the first socket (NUMA-node 0) of the machine idle. The different numbers of other tests come from the fact that the different machines have different numbers of cores on their sockets. We also investigated the slowdown when all cores on all other sockets run the second test. This means that we did every reported test twice: 15 measurements with the respective number of second tests on the same socket and 15 measurements with the respective number of second tests on the same socket plus all cores on other sockets also running the second test. The difference between these two kinds of test combinations is low, less than 3% in nearly all tests. Only two test combinations on 24-Core-Haswell produced a bigger difference: for lin-lin and lin-ket with 11 second tests on the first socket the first test (lin in this case) is 5.1% or respectively 5.7% slower if all 12 cores on the second socket perform the second test instead of being idle. Altogether this shows that the work on other sockets has little influence on the memory performance of the first socket as long as each core works on locally allocated memory. Hence we omit the test results with tests on other sockets.

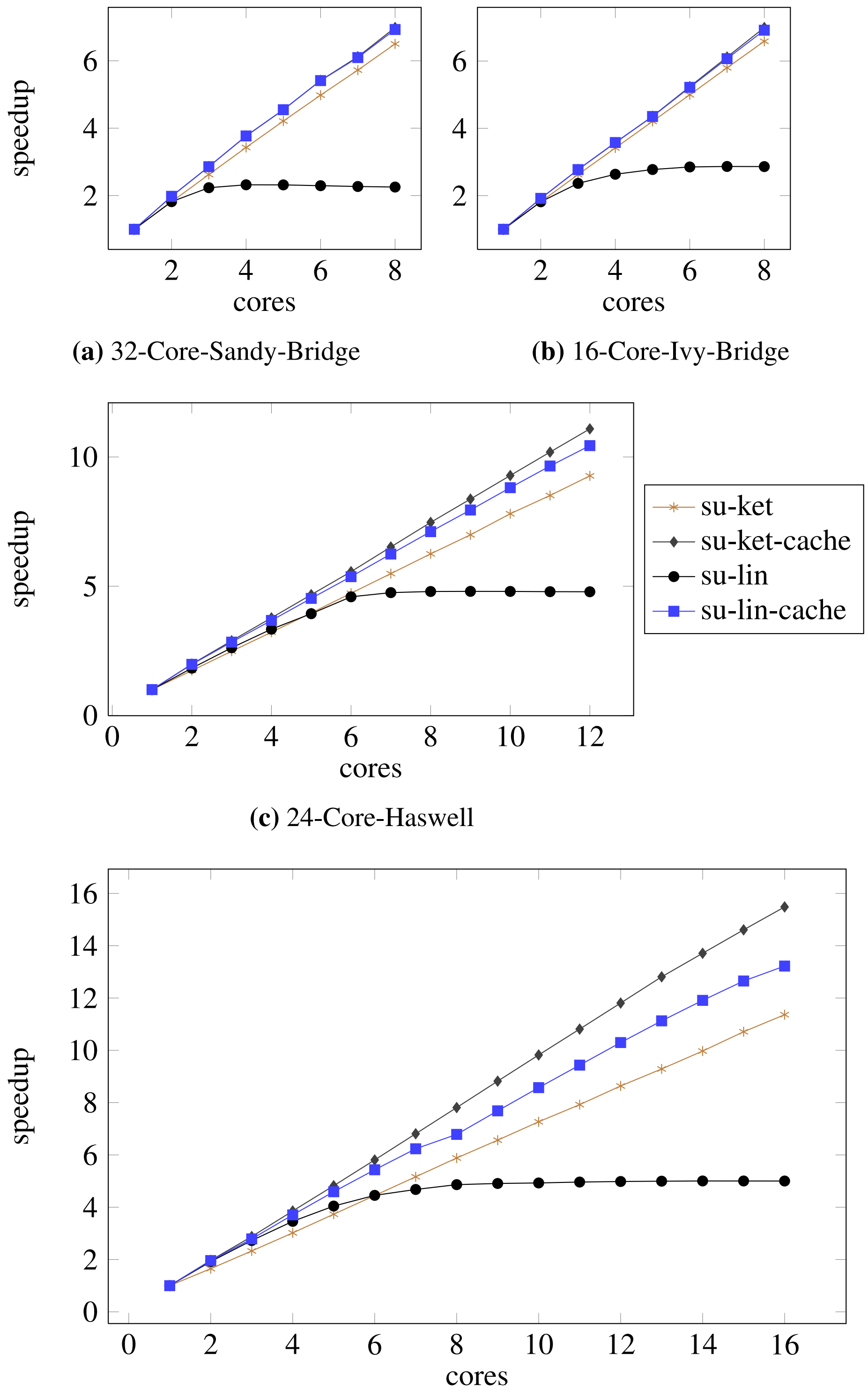
Let us now take a look at the measurement errors of this test. Like in the basic experiment we look at the relative difference between the largest and smallest measurement of the reported test combinations. On 32-Core-Sandy-Bridge and 16-Core-Ivy-Bridge all relative differences of the running times of the test on core 0 are below 3%. On 24-Core-Haswell and 32-Core-Broadwell all but three test combinations on 32-Core-Broadwell produce relative errors below 5%. The three test combinations and their errors are: ket-ket with 3 second tests (7.1%), lin-lin with 0 second tests (5.4%) and lin-ket with 14 second tests (5.1%).

The main result of this experiment is that the heavy memory bandwidth usage of the linear-tests slows down other memory accesses of cores on the same socket. Except for the 32-Core-Sandy-Bridge, the chain-test has larger performance losses than the linear-test. But in all cases the slowdown is severe. If we look at the test combinations in which all cores except 0 run the linear-test compared to the combinations in which core 0 is the only running core on the first socket, we get the following factors for the running time: 32-Core-Sandy-Bridge ket-lin: 2.46, lin-lin: 3.55, 16-Core-Ivy-Bridge: ket-lin: 2.87, lin-lin: 2.80, 24-Core-Haswell: ket-lin: 2.98, lin-lin: 2.52, 32-Core-Broadwell: ket-lin: 4.13, lin-lin: 3.21. The performance losses due to the chain-test as second test are much smaller (between 1.14 and 1.41 on all machines). Hence both available bandwidth and latency are heavily affected by a high memory bandwidth usage. The chain-test uses far less bandwidth and thus produces much smaller slowdowns.

The slowdown of the chain-test by linear-tests on the same core is also not affected by the condition if both tests run within the same process or not. Running some of the tests by hand synchronizing in parallel but within different processes produced no differences larger than the reported measurement errors on all machines. By testing with 3 and 7 linear-tests on the machines with 8 cores per socket we found differences of less than 2% to the average of the measurements within one process. On 24-Core-Haswell the test with 5 and 11 slowdown tests and on 32-Core-Broadwell the test with 7 and 15 slowdown tests showed no differences larger than 2% to the average results of the tests within one process.

Speedup experiment: This experiment uses the same kind of measurements as the slowdown experiment, chain-test slowed down by chain-tests and linear-test slowed down by linear-tests but on two different array sizes. This time we do not look at the slowdown of the test running on core 0. Instead we look at the possible performance gains by using additional cores even if they reside on the same socket. In order to do this, we compare the bandwidth reached by a single core with the sum of the bandwidths of several cores. The sum of bandwidths divided by the bandwidth of the single core 0 will be called speedup in this experiment. As we measure running times we compute in fact the average running time of core 0 (while running alone) divided by the average running time of a participating core and sum these values over all participating cores. We test on the array sizes of 1 MiB and 100 MiB in order to have tests which can be served by the cache and others which have to use the RAM. The resulting speedups on different numbers of cores are depicted in Figure 6.2 for our different machines. We restrict our tests to one socket as we have seen in the previous experiment that the different sockets do not influence each other much in our setting.

The test setting regarding measurements, loaded data and waiting times is the same as in the previous experiment. We look at the relative errors of the tests on each participating core. On 32-Core-Sandy-Bridge and 16-Core-Ivy-Bridge the relative difference between different measurements on the same core with the same test combination are below 1%. On 24-Core-Haswell the maximal relative error for all reported tests is 4.6%, on 32-Core-Broadwell the maximal error is 7.1%. The errors in this experiment can be higher than in the previous experiment as we look at the measurement errors of all participating cores instead of only those of core 0. We also take a look at the variance of the averages of different cores. The maximal relative differences on the tested machines are: 32-Core-Sandy-Bridge: 3.3%, 16-Core-Ivy-Bridge: 2.2%, 24-Core-Haswell: 6.4%, 32-Core-Broadwell: 13.2%. The variance between the different cores cannot be regarded as measurement errors of our measurements as we sum the bandwidths



(a) 32-Core-Sandy-Bridge

(b) 16-Core-Ivy-Bridge

(c) 24-Core-Haswell

(d) 32-Core-Broadwell

Figure 6.2. Speedup of memory operations by using different amounts of cores.

of the different cores, and thus different cores just contribute different bandwidths.

When we look at the results of this experiment (see Figure 6.2) we can see that the test combinations that can be done within the cache show a nearly optimal linear speedup. The speedup of the chain-test on 100 MiB arrays is also nearly linear but below the speedup of the tests on 1 MiB. The only test behaving totally different is the linear-test on the 100 MiB arrays. On all machines we can observe some kind of limit that can be reached by using a relatively small number of cores, but it cannot be increased by adding further cores. This indicates that the common resource of memory bandwidth is fully utilized at some point at which adding further cores brings no performance improvement. The speedup experiment is also an example for concave speedup functions.

	node 0	node 1
ket seq	0.098	0.027
ket all	0.080	0.021
lin seq	13.1	2.45
lin all	3.66	0.36

(a) 32-Core-Sandy-Bridge

	node 0	node 1
ket seq	0.113	0.050
ket all	0.092	0.044
lin seq	15.8	7.21
lin all	5.63	2.94

(b) 16-Core-Ivy-Bridge

	node 0	node 1
ket seq	0.117	0.072
ket all	0.090	0.054
lin seq	12.4	8.96
lin all	4.91	1.49

(c) 24-Core-Haswell

	node 0	node 1
ket seq	0.119	0.071
ket all	0.084	0.049
lin seq	11.8	7.79
lin all	3.72	1.77

(d) 32-Core-Broadwell

Table 6.2. Bandwidth of core 0 (residing on node 0) in GiB/s for the four different measurements of the basic experiment on 100 MiB. The first column reports the results if all participating cores allocate their array on node 0, the second if all arrays are allocated on node 1.

NUMA experiment: Another important question is what happens if the memory accesses are not happening locally but on another NUMA-node. For all other experiments each thread is pinned on its own core and allocates the array for its experiment by calling *new* what usually results in a memory allocation on its own NUMA-node (the NUMA-node of its own core). For this experiment we simply change the memory allocation. Instead of *new* we use *numa_alloc_onnode*. The only participating cores in this experiment are the cores on NUMA-node 0 (or on the first socket). For these cores we run the test combinations of the basic experiment for an array size of 100 MiB in order to prevent major cache effects. Each test combination (single core linear-test and chain-test and all cores linear-test and chain-test) is run twice, first with the arrays allocated on NUMA-node 0 and second with the arrays allocated on NUMA-node 1.

Once again each test combination is tested in five program runs and during each program run three measurements are taken. We report the average bandwidth of these 15 measurements for core 0 in Table 6.2. The relative difference between the smallest and the largest bandwidth is below 3% for all test combinations.

As expected the results are very similar to the basic experiment when the arrays are allocated on NUMA-node 0. If the arrays are allocated on NUMA-node 1, we observe a bandwidth drop of about one half on all two-socket machines (16-Core-Ivy-Bridge, 24-Core-Haswell and 32-Core-Broadwell). On the four-socket machine (32-Core-Sandy-Bridge) the performance drops are much more severe when the allocation is done on NUMA-node 1.

Main implications The main implications of these experiments can be summarized by:

1. Especially for latency-dependent jobs or in case of an already highly utilized memory connection using cache instead of main memory brings high performance benefits.
2. Bandwidth consuming jobs induce a severe performance penalty for other jobs on the same socket if these are latency-dependent or have high bandwidth needs themselves.
3. If several jobs are using the same last level cache, the sharing of the cache space can lead to severe slowdowns if the working set of a job no longer fits into cache.
4. If memory bandwidth is the bottleneck, adding more cores does not improve the performance.
5. Memory accesses on different sockets are independent from each other if socket-local data is accessed.
6. Accessing data from RAM of another NUMA node is slower than using the RAM of the own node.

In the context of this work not only the performance of the memory system is interesting but also the energy consumption. We provide some energy measurements for cache and memory access in Section 7.2.

6.2 Memory Scheduling

With the results of Section 6.1 we can now look at the possibilities of improving the memory efficiency by scheduling. First of all the main implications from Section 6.1 show that jobs using the same memory connection or last level cache can slow each other down severely. Hence there are relevant interdependencies between different application-internal schedulers and the OS scheduler. The memory bandwidth and last level cache can be seen as common resources of all jobs running on one socket. As the usage of these common resources is highly relevant for performance and efficiency, the scheduling system should coordinate the usage of them.

Of course scheduling cannot change the number of memory accesses, the locality (in address and time) or the general memory access pattern of a thread or an application. This falls in the responsibility of the algorithm or application development. Additionally, on most current machines it is not possible to manage the cache and bandwidth usage directly. A notable development to overcome this problem is Intel's Cache Allocation Technology (as described in a white paper from 2015 [71]). As long as there are no measures to distribute cache space and memory bandwidth directly, the memory-aware scheduling on the OS level has to rely on the placement of threads (regarding core and time), especially regarding the NUMA node and the common usage of L3 caches and memory connections. The application-internal schedulers can also change the allocation of workpackages (or other internal computing tasks) to threads in order to improve the memory efficiency. But there is usually no knowledge about future memory accesses of the application, especially for the OS scheduler. Also the potential benefits or losses regarding the goal function (if applications can use more or less memory bandwidth or cache space) and their dependency on memory latency are usually unknown. Hence memory scheduling in real cases is currently limited to application-internal schedulers and rather simple heuristics for the system scheduler. For example a simple heuristic for the system scheduler is to prefer to schedule a thread always to the same core in order to reuse already loaded cache items. The lack of information of the system scheduler might be overcome by enhanced interfaces as described in Chapter 4 or by observation techniques that can predict the future memory usage of threads.

Altogether the decision space in memory scheduling is limited by a lack of control mechanisms and decisions already made during application development which fix access patterns. In case of the system scheduler the available information is even more limited. On the other hand the measurements from Section 6.1 indicate a high performance/efficiency relevance of the memory system at least for some kinds of jobs. Hence information and decision space still have to grow for an efficient memory scheduling (at least for the system-level scheduler).

In contrast to scheduling practice where some publications about scheduling (especially regarding memory properties and performance) exist (see Section 3.2.2), memory and cache behavior are usually not included in the algorithms and models of scheduling theory. Something in scheduling theory that can be seen as related to scheduling in the memory hierarchy is the scheduling of communication between tasks. An often used model which is also used by Kwok and Ahmad in one of their DAG-scheduling overview articles [89] is described now:

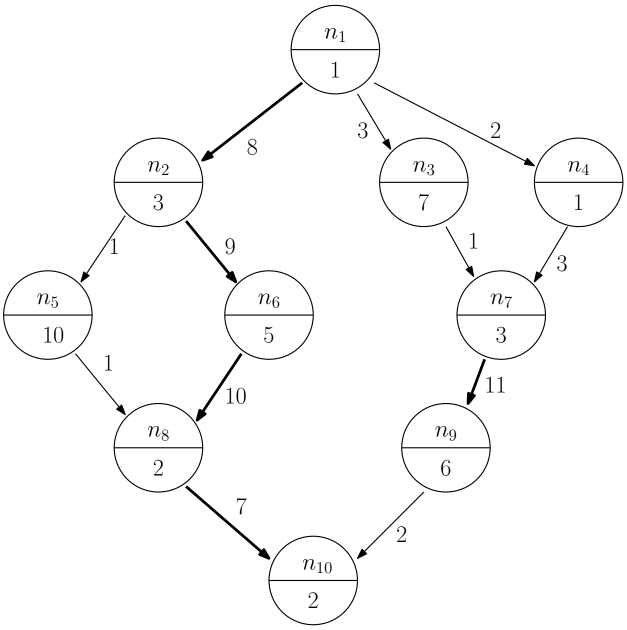


Figure 6.3. A task-DAG fitting the model from Kwok and Ahmad. The n_i denote the node IDs, the number below the weight of the node and the numbers next to the edges the edge weights.

Each node and edge in the task-DAG has a weight. The node weight is the computation cost of the corresponding job. The edge weight of an edge is called the communication cost which is only incurred when the incident nodes are scheduled on different processors. If we call the cores themselves or all cores sharing the same last level cache a processor, a DAG-scheduling algorithm which minimizes the communication cost can also be used as a heuristic in order to get a high reusage of cache items. If processors are NUMA nodes and the information transmission is realized through newly allocated memory parts, such a communication-efficient DAG-schedule reduces the non-local memory accesses. In both cases the DAG-scheduling model with communication cost is no perfect fit. Things are ignored that are important in real applications like inputs that are no outputs of other tasks or cores belonging to the same NUMA node or sharing the same cache. Especially for cache-efficient scheduling, the space limitations are not regarded in this DAG-scheduling model.

Own approach: As there is no widely accepted theoretical model for memory scheduling, we concentrated our research to memory scheduling for some problems relevant in practice similar to most of the publications from Section 3.2.2. We hope that the findings and used strategies for these easy problems might serve as components for a useful model for memory scheduling in the future. As described in Section 3.4.2 experiments on the full abstraction hierarchy are very difficult and expensive. Hence we restrict ourselves to problems of application-internal schedulers which are easier to evaluate and where experimental evaluation is not too costly. With application-internal schedulers, we circumvent problems with information-blocking interfaces and get additional decision space. Also working with only one application keeps the programming effort low and eases adaptions of the applications to the used scheduling strategies. The two most important common resources regarding the memory system are cache space and memory bandwidth. Both are distributed between the different threads of the applications which run on the same socket. We take a look at a problem for each resource, depicted in Section 6.2.1 and Section 6.2.2. For both problems we also improved the NUMA-locality of the memory requests.

6.2.1 Memory Bandwidth Distribution

In this section we take a look at problems in which the memory bandwidth is the resource which limits the overall performance or is at least a resource which has a big influence on the performance. If memory bandwidth is the bottleneck of our application or our set of applications, it is an obvious approach that in order to get the best performance the bandwidth should be used fully at all times so that nothing is left unused. More often there are situations where two resources play an important role: memory bandwidth and computation. A model for such situations (especially for the case of floating-point-intensive program kernels) is the Roofline-model introduced by Williams et al. [141]. In this model, programs (or important kernels) are characterized by their operational intensity which is the number of floating-point operations these kernels perform on average per loaded byte from memory. A machine has two main properties in this model: the bandwidth to the main memory and its peak floating-point performance. For our example graph (Figure 6.4) we assume a machine with a memory bandwidth of 100 GB/s and a peak floating-point performance of 800 GFlop/s. The blue line is entirely machine-dependent. The maximal performance of a job is then the height of the intersection between the vertical line through its operational intensity and the line characterizing the machine. In this example, job 2 has such a high operational intensity that it can reach the machine's peak floating-point performance whereas the operational intensity of job 1 is lower such that it is bandwidth-bound.

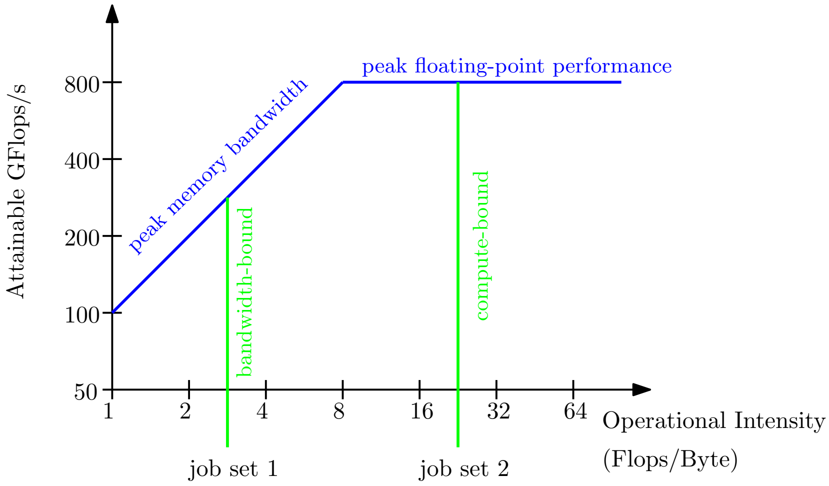


Figure 6.4. An example graph for the Roofline-model. The position of the peak memory bandwidth and peak floating-point performance lines is machine-specific. Job 1 is an example of a bandwidth-bound job, job 2 an example of a compute-bound job.

In their paper [141] Williams et al. also look at optimizations which are not important for our scheduling perspective. Although the Roofline-model is designed with floating-point-intensive kernels in mind, it might also be a good fit for other applications that use for example integer computations. If we have different jobs which have different characteristics regarding their relation between bandwidth usage and the usage of computation units, it is quite natural to ask how to schedule such jobs efficiently. Slow memory (latencies) is even a main reason for parallelization (see Section 2.2.1 Memory-Wall).

Basic idea: A job put into the Roofline-model is implicitly meant to use all available cores, otherwise no such job could reach the peak floating-point performance of the machine. If there are different jobs, some of them compute-bound, some of them (memory) bandwidth-bound, the idea is not to run the jobs one after another at the full parallelism of the machine, but to run jobs in parallel by running each job with a smaller degree of parallelism. This might help to use the computation and memory access capabilities of the machine better. For example if we have a job 1 which is bandwidth-bound and a job 2 which is compute-bound, we might run both jobs on half of the cores on each socket. We ignore parallelization overheads for this example. If this combination does not use the full memory bandwidth, then job 2 runs with half of its speed on all cores, but job 1 runs with more than half of its speed on all cores because it is not bandwidth-bound any more. In this case the combination uses the full computing capability of the machine and a higher bandwidth than job 2. If the combination uses all of the memory bandwidth, then job 2 might get slower than one half of its speed on all cores, but job 1 still is faster than one half because it gets more than half of the total bandwidth. In this case the combination uses all of the available bandwidth and more of the computation capabilities than job 1 alone. In both cases the resource which is in higher demand is always used fully in contrast to running both jobs one after another where both resources are not used fully for one job. It is also clear that the operational intensity of the combination is between the intensities of job 1 and job 2. Altogether the machine usage is improved and the efficiency increased. Of course the distribution of the available cores on the sockets has to fit the workloads of the participating jobs, but it is clear that job combinations have a potential to increase efficiency in the Roofline-model. For other works about memory bandwidth scheduling see Section 3.2.2.

Problem example The basic idea was experimentally investigated as part of the diploma thesis of Jochen Seidel [119]. Jochen Seidel did an external work at SAP which was concerned with the implementation and scheduling of queries in an in-memory database. The goal was to make queries faster and more efficient by a better utilization of the memory connection and by running different queries in parallel (the queries themselves are also parallelized). The main guidance for the diploma thesis was done by Jonathan Dees at SAP (database part) and the author of this work at the chair of Peter Sanders (NUMA-advice, memory modelling-advice and scheduling). The diploma thesis of Jochen Seidel contains two main parts: improving the NUMA-locality of the queries and combining different queries according to the basic idea described above. This paragraph is a description of the results of the diploma thesis of Jochen Seidel. Database queries looked like an easy to model type of memory bandwidth dependent tasks, as their main operation is to read through the database contents stored in memory.

In order to improve the NUMA-locality, Jochen Seidel developed a partitioning scheme for the distribution of the database contents over the different NUMA-sockets. Each query runs threads on each socket to work on the data stored on this socket by accessing the distribution scheme. For some queries a performance gain of up to 67% was observed for certain numbers of cores compared to a non-NUMA-aware, already existing solution. This NUMA-aware partitioning of the queries and the data also makes it sufficient to look at only one socket for the further modelling as all sockets run the same job combination. Especially if two queries run in parallel each query uses the same number of cores on all sockets.

Our goal for the query combination was to build a simple model which then should be used to combine queries in a way which leads to a significantly improved efficiency. As already noted, we planned to characterize the queries by two parameters: the computational effort and the needed memory bandwidth. Then query combinations similar to the combination described in the basic idea would lead to an increased efficiency. The experiments described here were all done at SAP on an Intel Xeon X7560 4-socket machine with 8 cores per socket (Hyperthreading was not used). In order to build a model which then could be used by a query scheduler, we had to model the memory bandwidth behavior of the machine and the running time behavior of the queries. The basic assumption of the model is that the running times of the queries can be reasonably well estimated by knowing the number of used cores per socket and the amount of competing memory bandwidth usage.

The machine model was built by running between 1 and 8 jobs (per socket) similar to the linear-test from Section 6.1 and computing the sum of the reached memory bandwidths similar to the speedup experiment from Section 6.1 (the speedup is then computed by dividing these bandwidths by the bandwidth reached by a single linear-test). The resulting speedup curves are very similar to the respective speedup curves of linear-test for 32-Core-Sandy-Bridge and 16-Core-Ivy-Bridge as shown in Figure 6.2. Especially the speedup grows fast for small numbers of cores until it reaches a limit and is a nearly constant function afterwards. Given a total request of x times the memory bandwidth of linear-test on a single core, we wanted to model the resulting memory bandwidth of each job (as we cannot use more than the bandwidth limit). We assumed that the available bandwidth is shared between different cores proportional to their bandwidth requests. Let $s(x)$ be a spline interpolation of the speedup function. We then used $f(x) = s(x)/x$ as the proportional factor which is used to reduce the bandwidth assigned to the competing requests. Hence the memory accesses of each job were slowed down by a factor of $1/f(x)$ in our model (note that $f(1) = 1$).

For the queries we used a very simple model by assuming that a query (when run sequentially) spends a fraction q of its running time for memory operations and the remaining part $(1-q)$ for computation. The parameter q is query-specific. The memory operations are assumed to be slowed down by the function f. Then we get the modelled execution time of a query with a running time t with one core per socket (on an otherwise empty system), when running on p cores per socket, with a fraction q of memory access time and a competing bandwidth requirement of k as:

$$\text{time}(p,q,k,t) = \frac{t}{p} \cdot \left(\frac{q}{f(p \cdot q + k)} + (1-q) \right)$$

For each query the parameter q was estimated by running all possible combinations of the query with different degrees of parallelism and different numbers of

Query 1	Query 2	ρ_{model} (%)	ρ (%)	$\rho - \rho_{\text{model}}$
1	1	0.00	2.11	2.11
5	1	0.80	3.23	2.44
5	5	0.00	9.98	9.98
6	1	0.00	3.70	3.70
6	5	0.76	5.30	4.54
6	6	0.00	4.20	4.20
7	1	1.17	-0.96	-2.13
7	5	0.04	0.36	0.32
7	6	1.13	0.30	-0.83
7	7	0.00	3.92	3.92
8	1	0.60	2.97	2.37
8	5	0.01	3.06	3.04
8	6	0.57	4.01	3.44
8	7	0.09	-4.55	-4.65
8	8	0.00	10.06	10.06
9	1	0.81	39.29	38.48
9	5	0.00	40.23	40.23
9	6	0.77	39.66	38.89
9	7	0.03	37.80	37.77
9	8	0.02	40.76	40.74
9	9	0.00	56.95	56.95

Table 6.3. Comparison of modelled and measured efficiency gains (in %). Table is an excerpt of a table in [119]. The first two columns are the IDs of the queries run in parallel.

linear-test threads on the remaining cores. Also this test only looked at the cores of one socket. For each combination the running time of the query was measured. q was then chosen such that the sum of squared errors between the modelled and the measured running times was minimized.

After the model building was done and all parameters were fixed, we could start with checking the predictive power of the model. In order to do this, different combinations consisting of two queries were run, each query getting half of the available cores per socket. By using unmeasured test runs of queries, it was made sure that all measured runs of queries run in parallel to their expected partner. Unfortunately many query combinations could not be measured due to some problems with the testing infrastructure. But the successfully completed experiments were already enough to show that the modelling error is far too large to make the model useful for scheduling decisions. Table 6.3 shows the measured and the modelled efficiency gains (and their difference) when running the two query types in parallel (each on half of the cores of each socket) instead of one after another (each on all cores). When running two identical queries in parallel, our model predicts an efficiency improvement of 0% as both queries are either memory- or compute-bound. In the real measurements (especially for query 9) we can see that the improvements can be quite large which indicates that the parallelization overhead is much more relevant than expected. On the other hand some combinations even lead to losses which indicates that there are other important parts missing in the model (maybe cache-related issues). These problems and several more are discussed in the thesis of Jochen Seidel.

In the context of this work the results of the modelling approach described above indicated that even simple-looking problems can be difficult to model. We took these problems as motivation to take a further step back and to explore the properties of memory hierarchies by simple synthetic jobs (like the ones in Section 6.1) instead of more complicated real problems. In our opinion a good understanding of the behavior of different synthetic jobs running in parallel is the basis for an understanding/useful modelling of real-world problems. An example of an insight from the measurements of Section 6.1 that might also contribute to the modelling problems described above comes from the slowdown experiment. Our model for the database query jobs implies some kind of symmetry, we measured the needed memory bandwidth of a query through the slowdown of the query by competing memory accesses. This needed memory bandwidth was then included in the computation of the slowdown of other queries running in parallel to the measured query. Especially we assumed that a query which is severely slowed down by competing memory accesses uses a high bandwidth itself, and thus it is also a large slowdown for other memory-dependent jobs. A look at the slowdown experiment shows that this symmetry does not hold for all kinds of memory access patterns. For example our chain-test is severely slowed down by linear-tests running in parallel, but it slows down other memory operations only slightly (see Figure 6.1).

6.2.2 Improvement of Cache Usage

As we have seen in Section 6.1, using cache instead of memory for the items to load and to store is beneficial for both latency and bandwidth. The main problem of cache memory is its limited size and that the cache cannot be managed directly on current machines.

Basic idea: The idea here is to run threads in parallel on cores which share a common cache in order to enable the common usage of items loaded into the cache. This is beneficial in two ways: shared items use cache space only once, and shared items have to be loaded only once from main memory.

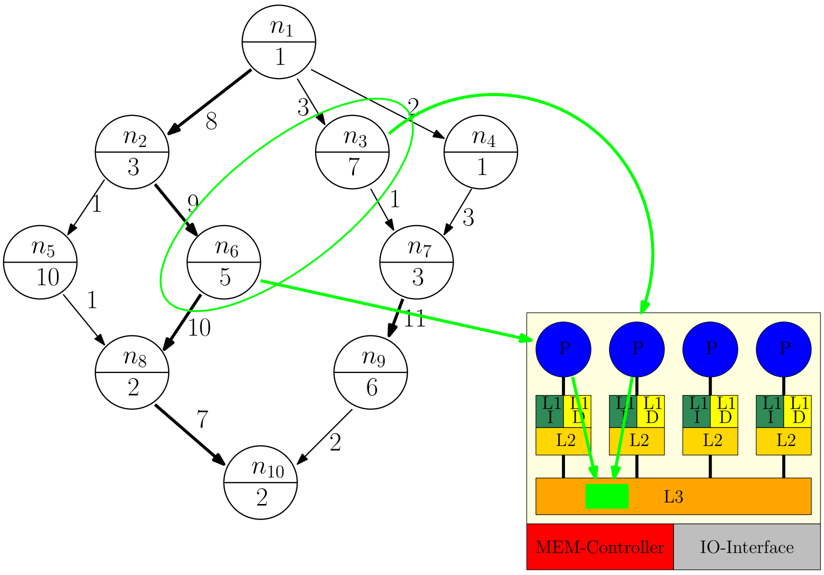


Figure 6.5. An example of cache co-usage in a DAG. If nodes n_3 and n_6 share some common input, scheduling them at the same time at the same processor with common L3-cache leads to a common usage of this input.

Cache sharing between tasks can thus be also a beneficial objective for the scheduling of tasks in task-DAGs. In order to enable cache co-usage, it is not only important to look at the inputs of tasks that are outputs of other tasks but also at common inputs that even might never be written by another task. Because of such common inputs it might be useful to schedule tasks together that are completely unrelated within the task-DAG. Figure 6.5 shows an example of this. Cache co-usage is more dedicated to application-internal schedulers as different applications usually do not share data. But cache sharing can also be influenced by system schedulers which place some threads of the same application on different cores which might share a common cache. The basic idea was first proposed by Blelloch and Gibbons [12] who together with others tested it on LU-decomposition (Chen et al. [31]) but found no improvement.

Problem example In order to test the basic idea, we need a task-DAG with a regular structure to keep the scheduling decisions simple. Thus, using a problem from numerical linear algebra, where very regular task-DAGs occur, is an obvious choice. The test case is the LU-decomposition of matrices, a very common and important problem for which already highly optimized solvers exist. A significant improvement of the performance of these solvers shows the usefulness of the basic idea. This goal was reached in a joint work [100] with Tobias Maier and Peter Sanders (for the individual contributions to the articles used in this work see Section 3.4.4). The research done for this article is the base for the remainder of this section. The measurements are new but based on software developed for our article. In the original article we were able to demonstrate a performance gain of 15% on large matrices over the highly optimized PLASMA-solver [21]. The experiments of Chen et al. [31] are part of a larger series of experiments in which they compare a Parallel Depth First scheduler with a scheduler based on Work Stealing. For most applications the Parallel Depth First scheduler leads to an improved constructive cache sharing and thus improved running times, but in case of the LU-decomposition their experiments show no improvement in running time. Our approach is more specialized to numerical linear algebra, and we developed two new methods: first, meta-tasks/meta-tiles to improve the constructive cache sharing between different cores and second, a (meta-tile-)column-wise distribution of the matrix among sockets to improve the NUMA-locality of memory accesses.

In order to explain the application of the basic idea, we first give a short introduction into the LU-decomposition. An LU-decomposition of a square nonsingular matrix A consists of two triangular matrices L (lower triangular matrix, all elements on the diagonal are ones and all elements above the diagonal are zero) and U (upper triangular matrix, all elements below the diagonal are zero) such that $A = L \cdot U$. As all entries of L on the diagonal and above and all entries of U below the diagonal are already defined, L and U can be stored within the same memory as A (the LU-decomposition can even be done in-place). The LU-decomposition is often used to solve systems of linear equations of the form $Ax = b$ by solving $Ly = b$ and $Ux = y$ which is easy for triangular matrices. For the description of the basic LU-decomposition we follow Trefethen and Bau [134, pages 147-162] where also a more detailed description can be found. The matrix U is computed from A by a sequence of Gaussian elimination steps which can be represented as lower triangular matrices. For example for a 3 × 3 matrix the first elimination step would be (we assume $a_{11} \neq 0$):

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \to L_1 A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & a'_{32} & a'_{33} \end{pmatrix} \text{ with } L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -a_{21}/a_{11} & 1 & 0 \\ -a_{31}/a_{11} & 0 & 1 \end{pmatrix}$$

Hence for a $n \times n$ matrix A the product $L_{n-1} \cdot \dots \cdot L_1 A$ is an upper triangular matrix. With $L = L_1^{-1} \cdot \dots \cdot L_{n-1}^{-1}$ we get the desired result. The computation is easy because the L_i are easy to invert (just negate the entries below the diagonal) and easy to multiply by combining their nonzero entries below the diagonal. It remains to be explained how we avoid to have an element $a_{ii} = 0$ at the beginning of a Gaussian elimination step. In general we want to have a_{ii} to have a large absolute value for numerical stability reasons. Hence we select the largest element of $a_{ii}, \ldots, a_{ni}$ as new a_{ii} by interchanging its row with the i-th row before performing the i-th elimination step (one of the mentioned elements must be different from zero as A is nonsingular). The row interchanges must also be done for the already computed part of L (which is $L_1^{-1} \cdot \cdots \cdot L_{i-1}^{-1}$). This method to increase the numerical stability is called partial pivoting and is the most commonly used method (Trefethen and Bau [134, pages 147-162] discuss this matter in more detail). The resulting decomposition with the partial pivoting is then $PA = LU$ with a permutation matrix P. The system of linear equations has then to be changed to $LUx = Pb$ but can be solved in the same manner.

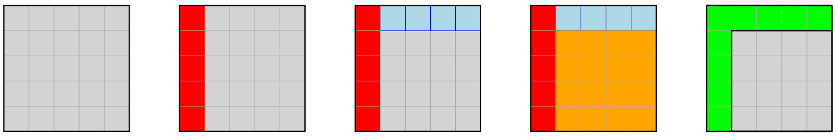


Figure 6.6. A schematic illustration of the LU-decomposition algorithm for tiled matrices.

Modern linear algebra algorithms usually work on tiled matrices which means that usually small quadratic sub-matrices are handled together. One advantage of tiled matrices is that the approach improves the operational intensity (see Section 6.2.1) as such tiles are loaded into cache and then each matrix entry within the tile is used for several arithmetical operations. The LU-decomposition of a tiled matrix consisting of $m \times m$ tiles of size $b \times b$ (with $mb = n$) is computed in steps similar to the Gaussian elimination steps but now performed on the tiles instead of the elements. Each step works on a matrix consisting of $k \times k$ tiles (beginning with $k = m$) and leaves an LU-decomposition of a $(k-1) \times (k-1)$ tiled matrix to the next step. Figure 6.6 illustrates such a step from a 5×5 to a 4×4 LU-decomposition. The first part of the step (called panel-task) is to compute the element-wise Gaussian elimination on the first tile column (the red part in Figure 6.6). Here the necessary row interchanges are determined, the uppermost, leftmost tile is LU-decomposed and the corresponding parts of L are computed. For each remaining tile of the first tile row (top-tiles, the blue parts in Figure 6.6) there is a task (called top-task) which applies the row interchanges (from the panel-task) to its tile column and applies the updates (given through the computed part of L) from the panel-task to its own tile. The remaining work to finish the step is to update the remaining part of the matrix (orange parts in Figure 6.6). For each tile of the remaining matrix there is an update-task. In our 3×3 example a'_{22} is computed by $a'_{22} = a_{22} - (a_{21}/a_{11}) \cdot a_{12}$ with $\ell_{21} = -a_{21}/a_{11}$ as the (second row, first column) entry of L and $u_{12} = a_{12}$ as the (first row, second column) entry of U we get $a'_{22} = a_{22} - l_{21} \cdot u_{12}$. In the general case we have for the k-th Gaussian elimination step that the k-th row of U has to be multiplied with

ℓ_{ik} (the ℓ_{ik} currently at this position, disregarding future row interchanges) and subtracted from the i-th row for all $k < i \leq n$. Hence for all $i, j \in \{k+1, \ldots, n\}$ we have $a'_{ij} = a_{ij} - \ell_{ik} \cdot u_{kj}$. For the tiled version we have analogously that an update-task has to compute $A'_{ij} = A_{ij} - L_{ik} \cdot U_{kj}$ with A_{ij}, A'_{ij} being the tile before and after the update and L_{ik} the tile in the i-th row computed by the panel-task and U_{kj} the tile computed by the respective top-task. As we do partial pivoting, we also perform row interchanges during the panel-task. The resulting row interchanges on tile columns left of the column the panel task works on are done by exchange-tasks. The exchange-tasks have little effort and are omitted for the remainder of this description.

We denote the panel-tasks by P, the tasks on the top-tiles by T and the update-tasks by U, we get the the task-DAG depicted in Figure 6.7 for our step on the 5×5 tiled matrix. The dashed lines denote that the task-DAG is embedded into a whole LU-decomposition. The depicted update-tasks are on the same level regarding the task-DAG but are arranged in a way that reflects the position of the tiles they update in the matrix. This ends the description of the usual approach of current algorithms for LU-decomposition which is especially followed by the PLASMA-solver and our approach which is based on PLASMA.

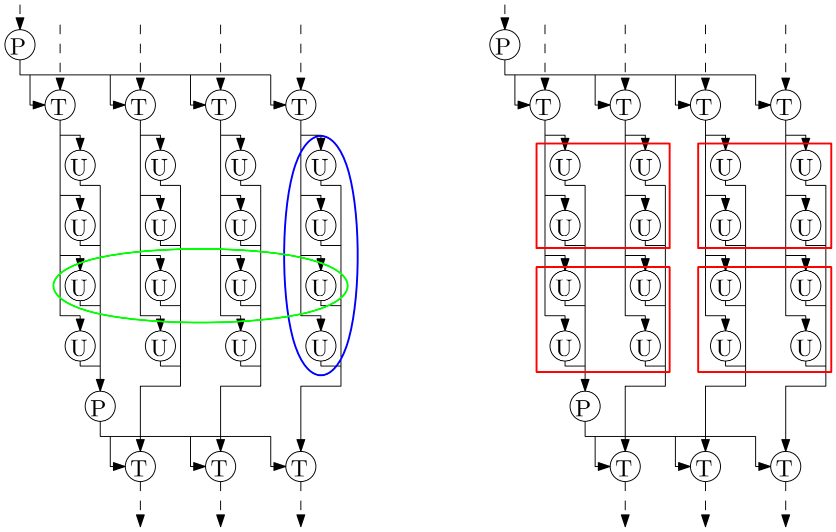


Figure 6.7. A schematic illustration of the LU-decomposition task-DAG for tiled matrices.

Throughout the whole LU-decomposition most of the work and running time is spent by the update-tasks. In terms of complexity the update-tasks are the only kind of jobs with a total computation effort cubic in the matrix size (the other kinds of jobs have a computation effort which is at most quadratic in the matrix size). Hence we focus on improving the execution time of these tasks. If we take a closer look at the inputs needed by an update-task, we find as relevant inputs: the tile to be updated (A_{ij}), the tile in the same row computed by the panel-task (L_{ik}) and the top-tile as computed by the top-task on the same column (U_{kj}). All three inputs have the same size (one tile) and are used to compute the updated tile by $A'_{ij} = A_{ij} - L_{ik} \cdot U_{kj}$. All update-tasks that work on tiles of the same row use the same tile L_{ik} computed by the the panel-task (indicated by the green ellipse in Figure 6.7). Also all update-tasks that work on tiles of the same column use the same top-tile U_{kj} (indicated by the blue ellipse in Figure 6.7). The idea of how to gain more constructive cache sharing within this problem is now to group the update-tasks into groups called meta-tasks (indicated by the red squares in Figure 6.7) that share common inputs. Accordingly, the tiles of the matrix which are updated by these tasks are grouped into meta-tiles. We use $M = h$ or Mh to denote the usage of meta-tiles consisting of $h \times h$ tiles (Figure 6.7 depicts a case of $M2$). These meta-tasks are then assigned as a group to a group of cores sharing a common cache (usually a NUMA-node). The split into the original tasks for the individual cores is then done locally (on the NUMA-node, this is illustrated in Figure 6.8). This leads to shared cache items which reduce the occupied cache space and the needed memory operations. Each L_{ik} and U_{kj} is used h times during a meta-task. As the input matrices do not always have dimensions which are multiples of h tiles and each step reduces the number of tile rows and columns by one, some meta-tiles are smaller. The boundaries between different meta tiles are kept the same for each step.

We also changed the LU-decomposition to make more memory-accesses NUMA-local. For each column of meta-tiles we store all meta-tiles on the same NUMA-node (the columns are assigned in a round-robin fashion). The assignment of update-tasks on meta-tiles is then done in a way such that a NUMA-node works on locally stored meta-tiles if possible. This has the additional benefit that the needed results from the top-tasks are already in cache when the tasks on the meta-tile are started. This is realized by using different queues for each NUMA-node and one additional queue for the panel-tasks. When a core on a NUMA-node becomes idle and there are no more tasks locally available, it fetches a new meta-task from the global queues. To do this it first looks into the queue with the panel tasks and then into the queue with the update-tasks with meta-tiles stored on that NUMA-node. Only if both queues do not contain a ready task, the NUMA-node might be assigned with an update-task on a meta-tile stored on a different NUMA-node. The queues are implemented as priority queues and the priorities are assigned to the meta-tasks in a way such that panel tasks and update-tasks of left parts of the matrix are prioritized against update-tasks of right matrix parts. The idea here is to work a bit faster on the critical path (the path with all panel tasks) in order to always have a large number of update-tasks ready.

Several other small improvements are also included in our approach to the LU-

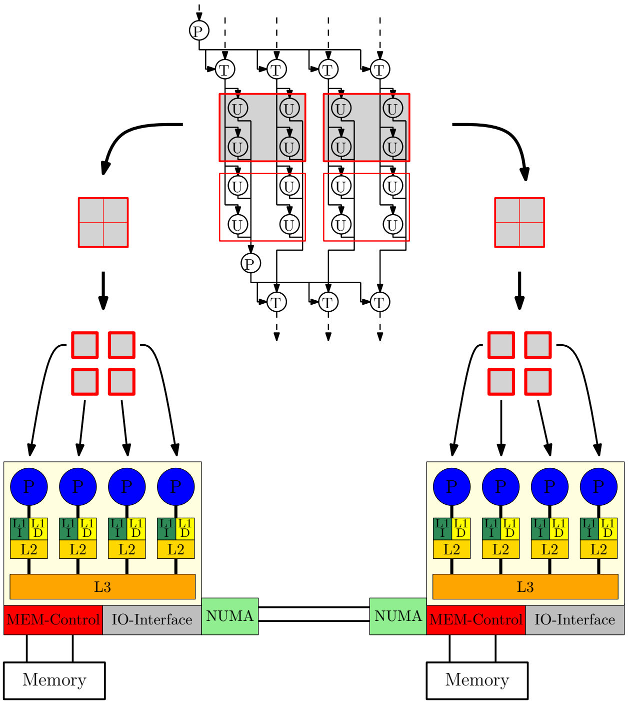


Figure 6.8. An illustration of the scheduling of meta-tasks.

decomposition. The central scheduling of meta-tasks and their local split reduces the contention on the global work distribution queue (or more precisely the mutex protecting it) which might reduce the waiting times. The task-DAG generation is done by only one core, but in parallel the other cores work on already generated tasks. We also restrict the panel computation to cores which use the same last-level cache in order to speed up the data exchange between the participating cores.

Experiments In order to show the advantage of our new scheduling approach, we compare our implementation (called META) with the PLASMA-routine for LU-decomposition (called PLASMA) which is also the basis on which META is developed. In our joint work [100] with Tobias Maier and Peter Sanders we ran most of our experiments on the 32-Core-Sandy-Bridge and some further experiments on the 16-Core-Ivy-Bridge (as introduced at the beginning of Section 6.1, but with Ubuntu 12.04. instead of Ubuntu 14.04. on the 32-Core-Sandy-Bridge). We broaden the perspective of the experiments by performing some new experiments on all four machines introduced at the beginning of Section 6.1.

In the experiments for our paper [100] we used PLASMA version 2.6.0 and linked with Intel's Math Kernel Library version 11.0. For the new experiments done for this work we use the most recent available software. The compilers and the MKL are part of the Intel Parallel Studio XE 2017, all machines run Ubuntu 16.04.2 LTS (Linux kernel 4.4.0) as operating system, and we compare with PLASMA [88] version 2.8.0 (released in November 2015). PLASMA 2.8.0 is the newest available version as PLASMA is currently undergoing substantial changes, as explained in the following quote from the PLASMA main page ([111]):

> PLASMA is in the process of porting form QUARK to OpenMP. At the same time, it is moving from its ICL SVN repository to this Bitbucket Mercurial repository. The content of this repository reflects the progress of the transition. Before the transition is complete, the last release of the old PLASMA is available here:
> https://bitbucket.org/icl/plasma/downloads/plasma-2.8.tar.gz

In our paper [100] we did all our measurements with Intel SpeedStep turned off (which also disabled Intel Turbo Boost). As there seems to be no possibility to switch off SpeedStep and Turbo Boost on the newer machines (24-Core-Haswell and 32-Core-Broadwell) and it seems to be more likely that these features are turned on during normal operation, these features are switched on for our new experiments. This results in an increased performance, and thus the absolute performance values are not comparable to the values measured in our paper. For META we use the code written by Tobias Maier for the experiments for our paper which is also publicly available (https://algo2.iti.kit.edu/2464.php).

The main value for comparisons is the performance in GFlop/s reached during the computation of the LU-decomposition. META as well as PLASMA use $\frac{2}{3}n^3 - \frac{1}{2}n^2 + \frac{5}{6}n$ as the number of floating-point operations needed to compute an LU-decomposition of a matrix of size $n \times n$. As META does the same computations as PLASMA only in a different order and on different cores, it is clear that the number of floating-point operations is the same. As noted above we denote META with a meta tile size of $h \times h$ as Mh and add col for the NUMA-optimized version and rand for the non-NUMA-optimized version. We always use the tiled version of PLASMA, and as the performance of PLASMA is very dependent on the tile size, we perform every experiment for different tile sizes for PLASMA. In our new experiments the tested tile sizes are $b = 256, 448, 480, 512$ which are denoted by PLASMA b. For META the tile size is always 256×256. META and PLASMA are always run with a number of threads that equals the number of cores of the used machine.

The matrix generator in META is implemented in a way such that the same matrices are generated as in the matrix generator of PLASMA. The generator does not include randomness which means that all matrices of the same size are identical. Hence META and PLASMA always decompose the same matrix in all runs for matrices of the same size. Warmups which means running some numerical computations before the measured test seem to be important to reduce the variance of the measurements. PLASMA does one unmeasured test run of an LU-decomposition of a matrix of the given size before running the measured LU-decompositions. META uses three LU-decompositions for a matrix size of 16384 × 16384 as warmup. Another notable implementation difference between PLASMA and META is the way the matrix is stored in memory. PLASMA allocates the matrix as one large array, META instead uses an array pointing to each tile and allocates each tile individually. This more complicated allocation of META is used for the NUMA-aware storage of the matrix tiles. Each tile is allocated during the matrix generation by the core that is responsible for generating the respective tile. As far as we know (see also the NUMA experiment in Section 6.1), memory allocations are usually done on the NUMA-node the respective core resides on. The generation tasks are bundled in meta-tasks (of the used meta-tile size in the respective experiment) and are assigned to the NUMA-nodes. Hence all tiles of one meta-tile are allocated and generated on the same NUMA-node. In case of the NUMA-optimized META (col) the generation is done in a way such that each column of meta-tiles is generated on the same NUMA-node (this column is then stored on this node). If META is run without the NUMA-optimization (rand), then all generation meta-tasks are inserted into the same queue which is then used to distribute these tasks among all NUMA-nodes. Hence the NUMA-nodes where a meta-tile is stored can vary depending on the execution of the generation.

We evaluate the performance of META and PLASMA for four different matrix sizes on each machine: 8192, 16384, 32768 and 65536 (all matrices are quadratic). For each configuration (matrix size, machine parameters) we run each program three times and each time five LU-decompositions are computed. Between two program runs we let the machine cool down for five minutes. The programs report the average running times of the five computations and a deviation. In the Table 6.4, Table 6.5 and Table 6.6 we report the average of the three measured averages in the upper left of each cell. In the lower left we report the difference between the largest and smallest average measured by the three program runs. PLASMA computes the deviation as the standard deviation normalized by the number of computations (here always 5). META computes a different kind of deviation, but the reported values are recomputed to match the deviation definition of PLASMA. The largest standard deviation of the three runs is reported in the lower right of each cell. Preliminary experiments showed that PLASMA sometimes profits from running with NUMA-interleaved memory (done by prefixing the call with `numactl --interleave=all`). Each PLASMA experiment is run twice, with and without NUMA-interleaved memory. If the average performance of the experiment with the NUMA-interleaved memory is higher, we report the results for that experiment, otherwise we report the results of the non-interleaved experiment. Whether the respective cell reports the performance of an interleaved computation or a non-interleaved computation is indicated by an i or an n in the upper right of the cell.

By looking at the results written down in Table 6.4, we see that META M3 col is at least 10% faster than the fastest PLASMA configuration for each matrix size on 32-Core-Sandy-Bridge and that the improvements on this machine are even larger for smaller matrix sizes. Contrary to this, the advantages of META against PLASMA with the best *b* on the 16-Core-Ivy-Bridge are very small except for the smallest matrix size. The advantages of META on 24-Core-Haswell and 32-Core-Broadwell (Table 6.5 and Table 6.6) are a bit larger than the advantage on 16-Core-Ivy-Bridge but much smaller than the advantage on 32-Core-Sandy-Bridge. Comparing the different configurations of META, we observe that META profits the most from the meta-tile and NUMA-optimizations on 32-Core-Sandy-Bridge. On all machines the meta-tile optimization brings relevant improvement compared to M1, but the additional NUMA optimization seems to be less relevant on the machines with two NUMA-nodes (they even look like a disadvantage on 16-Core-Ivy-Bridge). Also there is a clear advantage of META (M3 col/M5 col) against PLASMA 256 on all machines. By looking at the results of the NUMA experiment in Section 6.1, it is clear that the performance penalty for memory accesses on other NUMA-nodes instead of local accesses is the highest for 32-Core-Sandy-Bridge. Hence optimizations to avoid such accesses bring the most benefit there. As 32-Core-Sandy-Bridge is the only machine in this experiment with four NUMA-nodes (four sockets), it looks as if META has bigger advantages on four socket machines. On 24-Core-Haswell and 32-Core-Broadwell the advantage of PLASMA 480 compared to PLASMA 256 for the two larger matrix sizes is between 8% and 10%. The size of the 480 and 512 tiles are 1.76 MiB and 2 MiB. Hence if each core on one socket (common L3 cache) uses three different tiles for its current computation, no L3 cache of the machines in this experiment is large enough to keep all these tiles. Hence we assume that some clever cache-prefetching mechanism is responsible for the performance improvements of PLASMA with the larger tile sizes.

In Figure 6.9 we take a look at the running times of the single update-tasks. In META the running time of each update-task is logged. For each machine we take one log for each configuration of META (always for matrix size 16384) and plot the histogram for these running times. The x-axis denotes the running time of an update-task in milliseconds, and the curves show how many thousand update-tasks were run with a running time in the same bin (all histograms use 100 bins). We can clearly see the improvements of the M3/M5 configurations compared to the M1 configurations and that the relative differences between the peaks are smaller for the two-socket machines. We can also see that the NUMA-optimization makes little difference for the two-socket machines. On the other hand the sharing of cache contents seems to work as expected.

	8192		16384		32768		65536	
META M3col	417.5		494.4		512.1		519.8	
	0.8	4.6	0.8	1.9	0.6	1.9	1.1	0.4
META M3rand	400.5		462.8		480.3		490.4	
	2.0	2.1	2.4	1.1	1.2	1.2	0.8	1.8
META M1col	396.0		439.6		447.9		456.1	
	0.7	1.0	1.4	0.6	1.1	1.3	0.4	1.5
META M1rand	386.9		433.4		451.9		466.8	
	0.8	2.3	3.2	2.2	0.4	3.6	0.8	2.6
PLASMA 256	339.7	i	405.9	i	429.3	i	448.6	i
	4.1	6.4	1.3	1.2	0.2	1.4	0.3	0.4
PLASMA 448	238.4	n	417.2	n	452.0	n	465.0	n
	2.6	3.5	5.5	4.4	8.2	2.8	3.7	2.3
PLASMA 480	220.6	n	413.6	n	457.2	n	467.2	n
	1.8	3.5	2.4	3.9	3.7	1.6	2.3	3.0
PLASMA 512	196.7	n	403.9	i	458.8	n	471.9	n
	4.8	2.2	3.6	3.2	1.2	1.7	3.2	0.9

(a) 32-Core-Sandy-Bridge

	8192		16384		32768		65536	
META M3col	287.1		306.7		317.8		323.3	
	0.6	0.4	0.5	0.2	0.7	0.2	0.3	0.2
META M3rand	288.8		307.7		318.6		322.9	
	1.0	0.8	3.6	3.5	0.5	0.7	0.7	1.0
META M1col	279.4		297.0		306.9		313.3	
	0.8	0.3	0.2	0.3	0.2	0.2	0.0	0.2
META M1rand	283.7		298.9		308.1		313.7	
	0.4	0.8	1.5	0.4	0.5	1.1	0.2	0.5
PLASMA 256	271.4	i	290.4	i	300.7	i	306.2	n
	0.9	0.9	0.2	0.7	0.1	0.4	0.8	0.5
PLASMA 448	235.2	i	301.8	n	312.4	i	318.9	n
	2.3	8.0	0.2	1.1	0.6	0.3	0.4	0.5
PLASMA 480	230.6	i	304.6	i	316.5	n	322.0	n
	4.3	4.1	0.5	0.5	0.5	0.3	0.5	0.3
PLASMA 512	213.2	i	300.9	i	315.7	i	321.8	i
	2.7	3.1	0.7	1.1	0.5	0.3	0.7	0.3

(b) 16-Core-Ivy-Bridge

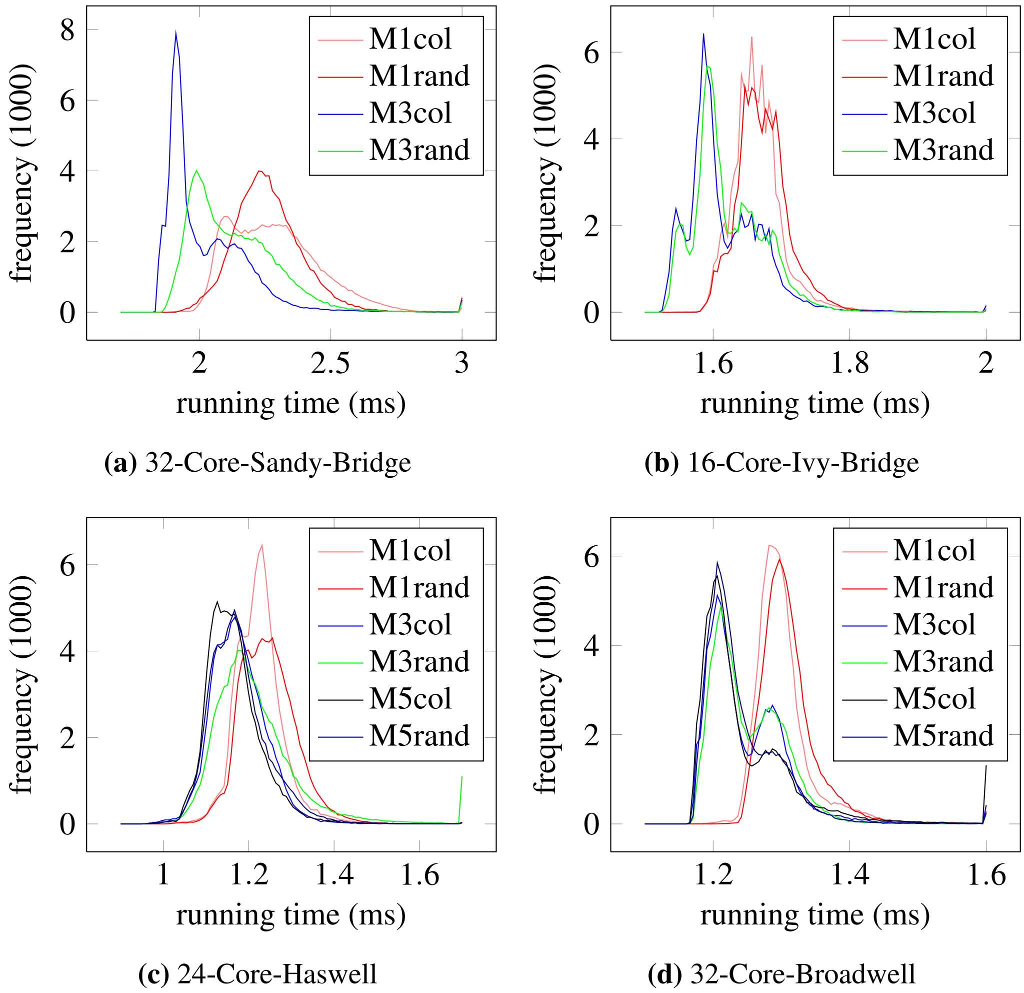
Table 6.4. Performance measurements of the LU-decomposition (32-Core-Sandy-Bridge and 16-Core-Ivy-Bridge).

	8192		16384		32768		65536	
META M3col	557.1		616.6		641.3		657.1	
	2.6	3.6	2.8	1.5	1.8	0.8	1.0	0.4
META M3rand	546.3		604.3		632.0		646.9	
	7.2	4.4	3.8	8.6	2.0	2.2	4.0	2.7
META M5col	551.6		619.4		646.4		661.3	
	5.8	4.8	1.6	1.4	1.6	0.7	1.3	0.4
META M5rand	541.3		610.2		634.1		652.0	
	4.4	6.9	2.6	8.4	2.9	6.9	2.1	3.3
META M1col	546.5		594.9		615.0		631.6	
	5.2	3.5	0.9	1.2	0.9	0.8	1.3	0.8
META M1rand	537.7		580.5		603.6		621.3	
	3.7	5.5	9.6	7.8	3.0	2.8	1.8	1.9
PLASMA 256	511.3	n	566.7	n	584.9	n	599.7	n
	4.6	9.3	2.8	3.6	0.8	1.4	1.0	0.4
PLASMA 448	400.4	n	587.1	n	620.3	n	637.9	n
	5.8	4.3	6.0	7.4	0.6	1.6	1.2	0.6
PLASMA 480	387.6	n	596.4	n	635.6	n	652.8	n
	12.3	6.7	3.2	3.3	0.4	1.3	1.2	0.5
PLASMA 512	338.4	n	570.2	n	615.4	n	631.0	n
	3.1	5.5	2.2	3.9	0.5	0.9	1.3	0.7

Table 6.5. Performance measurements of the LU-decomposition (24-Core-Haswell).

	8192		16384		32768		65536	
META M3col	658.0		758.4		800.3		826.0	
	3.6	4.2	1.2	1.9	1.7	0.5	1.6	0.3
META M3rand	648.6		759.9		796.8		823.0	
	2.4	4.6	6.6	9.2	2.4	9.4	5.6	4.7
META M5col	643.8		761.9		809.9		834.8	
	3.3	5.1	3.0	5.0	1.0	1.9	0.6	0.3
META M5rand	642.0		767.6		802.2		827.8	
	5.7	4.5	0.3	1.2	18.1	9.1	10.6	9.8
META M1col	643.6		727.1		762.1		784.7	
	3.6	3.9	1.8	2.4	2.0	1.6	4.7	1.5
META M1rand	637.0		727.6		759.6		785.8	
	4.7	2.6	3.8	2.5	3.2	1.8	2.7	4.6
PLASMA 256	577.4	i	700.9	i	732.4	n	751.4	n
	7.1	8.3	1.5	2.0	2.0	2.5	1.9	2.0
PLASMA 448	400.5	i	717.7	i	789.8	n	813.0	n
	14.3	13.1	1.9	3.5	0.9	2.3	3.1	1.1
PLASMA 480	376.1	n	714.4	i	801.0	n	825.3	n
	2.4	8.3	2.7	5.2	3.4	4.5	2.3	0.8
PLASMA 512	313.5	i	665.8	i	763.9	i	789.3	i
	1.6	3.9	11.3	7.8	1.2	1.2	2.0	0.7

Table 6.6. Performance measurements of the LU-decomposition (32-Core-Broadwell).



(a) 32-Core-Sandy-Bridge

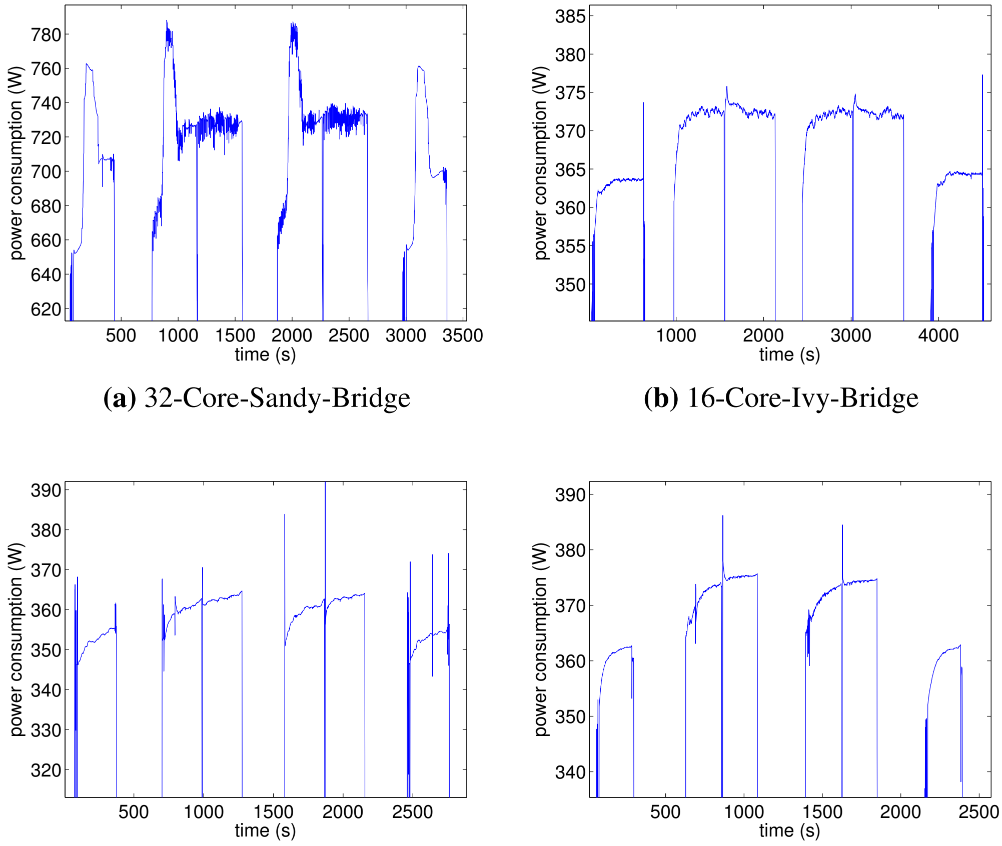
(b) 16-Core-Ivy-Bridge

(c) 24-Core-Haswell

(d) 32-Core-Broadwell

Figure 6.9. Histograms of the running times of the update-tasks for one run of matrix size 16384.

We also take a quick look at the power consumption in Figure 6.10. We use a a "watts up? PRO" from ThinkTank Energy Products Inc. to measure the power consumption of the whole machine while it computes an LU-decomposition. All tests are done with a matrix size of 65536, and both programs only perform one measured run per invocation. Between two program runs we let the machine cool down for 5 minutes. In order to show that the measurements are not disturbed by heat effects, we run one run of META followed by two runs of PLASMA and finish with one run of META. On 32-Core-Sandy-Bridge and 16-Core-Ivy-Bridge we run META in the configuration M3 col and on 24-Core-Haswell and 32-Core-Broadwell in the configuration M5 col. For PLASMA we always select the fastest tile size (512 on 32-Core-Sandy-Bridge and 480 on the other machines) and run



(a) 32-Core-Sandy-Bridge

(b) 16-Core-Ivy-Bridge

(c) 24-Core-Haswell

(d) 32-Core-Broadwell

Figure 6.10. Power consumption of the LU-decomposition for one run of matrix size 65536. The sequence on all machines consists of one run of META, two runs of PLASMA and one last run of META. The much wider peak of PLASMA is due to the different warmup.

without interleaving. In each of the four images of Figure 6.10 we can clearly recognize the four peaks separated by the five minute breaks (the PLASMA peaks are wider because of the different warmup). We also have to zoom into the measurements in order to recognize the differences in the power consumption. For 32-Core-Sandy-Bridge we ignore the peaks at the beginning of the computations which are probably artifacts of the fan regulation. By approximation of the images we estimate that the power consumption of META is 2-3% less than the power consumption of PLASMA although META has the higher performance (and thus computes more floating-point instructions per second).

Altogether the experiments show clear advantages of meta-tiles and meta-tasks on all machines compared to the scheduling of single tiles. The additional comparisons with PLASMA show that these improvements have the potential to improve even already well-optimized numerical linear algebra libraries. The NUMA-optimizations are most beneficial on the four-socket machine (32-Core-Sandy-Bridge) on which the difference between NUMA-local and non-NUMA-local memory accesses is larger than on the other machines. These results indicate that including the usage of the memory-hierarchy into scheduling approaches is beneficial (at least in some cases).

Cache-optimized LU-decomposition in the context of this work META contains a very simple example of the mission-type tactics (Auftragstaktik) as described in Section 4.2. We have two layers of task distribution (meta-tasks among the sockets and tasks among the cores), and the upper layer (central priority queues) does not need to know which task is executed on which core as long as all tasks of a meta-task are done. Hence a meta-task can be seen as an order to a NUMA-node which leaves some freedom about the decision which core on that node runs which contained task.

Even though META is developed as part of finding an approach for a memory model for scheduling, META does not use such a model. In case of META we do not trade off cache and memory access improvements of scheduling decisions with possible negative effects. Tiles are aggregated to meta-tiles without regarding negative effects. For example it is possible that update-tasks assigned to one socket as part of a meta-task are waiting to be executed while there are idle cores on other sockets (in case of meta-tile sizes larger than 1). A possible compromise between the cache optimization and the efficient usage of cores could be that in case of a small number of remaining meta-tasks these are split up and the single tasks are distributed directly among the cores (without regarding the sockets any more). Here one would need a model to decide (at runtime) when this behavior change should be done.

Also the results of our study of the LU-decomposition are not directly transferable to the scheduling of independent applications as these can typically not share cache contents because of running in different address spaces. On the other hand it shows that different threads of one application that share cache contents should probably run on the same socket at the same time.

7

Power and Energy

Energy consumption is a major cost factor of computing and is especially important in computing centers (see Section 2.1). For mobile devices the endurance of the battery is a very important decision factor for purchase decisions between different mobile phones, laptops and other devices. Also this endurance is not only influenced by the battery size (which can often not easily be increased due to size and weight restrictions) but also by the power consumption of the device. A high power consumption is not only problematic on the supply side but also on the output side. If energy is used for computing, the inevitable output is heat. This heat has to be dissipated as too high temperatures will destroy the computing device. Appropriate cooling devices are costly, require space and weight and often produce noise. All this can be reduced if less heat is produced. The heat problem and power consumption is also an important problem that fueled parallelization efforts in order to overcome this problem (see Section 2.2.1). A typical feature of current processors to deal with high temperature is to reduce clock speed (for example Intel Turbo Boost), hence energy consumption is not only important for the cost of energy and heat dissipation but also influences the working speed. Thus energy consumption is an important topic for computing devices and the scheduling of these devices.

The energy usage and heat production of a computing system is not only influenced by components directly related to the computation like processors, memory, hard disks and others but also by cooling fans or air conditioning systems in data centers. One might think that fans are of little importance for the system as a whole, but during our experiments we made a different observation. On one of our used machines, 32-Core-Sandy-Bridge (see Section 6.1 for details), which uses $\sim$ 180W in idle mode with fan speed regulated by its temperature and up to 820W during experiments on all cores, the power usage in idle mode increases to $\sim$ 380W if the fans are set to full speed. Even if such things are important for the energy usage we restrict ourselves in this work to things that can be influenced by scheduling decisions.

There are many different properties that can be influenced by scheduling decisions and can relevantly influence the power consumption of a computing system:

- The clock frequency/speed of computing cores.
- The efficient usage of the memory system by efficiently using caches (reduction of energetically expensive memory accesses).
- The usage of special devices capable of reducing energy consumption like special function units that can compute a result energetically cheaper than others.
- Sleep states of the computing device in times without work.
- Quality parameters of results which influence the computing effort.

In this work we focus on two of these properties: the clock speed (see Section 7.1) and the usage of the memory system (see Section 7.2). In case of the clock frequency we especially focus on energy-optimal clock speed settings for malleable jobs which are in the main focus of this work. The efficient memory usage of parallel systems as described in Chapter 6 is another topic of this work. In this chapter we focus on the energy usage effects of memory operations. Besides the obvious realization that each work has to make a cut between things to include and not to include the other properties mentioned above are less connected to this work. The sleep states (or switching the machine on and off) is not especially interesting in connection with parallel jobs. The quality parameters and the usage of special computing devices have not developed widely accepted models for parallel jobs yet and are difficult to evaluate experimentally. This might change in the future and then the energy-efficient scheduling for such problems will become a more important topic.

7.1 Clock Speed

As inevitable basis for scheduling of clock speeds we need a model of the energy consumption and the resulting work progress for different clock speeds. As there are different models in use for frequency schedules, we introduce the model from our article about frequency scheduling [117] (joint work with Peter Sanders). We start with modelling the energy consumption of a single core. If the power consumption P stays the same for a time interval of length T, the energy consumption is just $E = P \cdot T$. In order to estimate the power consumption, we take a look at the basic element of modern processors: the field-effect transistor. The gate of such a transistor behaves similar to a capacitor, the electric charge is proportional to the (supply) voltage. The energy used to load a capacitor is proportional to the product of charge and voltage. The number of times such load operations take place per second is proportional to the clock frequency. The speed to load or unload the gate of a transistor is proportional to the voltage (or maybe the voltage above a threshold). Hence the voltage (voltage above threshold) is usually proportional to the clock frequency. Altogether we can summarize that the power consumption of a transistor is proportional to the cubic operating frequency. Hennessy and Patterson ([64], example on page 23) get the same result by using a similar argumentation. Thus one can assume $E = f^3 \cdot T$ for a single core with operating frequency f and a fitting energy unit size that removes the need for a further constant factor. In order to broaden the applicability of our results, we assume $E = f^\alpha \cdot T$ with $\alpha > 2$ for the energy usage of a single core running for a time T with frequency f. If p cores run in parallel, we just add a factor of p for the energy usage:

$$E = p \cdot f^\alpha \cdot T$$

Now we look a the work done while running with a clock speed of f. In the sequential case we just assume that the work done is just proportional to the running time and the clock speed. If we use a fitting unit for the work w, we get: $w = f \cdot T$. When running in parallel, jobs often have no optimal linear speedup due to overheads. In order to make this part of the model, we use a (job-specific) speedup function $s_j(p)$. We thus use

$$w = s_j(p) \cdot f \cdot T$$

for the work done in a job j when running for a time T on p cores with a frequency f. We further assume that the frequencies can be selected from $\mathbb{R}_{>0}$ and that the frequencies are the same for all cores working on the same job but can be individually adjusted instantly for each core. Also the energy consumption of unused cores and the energy consumption independent of the clock speed are ignored in our model. Enhancements of our model to be more realistic and the resulting adaptions of the scheduling algorithms are discussed in Section 7.1.2.

An important realization for energy scheduling (and especially regarding this model) is that we need to bound the running time or include it into the objective function. If our objective is just energy minimization, then running all jobs sequentially with the minimal possible operating frequency is the optimal solution. This cuts down the parallelization overheads, and with higher clock speeds the energy consumption always grows faster than the working speed (as long as the basic energy consumption of the machine is not regarded). In most real cases there is probably a tradeoff between running time and the energy consumption which is influenced by many factors and is also situation-dependent. In order to get general, abstract problems, one has to bound either energy usage or running time. Most works mentioned in Section 3.1.5 (where we take a look at other work in energy scheduling) assume deadlines. Some of these works use more general or more complicated (and probably more realistic) energy models.

7.1.1 Clock Speed for Malleable Jobs

Here we take a deeper look into the problem *Minimize the Used Energy* already described in Section 5.2.5 (Corollary 4) and in a joint work with Peter Sanders [117]. The paper [117] does not contain all necessary proofs, especially these proofs will be given here as well as the other relevant parts to make this work self-contained. This section is hence mainly a copied part of the article [117] with added proofs and some smaller changes. We recapitulate the relevant properties of the problem for this section: The energy consumption and work dependence fit the model described at the beginning of Section 7.1, we only have to take a closer look at the speedup functions. The speedup functions s_j (with $s_j(0) = 0$ and $s_j(1) = 1$) of the jobs are concave and further fulfil this property: A function h with $h(0) = 0$ and further defined as $h : p \mapsto \sqrt[\alpha-1]{s^\alpha(p)/p}$ with $\alpha > 2$ must be strictly monotonically increasing for all $p < \bar{p}$ (for a $\bar{p} \in \mathbb{N}$) and monotonically decreasing for all $p > \bar{p}$ and additionally concave on $(0, \bar{p}]$. As already described in Section 5.2.5 for the problem *Minimize the Used Energy*, a lot of commonly assumed speedup functions fulfil these properties. All jobs have a common release time and deadline and must be scheduled on a machine with parallel identical cores such that the used energy is minimized. Each job has to do an amount w_j of work. The goal of this section is to provide a function E_j for the energy usage of each job given the average (possibly non-integer) amount of used cores between the release time and deadline. It is also shown that it is optimal for a job with a given average amount of used cores only to use the two adjacent (integer) core amounts of the average amount. Within this section we focus on one job with a given average core usage during a time interval $[0, T]$ which has release time 0 and deadline T. Hence we can omit the job-related indices for formulas, constants and variables.

A malleable job can run on different numbers of cores during its execution time. The main problem in order to reach the goal of this section is that depending on the speedup function the most energy-efficient schedule for the job can require different clock speeds when running on different amounts of cores.

Lemma 7.1.1. *In an optimal solution for any number of cores p used during the computation of the job the processing frequency is always the same if p cores are used.*

Proof. Let us assume there are two intervals I_1, I_2 and the job runs with frequency f_1 in I_1 and a different frequency f_2 in I_2 (w.l.o.g. $f_1 < f_2$) and in both intervals the job uses the same amount of p cores. W.o.l.g. we can assume that both intervals have the same length/duration t. The sum of work done and the sum of energy used in these two intervals is then:

$$\begin{array}{lcccc} w_{12} & = & s(p)\cdot f_1\cdot t & + & s(p)\cdot f_2\cdot t \\ E_{12} & = & p\cdot f_1^{\alpha}\cdot t & + & p\cdot f_2^{\alpha}\cdot t \end{array}$$

In order to prove the lemma, we have to show that we can do the same amount of work with the same amount of cores and less energy in these intervals, and hence that a solution where one job combines different frequencies with the same amount of cores cannot be optimal.

If we set $f_{new} = 1/2\cdot(f_1+f_2)$ for both intervals then the work done w_{new} in both intervals with the new frequency is the same as w_{12}. For the energy we get $E_{new} = p\cdot f_{new}^{\alpha}\cdot 2t$. The function $E(f) = 2t\cdot p\cdot f^{\alpha}$ is strictly convex regarding the variable f. Hence

$$E_{new} = E(\frac{1}{2}f_1+\frac{1}{2}f_2) \;<\; \frac{1}{2}E(f_1)+\frac{1}{2}E(f_2) = E_{12}$$

As we could reduce the energy usage, the solution with different frequencies cannot be optimal. □

With this lemma we can now compute the energy usage of a job if it uses p cores during the whole interval $[0,T]$: the work done is $w = s(p)\cdot f\cdot T$, thus we can compute the needed clock speed as $f = \frac{w}{T}\cdot\frac{1}{s(p)}$. Hence the used energy is:

$$E(p) = p\cdot f^{\alpha}\cdot T = \frac{w^{\alpha}}{T^{\alpha-1}}\cdot\frac{p}{s^{\alpha}(p)} = \frac{w^{\alpha}}{T^{\alpha-1}}\cdot h^{-\alpha+1}(p)$$

Hence the smallest energy usage is reached for $\bar{p}$ as h reaches its maximum there. That the smallest energy usage is reached for $\bar{p}$ cores of course also holds for any subinterval.

As the goal of this section is to find an optimal schedule for a single job with a possible non-integer average core usage, we need to handle non-integer core numbers. For a non-integer core number we introduce the notation $p+\tau$ for the rest of the section where p is the integer part and $\tau\in[0,1)$. The main goal of this Section is to compute the minimal energy consumption (and according optimal schedule) of a job with work w to be done in time T when the job can use an average number of $p+\tau$ cores. A job runs on an average core number $p+\tau = T^{-1}\sum_{p=0}^{m} t_p p$ during time $T = \sum_{p=0}^{m} t_p$ if it runs on p cores for time t_p (m is the total number of cores of the machine like in Chapter 5 and thus the maximal degree of parallelism). All core numbers in this section are not bigger than $\bar{p}$, otherwise you could shrink all larger core numbers to $\bar{p}$ without using more energy (see above). Hence we use as general assumption that the number of cores used in any time interval is bounded above by $\bar{p}$. Most lemmata in this section state that there is an optimal solution in which some property holds. The proofs are mostly done by assuming this property does not hold for a solution and then showing that an at least as good solution with this property exists (the energy usage is not higher).

Lemma 7.1.2. *If the average number of cores is τ, then it is optimal to run on* 1 *core for time τT and not to run for time* $(1-\tau)T$.

Proof. We consider the case of a job which runs on p cores for a time interval I_1 of length t_1 and runs on 0 cores (or does not run) for a time interval I_2 of length t_2. Let w be the total work done in I_1 (and I_2) and $E_{old} = \frac{w^\alpha}{t_1^{\alpha-1}} \cdot \frac{p}{s^\alpha(p)}$ be the energy used. Set $\nu := \frac{t_1 \cdot p}{t_1+t_2}$. As the average core usage is less than 1 we know that $\nu < 1$. We will now use 1 core for time $\nu(t_1+t_2)$ and 0 cores for time $(1-\nu)(t_1+t_2)$ and show that this does the same work with no more energy. The energy used to do work w with 0 and 1 core is

$$E_{new} = \frac{w^\alpha}{(\nu \cdot (t_1+t_2))^{\alpha-1}} \cdot \frac{1}{s^\alpha(1)} = \frac{w^\alpha}{t_1^{\alpha-1}} \cdot \frac{1}{p^{\alpha-1}} = \frac{s^\alpha(p)}{p^\alpha} \cdot E_{old} \leq E_{old}$$

If $s(p)$ is strictly concave we even have $E_{new} < E_{old}$ as $s(p) < p$ in this case.

The repeated use of this argument for all intervals where the job runs on more than one core shows that if the average number of cores used during time T is $\tau \leq 1$, then it is optimal to use one core during time $\tau \cdot T$ and 0 cores during time $(1-\tau) \cdot T$. □

Lemma 7.1.3. *If the average number of cores is $p+\tau \geq 1$, then it is optimal to run on at least* 1 *core throughout* $[0,T]$.

Proof. Let us assume there is an interval I_1 where the job does not run. As the average number of used cores during $[0,T]$ is at least 1 there must be an interval I_2 where the job runs on at least 2 cores. Let the number of cores used during I_2 be p and w.l.o.g. let I_1 and I_2 have the same length t. If the amount of work done in I_2 is w, we run on frequency $f_{old} = w/(s(p)\cdot t)$ during I_2 and the energy usage is:

$$E_{old} = \frac{w^\alpha}{t^{\alpha-1}} \cdot h^{-\alpha+1}(p)$$

We want to show now that if we run on one core in I_1 and on $p-1$ cores in I_2, we can do the same amount of work as before with no more energy usage. The interesting question is how to split the work between the two intervals. The amount of work done in interval I_1 will be βw (with $\beta \in (0,1)$), and thus the amount of work done in I_2 will be $(1-\beta)w$. If we set $1-\beta = s(p-1)/s(p)$, we get for the energy usage:

$$\begin{aligned}
E_{new} &= \frac{\beta^\alpha w^\alpha}{t^{\alpha-1}} \cdot h^{-\alpha+1}(1) + \frac{(1-\beta)^\alpha w^\alpha}{t^{\alpha-1}} \cdot h^{-\alpha+1}(p-1) \\
&= \frac{w^\alpha}{t^{\alpha-1}} \left(\left(1 - \frac{s(p-1)}{s(p)}\right)^\alpha \cdot \frac{1}{s^\alpha(1)} + \frac{p-1}{s^\alpha(p)} \right) \\
&= \frac{w^\alpha}{t^{\alpha-1}} \cdot s^{-\alpha}(p)\,(p-1+(s(p)-s(p-1))^\alpha) \\
&= E_{old} + \frac{w^\alpha}{t^{\alpha-1}} \cdot s^{-\alpha}(p)(-1+(s(p)-s(p-1))^\alpha)
\end{aligned}$$

As the speedup can grow at most by one if you add one core $-1+(s(p)-s(p-1))^\alpha \leq 0$ and thus the lemma is proven. For strictly concave $s(p)$ even $-1+(s(p)-s(p-1))^\alpha < 0$ holds which makes not running for some time interval a suboptimal solution. □

Lemma 7.1.4. *Assume a job that runs on p_1 cores for a time interval I_1 of length t_1 and runs on p_2 cores for a time interval I_2 of length t_2 and has to do the amount w of work while running on these two intervals.*

If we distribute the amount of work between these two intervals such that the energy consumption is minimized, the energy usage during these two intervals is:

$$E = w^\alpha \cdot (t_1 h(p_1) + t_2 h(p_2))^{-\alpha+1}$$

Proof. If we do βw work during I_1 and $(1-\beta)w$ work during I_2, we get the following energy as a function of $\beta \in (0,1)$:

$$\begin{aligned}
E(\beta) = \quad & \frac{\beta^\alpha w^\alpha}{t_1^{\alpha-1}} \cdot \frac{p_1}{s^\alpha(p_1)} + \quad & \frac{(1-\beta)^\alpha w^\alpha}{t_2^{\alpha-1}} \cdot \frac{p_2}{s^\alpha(p_2)} \\
= \quad & \frac{\beta^\alpha w^\alpha}{t_1^{\alpha-1}} \cdot h^{-\alpha+1}(p_1) + \quad & \frac{(1-\beta)^\alpha w^\alpha}{t_2^{\alpha-1}} \cdot h^{-\alpha+1}(p_2)
\end{aligned}$$

We now have to find the minimum of $E(\beta)$. We do this by computing the β with $E'(\beta) = 0$. This β is a minimum because $E''(\beta) > 0$ for all $\beta \in (0,1)$. For $A = t_1^{-\alpha+1} \cdot h^{-\alpha+1}(p_1)$ and $B = t_2^{-\alpha+1} \cdot h^{-\alpha+1}(p_2)$ the minimizing β is

$$\beta = \frac{\sqrt[\alpha-1]{B}}{\sqrt[\alpha-1]{A} + \sqrt[\alpha-1]{B}}$$

Thus with the optimal β for E we get the value:

$$
\begin{aligned}
E =& w^{\alpha} \cdot ((\beta^{\alpha} A + (1-\beta)^{\alpha} B) \\
=& w^{\alpha} \cdot (\sqrt[\alpha-1]{A} + \sqrt[\alpha-1]{B})^{-\alpha} \cdot (B^{\frac{\alpha}{\alpha-1}} A + A^{\frac{\alpha}{\alpha-1}} B) \\
=& w^{\alpha} \cdot (\sqrt[\alpha-1]{A} + \sqrt[\alpha-1]{B})^{-\alpha} \cdot AB \cdot (B^{\frac{1}{\alpha-1}} + A^{\frac{1}{\alpha-1}}) \\
=& w^{\alpha} \cdot (\sqrt[\alpha-1]{A} + \sqrt[\alpha-1]{B})^{-\alpha+1} \cdot AB \\
=& w^{\alpha} \cdot (A^{\frac{-1}{\alpha-1}} + B^{\frac{-1}{\alpha-1}})^{-\alpha+1} \\
=& w^{\alpha} \cdot (t_1 h(p_1) + t_2 h(p_2))^{-\alpha+1}
\end{aligned}
$$

□

Lemma 7.1.4 is a central result of this section, because it enables us to define a minimal energy usage even when the work must be split between two intervals where different numbers of cores are used.

Lemma 7.1.5. *If the average number of cores is $p + \tau \geq 1$, then it is optimal to run on $p+1$ cores for time τT and to run on p cores for time $(1-\tau)T$. General assumption: $p + \tau \leq \bar{p}$*

Proof. We consider the case of a job which runs on p_1 cores for a time interval I_1 of length t_1 and runs on p_2 cores for a time interval I_2 of length t_2 (because of Lemma 7.1.3 $p_1, p_2 > 0$) w.l.o.g. $0 < p_1 < p_2 \leq \bar{p}$. Let w be the total work done in both intervals. From Lemma 7.1.4 we know how the work w is optimally distributed between I_1 and I_2 and what the energy usage is if the work is optimally distributed: $E = w^{\alpha} \cdot (t_1 h(p_1) + t_2 h(p_2))^{-\alpha+1}$

We set $p := \lfloor \frac{t_1 p_1 + t_2 p_2}{t_1 + t_2} \rfloor$ and $\tau := \frac{t_1 p_1 + t_2 p_2}{t_1 + t_2} - p$ then $p + \tau$ is the average number of cores used during I_1 and I_2 and $p_1 \leq p < p+1 \leq p_2$. We now want to show that using p cores during time $(1-\tau)(t_1+t_2)$ and $p+1$ cores during time $\tau \cdot (t_1 + t_2)$ is an optimal solution to do the work w during $I_1 \cup I_2$. In order to do this, it is sufficient to show that $t_1 h(p_1) + t_2 h(p_2) \leq \tau \cdot (t_1 + t_2) h(p+1) + (1-\tau)(t_1 + t_2) h(p)$.

If we set $r := \tau \frac{t_1+t_2}{t_1} \cdot \frac{p_2-(p+1)}{p_2-p_1}$, then $1 - r = (1-\tau)\frac{t_1+t_2}{t_1} \cdot \frac{p_2-p}{p_2-p_1}$, and with $s := \tau \frac{t_1+t_2}{t_2} \cdot \frac{p+1-p_1}{p_2-p_1}$ we get $1 - s = (1-\tau)\frac{t_1+t_2}{t_2} \cdot \frac{p-p_1}{p_2-p_1}$. With this we have:

$$
\begin{aligned}
t_1 h(p_1) + t_2 h(p_2) &= r t_1 h(p_1) + s t_2 h(p_2) + (1-r) t_1 h(p_1) + (1-s) t_2 h(p_2) \\
&= \tau \cdot (t_1 + t_2) \cdot \left(\frac{p_2 - (p+1)}{p_2 - p_1} h(p_1) + \frac{p+1-p_1}{p_2 - p_1} h(p_2) \right) \\
&\quad + (1-\tau)(t_1 + t_2) \cdot \left(\frac{p_2 - p}{p_2 - p_1} h(p_1) + \frac{p - p_1}{p_2 - p_1} h(p_2) \right) \\
&\leq \tau \cdot (t_1 + t_2) h(p+1) + (1-\tau)(t_1 + t_2) h(p)
\end{aligned}
$$

because h is concave for $p_1, p, (p+1), p_2 \in (0, \bar{p}]$.

The repeated use of this argument for all intervals with different numbers of cores shows that if the average number of cores used during time T is $p+\tau$ with $p \geq 1$, then it is optimal to use $p+1$ cores during time $\tau \cdot T$ and p cores during time $(1-\tau) \cdot T$. □

With Lemma 7.1.2 and Lemma 7.1.5 we can define the energy usage for an average number of cores $p+\tau$ as the optimal energy usage of this case:

Definition 7.1.6. *A job which does work w during time T on an average number of cores $p+\tau$ with $p \in \mathbb{N}_0,\ p+\tau \leq \bar{p}$ uses energy*

$$\overline{E}(p+\tau) := E(p,\tau) := \frac{w^\alpha}{T^{\alpha-1}} \cdot (\tau \cdot h(p+1) + (1-\tau)h(p))^{-\alpha+1}$$

It is immediately clear that $\overline{E}$ is a continuous function on $(0,m]$ and has the same values as the function E on integer p (see above).

Lemma 7.1.7. *The function $\overline{E}(p+\tau)$ as defined in Definition 7.1.6 is strictly convex and strictly decreasing on $(0,\bar{p}]$ and has its minimum at $\bar{p}$.*

Proof. Let $E_\tau(p,\tau) := \frac{\delta}{\delta\tau}E(p,\tau)$, then we have:

$$\begin{aligned} E_\tau(p,\tau) &= -\frac{w^\alpha}{T^{\alpha-1}} \cdot (\alpha-1)(h(p+1)-h(p)) \\ &\quad \cdot (\tau \cdot h(p+1) + (1-\tau)h(p))^{-\alpha} \end{aligned}$$

As $E_\tau(p,\tau) < 0$ as long as $p+1 \leq \bar{p} \Leftrightarrow h(p+1)-h(p) > 0$, it is clear that $\overline{E}(p+\tau)$ is strictly decreasing on $(0,\bar{p}]$. Also for $1 \leq p \leq \bar{p}$ it is clear that $E_\tau(p,0)$ and $E_\tau(p-1,1)$ exist as one-sided derivatives, and are computable with the formula above.

The thing left to show is that $\overline{E}(p+\tau)$ is strictly convex on $(0,\bar{p})$ and has the minimum at $\bar{p}$. We will first show that $\frac{\partial^2 E(p,\tau)}{\partial\tau^2} = E_{\tau\tau}(p,\tau) > 0$ for all $p+1 \leq \bar{p}$.

$$\begin{aligned} E_{\tau\tau}(p,\tau) &= \frac{w^\alpha}{T^{\alpha-1}} \cdot \alpha(\alpha-1)(h(p+1)-h(p))^2 \\ &\quad \cdot (\tau \cdot h(p+1) + (1-\tau)h(p))^{-\alpha-1} \end{aligned}$$

Thus $E_{\tau\tau}(p,\tau) > 0 \Leftrightarrow h(p+1)-h(p) > 0 \Leftrightarrow p+1 \leq \bar{p}$. It remains to check that

$$\begin{aligned} E_\tau(p,1) &\leq E_\tau(p+1,0) \\ \Leftrightarrow h(p+1)-h(p) &\geq h(p+2)-h(p+1) \end{aligned}$$

The last inequality is true for $p+2 \leq \bar{p}$ because h is concave and true for $p+1 = \bar{p}$ because $h(p+1) \geq h(p), h(p+2)$. Thus we have shown that $\overline{E}$ is strictly convex for $p+1 \leq \bar{p}$ and E_τ is strictly increasing for $p+1 \leq \bar{p}$.

The fact that $\overline{E}$ has its minimum at $\bar{p}$ directly comes from the fact that h has its maximum at $\bar{p}$. □

Definition 7.1.8. *For $1 \leq p \leq \bar{p}$ we define the left $\overrightarrow{E}(p) := E_\tau(p-1,1)$ and right $\overleftarrow{E}(p) := E_\tau(p,0)$ derivative of $\overline{E}$. For non-integer $0 < p+\tau < \bar{p}$ the left and right derivative are the same and we define $\overrightarrow{E}(p+\tau) := \overleftarrow{E}(p+\tau) := E_\tau(p,\tau) =: \overline{E}'(p+\tau)$. For $p+\tau > \bar{p}$ we also set $\overline{E}(p+\tau) = \overline{E}(\bar{p})$ as $\overline{E}(\bar{p})$ is the minimal energy the job can use. This also leads to $\overleftarrow{E}(\bar{p}) = 0$.*

With this definitions of the derivative and the left and right derivatives we can directly see that $\overline{E}(p+\tau)$ is continuously differentiable on $\mathbb{R}_{>0} \setminus \mathbb{N}$. Also the left derivative $\overrightarrow{E}$ is continuous from the left and the right derivative $\overleftarrow{E}$ is continuous from the right.

Lemma 7.1.9. *We have that $\overline{E}(0+\tau) \xrightarrow[\tau\to 0]{} \infty$ and $\overline{E}'(0+\tau) \xrightarrow[\tau\to 0]{} -\infty$ and $\overrightarrow{E}(p+\tau) \leq \overleftarrow{E}(p+\tau)\ \forall p+\tau \in (0,\bar{p}]$ and $\overrightarrow{E}(p+\tau), \overleftarrow{E}(p+\tau)$ are strictly increasing on $(0,\bar{p}]$.*

Proof.

$$
\begin{aligned}
E(p,\tau) &= \frac{w^\alpha}{T^{\alpha-1}} \cdot (\tau \cdot h(p+1) + (1-\tau)h(p))^{-\alpha+1} \\
E_\tau(p,\tau) &= -\frac{w^\alpha}{T^{\alpha-1}} \cdot (\alpha-1)(h(p+1)-h(p)) \\
&\quad \cdot (\tau \cdot h(p+1) + (1-\tau)h(p))^{-\alpha}
\end{aligned}
$$

As $h(1) = 1$ and $h(0) = 0$, we have:

$$
\begin{aligned}
\overline{E}(0+\tau) &= \frac{w^\alpha}{T^{\alpha-1} \cdot \tau^{\alpha-1}} \xrightarrow[\tau\to 0]{} \infty \\
\overleftarrow{E}(0+\tau) &= -\frac{w^\alpha(\alpha-1)}{T^{\alpha-1} \cdot \tau^{\alpha}} \xrightarrow[\tau\to 0]{} -\infty
\end{aligned}
$$

As $h(\bar{p}) \geq h(\bar{p}+1)$, we have $\overleftarrow{E}(\bar{p}) \geq 0$

The rest of the lemma was proven in the proof of Lemma 7.1.7. □

With Lemma 7.1.9 we can define the inversion of the derivative of the energy function:

Definition 7.1.10. *We have $\overline{E}$ as in Definition 7.1.6 and the left and right derivatives as in Definition 7.1.8. Then for any $c \in (-\infty, 0]$ we define $(\overline{E}')^{-1}(c) := p+\tau$ for $p+\tau \in (0,\bar{p}]$ with $\overrightarrow{E}(p+\tau) \leq c \leq \overleftarrow{E}(p+\tau)$.*

Lemma 7.1.11. *$(\overline{E}')^{-1}$ as defined in Definition 7.1.10 is a well-defined, continuous and monotonically increasing function on $(-\infty, 0]$.*

Proof. First we have to show that there exists only one $p^* + \tau^*$ for each c which fulfills $(\overline{E}')^{-1}(c) = p^* + \tau^*$.

$\overrightarrow{E}(p+\tau)$ and $\overleftarrow{E}(p+\tau)$ are strictly increasing on $(0,\bar{p}]$ and $\overrightarrow{E}(p+\tau) = \overleftarrow{E}(p+\tau) = \overline{E}'(p+\tau)$ for $\tau \neq 0$. Hence we have the following sequence:

$$\dots \overleftarrow{E}(p) \leq \overline{E}'(p+\tau) \leq \overrightarrow{E}(p+1) \leq \overleftarrow{E}(p+1) \dots$$

For all $\tau \in (0,1)$ and $p \in \{0,\dots,\bar{p}-1\}$. Thus there are two possibilities for c:

1. $\overrightarrow{E}(p) \leq c \leq \overleftarrow{E}(p)$ for a certain p, then we have as inverse: $(\overline{E}')^{-1}(c) = p+0$
2. $\overleftarrow{E}(p) < c < \overrightarrow{E}(p+1)$ for a certain p, in this case there exists a unique $\tau \in (0,1)$ with $\overline{E}'(p+\tau) = c$ and the inverse is $(\overline{E}')^{-1}(c) = p+\tau$

As only one case with only one p can occur, there exists only one inverse.

From the sequence above and the strict increase of $\overline{E}'(p+\tau)$ it is also clear that $(\overline{E}')^{-1}(c)$ is monotonically increasing.

As $\overline{E}'(p+\tau)$ is a strictly increasing continuous function, it is clear that $(\overline{E}')^{-1}(c)$ is continuous on $(\overleftarrow{E}(p), \overrightarrow{E}(p+1))$. From the construction of $\overleftarrow{E}(p)$ and $\overrightarrow{E}(p+1)$ and the fact that $(\overline{E}')^{-1}(c) = p$ for all $c \in [\overrightarrow{E}(p), \overleftarrow{E}(p)]$ we have continuity on $(-\infty, 0]$. □

We now summarize the findings of this section in one theorem:

Theorem 7. *We have a malleable job with a concave speedup function s, an amount of work w and an additional function $h : p \mapsto \sqrt[\alpha-1]{s^{\alpha}(p)/p}$ which is strictly monotonically increasing for all $p < \bar{p}$ (for a $\bar{p} \in \mathbb{N}$) and monotonically decreasing for all $p > \bar{p}$ and additionally concave on $(0,\bar{p}]$. This job has to run in the time interval $[0,T]$ on a parallel machine with identical cores which can change their operating frequency. While running on p cores for a time t, the job needs the energy amount $E = p \cdot f^{\alpha} \cdot t$ with $\alpha > 2$. If this job gets an average amount $x = p+\tau$ ($p = \lfloor x \rfloor$) of cores during $[0,T]$, the minimal possible amount of needed energy is $\overline{E}(p+\tau) = \frac{w^{\alpha}}{T^{\alpha-1}} \cdot (\tau \cdot h(p+1) + (1-\tau)h(p))^{-\alpha+1}$ for $p+\tau \leq \bar{p}$, otherwise it is $\frac{w^{\alpha}}{T^{\alpha-1}} \cdot h^{-\alpha+1}(\bar{p})$. The minimum energy amount is used if the job runs* *on p cores for a time $(1-\tau)T$ and on $p+1$ cores for a time τT or on $\bar{p}$ cores all the time. The function $\overline{E}(p+\tau)$ is strictly convex and strictly decreasing on $(0,\bar{p}]$ and has its minimum at $\bar{p}$ and $\overline{E}(0+\tau) \xrightarrow[\tau\to 0]{} \infty$. Also $\overline{E}(p+\tau)$ is continuously differentiable on $\mathbb{R}_{>0}\setminus\mathbb{N}$, and the left derivative $\overrightarrow{E}$ is continuous from the left and the right derivative $\overleftarrow{E}$ is continuous from the right.*

7.1.2 Enhancements

In this section we look at different modelling enhancements and how they can be included into our model and frequency scheduling algorithm. General enhancements are described in Section 5.2.6. This section considers only energy-specific enhancements.

Energy Consumption of Frequency-Independent Parts

The easiest thing to include in our model is the energy consumption of frequency-independent core parts by $E = p\cdot f^{\alpha}\cdot T + c\cdot p\cdot T$ (for integer core amounts p). In this case we need a constant factor as the "correct" unit sizes of E might be different for $p\cdot f^{\alpha}\cdot T$ and $p\cdot T$. Such frequency-independent core parts might be the memory controllers or power supply units which always consume the same power as long as the core is working. Our article [117] (joint work with Peter Sanders) also looks briefly into this kind of enhancement, but here we use a slightly different approach.

In order to find out what we have to change in our scheduling, we look through the properties of the schedule. Let us first take a look at a single job with work amount w, speedup function s and with $\bar{p}$ as core amount of the minimal energy consumption in the basic model. The energy consumption is $\overline{E}(p+\tau) = \frac{w^{\alpha}}{T^{\alpha-1}}\cdot(\tau\cdot h(p+1)+(1-\tau)h(p))^{-\alpha+1}$ in the basic model. If we add the term $c(p+\tau)T$ (possibly non-integer core amounts are used) to the energy function, the new core amount $\bar{p}_{new}$ with the minimal energy consumption can be smaller. $\overline{E}_{new}(p+\tau) = \frac{w^{\alpha}}{T^{\alpha-1}}\cdot(\tau\cdot h(p+1)+(1-\tau)h(p))^{-\alpha+1}+c(p+\tau)T$ remains strictly convex on $(0,\bar{p}]$ as the second derivative does not change and thus $\bar{p}_{new}$ can be computed by simply finding the $p+\tau$ with $0 = \overline{E}'_{new}(p+\tau) = \overline{E}'(p+\tau)+c\cdot T$. If the sum of the $\bar{p}_{new}$ of all jobs is less than or equals m, then we get the easy solution of giving each job an amount of $\bar{p}_{new}$ cores. If the sum of the $\bar{p}_{new}$ for all jobs is larger than the available amount of m cores, then an optimal schedule uses all of the available cores as $\overline{E}_{new}$ is strictly decreasing on $(0,\bar{p}_{new}]$. In this case we already know the additional energy usage by the newly modelled parts which is $c\cdot m\cdot T$. All the derivatives of the energy functions get the same additive term compared to the basic model. Hence we can just use the optimal solution computed with the basic model.

Discrete Frequency Steps

On real machines it is unfortunately often not possible to run on every clock speed in $\mathbb{R}_{>0}$ but only on a discrete set of clock speeds. We assume that an ordered set of usable frequencies is given by $v_1 < \cdots < v_k$. Ishihara and Yasuura [72] showed that in case of discrete available frequencies and voltages an optimal solution for one job on one core only uses the two adjacent discrete frequencies of the optimal continuous frequency. In our more complex case with several parallel, malleable jobs on a parallel machine we will not be able to show optimality but only an approximation. We assume that we have an optimal schedule computed for continuous frequencies. This schedule will now be adapted to the case of discrete frequencies. It is clear that the optimal schedule for continuous frequencies uses at most as much energy as the optimal schedule for discrete frequencies as the schedule for the discrete case is a feasible solution for the continuous case.

We assume that we have a given time interval of length T in which a job runs on p cores with a frequency f. The optimal schedule of a job (in the continuous case) only consists of such time intervals (the T here does usually not coincide with the T used for the whole schedule). We compute the approximation for each such interval. Let v_i and v_{i+1} be the discrete frequencies such that $v_i < f < v_{i+1}$ (we assume that such frequencies exist). If we run the job for a time $\frac{f-v_i}{v_{i+1}-v_i}T$ on frequency v_{i+1} and for a time $(1-\frac{f-v_i}{v_{i+1}-v_i})T$ on frequency v_i, the same amount of work is done as if the job is run for a time T on frequency f. The only thing that remains to be checked is the relative additional amount of used energy. Let E_c be the energy usage of the continuous solution and E_d the energy usage of the discrete solution for our considered interval, then we have: $E_c = p \cdot f^\alpha \cdot T$ and $E_d = p\frac{f-v_i}{v_{i+1}-v_i}v_{i+1}^\alpha \cdot T + p(1-\frac{f-v_i}{v_{i+1}-v_i})v_i^\alpha \cdot T$. We now give an upper bound for the relative energy usage of the discrete solution E_d/E_c:

$$\begin{aligned}\frac{E_d}{E_c} &= \Big(\frac{f-v_i}{v_{i+1}-v_i}v_{i+1}^\alpha + (1-\frac{f-v_i}{v_{i+1}-v_i})v_i^\alpha\Big)\cdot f^{-\alpha}\\ &= \Big(\frac{v_{i+1}^\alpha - v_i^\alpha}{v_{i+1}-v_i}(f-v_i)+v_i^\alpha\Big)\cdot f^{-\alpha}\end{aligned}$$

Let us take a look at the function $g : f \mapsto f^\alpha$ for $\alpha > 2$. The gradient $g'(f) = \alpha f^{\alpha-1}$ is strictly monotonically increasing for $f \in (v_i, v_{i+1})$. $\frac{v_{i+1}^\alpha - v_i^\alpha}{v_{i+1}-v_i}$ is the "average" gradient of g on (v_i, v_{i+1}). Hence we have:

$$\begin{aligned} f^\alpha \geq& v_i^\alpha + \alpha v_i^{\alpha-1}(f - v_i) \\ \frac{v_{i+1}^\alpha - v_i^\alpha}{v_{i+1} - v_i} \leq& \alpha v_{i+1}^{\alpha-1} \end{aligned}$$

With this we get:

$$\begin{aligned} \frac{E_d}{E_c} \leq& \frac{\alpha v_{i+1}^{\alpha-1}(f - v_i) + v_i^\alpha}{\alpha v_i^{\alpha-1}(f - v_i) + v_i^\alpha} \\ & \text{by setting } \; v_{i+1} = \delta v_i \; \text{ we get:} \\ \frac{E_d}{E_c} \leq& \frac{\alpha \delta^{\alpha-1} v_i^{\alpha-1}(f - v_i) + v_i^\alpha}{\alpha v_i^{\alpha-1}(f - v_i) + v_i^\alpha} \\ =& \frac{\alpha \delta^{\alpha-1}(f - v_i) + v_i}{\alpha(f - v_i) + v_i} \end{aligned}$$

As $\delta^{\alpha-1} > 1$ and $\alpha, v_i, (f - v_i) > 0$ the value of $\frac{\alpha\delta^{\alpha-1}(f-v_i)+v_i}{\alpha(f-v_i)+v_i}$ is increasing for an increasing value of $f - v_i$. Hence we can replace $f - v_i$ by the larger $(\delta - 1)v_i$ (as $f < v_{i+1}$) and we get:

$$\frac{E_d}{E_c} \leq \frac{\alpha \delta^{\alpha-1}(\delta - 1) + 1}{\alpha(\delta - 1) + 1}$$

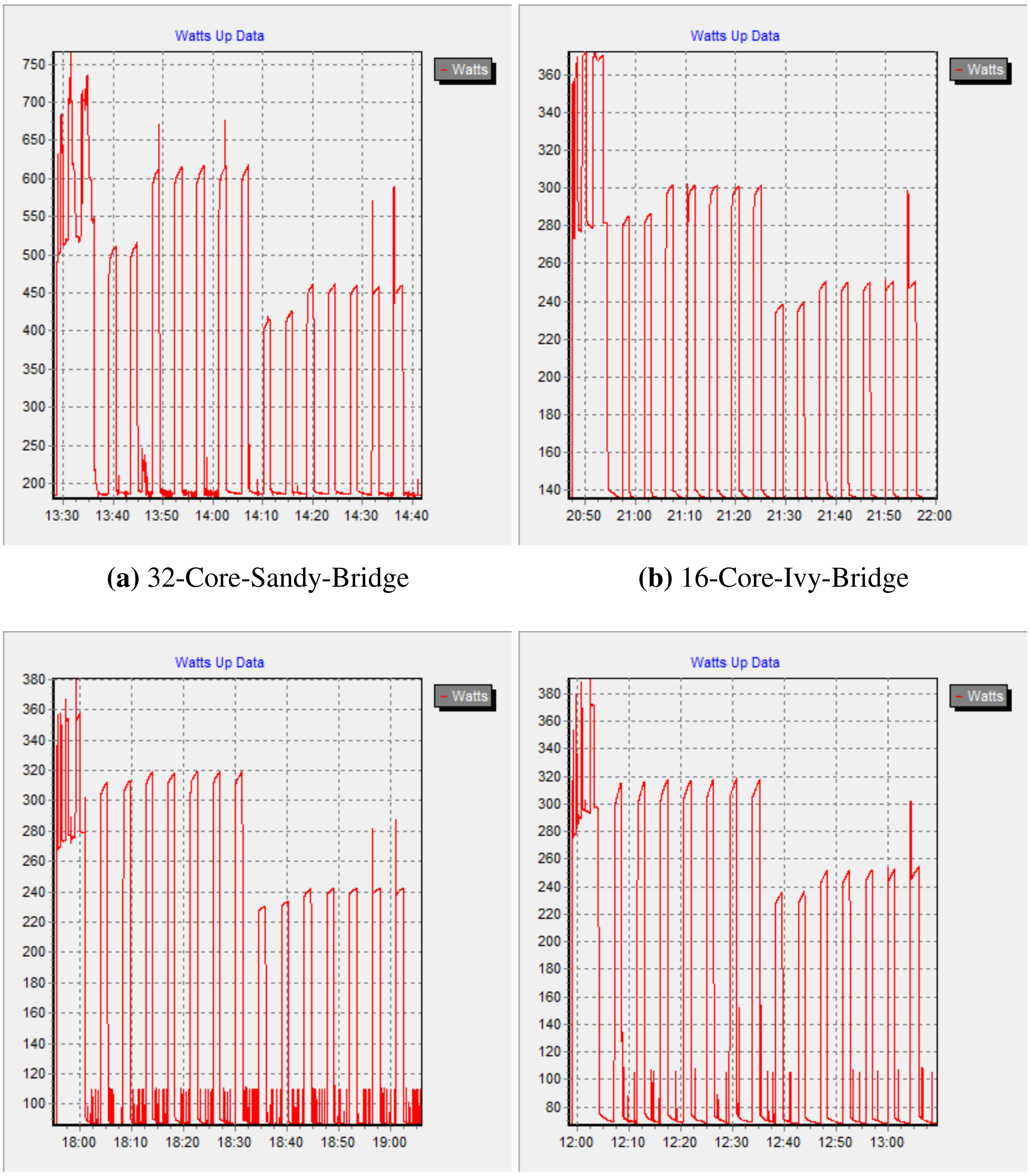
The approximation quality increases for a decreasing α. We assume $\alpha = 3$ for the remainder of this section. δ is the factor between different discrete frequency steps. If we know an upper bound for δ, then we can compute the ratio E_d/E_c which is also an upper bound for the approximation ratio. If each discrete frequency is at most 10% larger than the previous one (10 such steps are equivalent to a factor of 2.59), we get $E_d/E_c \leq 1.05$. Even for $\delta = 1.3$ we still have $E_d/E_c \leq 1.33$.

7.2 Memory Hierarchies

In Chapter 6 we look at the memory hierarchy as the efficient usage (e. g. through optimized scheduling) becomes more and more important for the overall performance and efficiency. In this section we take a closer look at the energy consumption of memory operations. Also in case of energy consumption the memory hierarchy has a considerable impact which is too large compared to the energy usage of the computation units to be ignored. There might be possibilities to measure the power consumption of processors and memory modules by specialized methods, but the electricity bill and air conditioning efforts due to the machine usage are determined by the power consumption of the whole machine. Hence we measure the power consumption of the whole machine while performing different operations.

We measure the power consumption of the machines by a "watts up? PRO" from ThinkTank Energy Products Inc.. The measured machines are the four machines used for the experiments in Section 6.1. The measurements take place on otherwise idle machines. The test operations/programs are the linear-test and the chain-test on all cores which are introduced in Section 6.1. These tests perform mainly memory operations and only a low amount of computations and almost no floating-point computations. In order to have a comparison with a floating-point-intensive workload, we also run the LINPACK benchmark from Intel's Math Kernel Library 11.2 for Linux (xlinpack_xeon64). On all machines the LINPACK benchmark is run twice for a number of equations and a leading array size of 10 000 and then once for a number of equations and a leading array size of 25 000, 30 000 and 40 000. On all machines the run for a size of 40 000 takes more time than all other runs together. The reached GFlop/s are (numbers for the test with size 40 000): 32-Core-Sandy-Bridge: 447 GFlop/s, 16-Core-Ivy-Bridge: 336 GFlop/s, 24-Core-Haswell: 734 GFlop/s, 32-Core-Broadwell: 959 GFlop/s. For the linear-test and the chain-test we select a number of iterations over the array such that a measured test run roughly takes 10 seconds. The tests are run in one program run per array size during which 6 measured test runs are performed. The reached bandwidths (of core 0) of these tests differ less than 2% from those reported in the "all" rows in Table 6.1 (basic experiment). Between the different tests (LINPACK or memory tests) we allow the machine to cool down for 180 seconds (the longer cool down time compared to the experiments in Section 6.1 is in order to produce larger gaps in the power consumption graphs). For each machine we measure one run starting with the LINPACK test continuing with the linear-tests for the different array sizes (in that order: 1 MiB, 2 MiB, 10 MiB, 20 MiB, 100 MiB, 200 MiB, 1000 MiB) and finishing with the chain-tests for the different array sizes (in the same order as the linear-tests). The graphs of the measured power consumptions for each machine are presented in Figure 7.1.

In Figure 7.1 the base lines are set to the lowest power consumption during the test run which is just the idle power consumption of the respective machine. In all test runs the LINPACK benchmark has the highest power consumption (at least the peaks). For all machines the additional power usage (additional to the idle power) of the chain-tests is about half of the additional power usage of the LINPACK benchmarks. The chain-tests that can run within the cache (1 MiB, 2 MiB) use less power than those which have to use the main memory. The power consumption



(a) 32-Core-Sandy-Bridge

(b) 16-Core-Ivy-Bridge

(c) 24-Core-Haswell

(d) 32-Core-Broadwell

Figure 7.1. The power consumption of the memory experiments. All tests are started with some LINPACK benchmark runs from Intel's Math Kernel Library 11.2 for Linux, then all cores perform the linear-test on 1 MiB, 2 MiB, 10 MiB, 20 MiB, 100 MiB, 200 MiB, 1000 MiB, then the cores perform the chain-test for the same sequence of array sizes. Between the 15 different tests the power consumption always drops near the baseline (due to the 180 second breaks). In all pictures above we have on the left side a wide peak above the baseline which is the power consumption of the LINPACK benchmark. To the right the peak is followed by the 14 peaks for the memory experiments.

of the linear-tests is for all machines in the middle between the chain-tests and the LINPACK benchmarks. Also for the linear-tests the power consumption of those tests which can run within the cache is at most the power usage of those which have to use the main memory. On the four-socket machine 32-Core-Sandy-Bridge the differences between the tests within the cache and those with larger arrays are especially large.

Compared to the floating-point-intensive benchmarks from Intel's MKL which probably are near the peak performance of the machine the power consumption of the memory tests is quite high. This shows that memory operations have the potential to use a significant amount of the machine's total power consumption. The tests on arrays of size 1 MiB and 2 MiB which fit into L3-cache produce a 4 to 5 times higher bandwidth (see Table 6.1) and use at most as much power as the tests on larger array sizes. This indicates that the high energy usage of the memory tests cannot come from the computation effort of the tests (which is proportional to the bandwidth) but from the memory accesses themselves. The bandwidth of the linear-tests is between 37 and 64 times as high as the bandwidth of the chain-tests, but the linear-tests' additional energy usage is only between 1.3 and 1.6 times the energy usage of the chain-tests. This indicates that a random access is energetically much more costly than reading a single element of a linearly read array.

In Section 6.2.2 we present a cache- and NUMA-optimized scheduling for the LU-decomposition. We also measure the power consumption of our optimized algorithm and the competitor. Our algorithm saves around 2-3% power compared to our competitor even though it is faster (up to more than 10% on one machine). This fits the findings of this section that memory operations are power-intensive and further indicates that applications can be made more energy-efficient by optimizing their cache- and NUMA-locality.

8

Conclusion

8.1 Summary

This work provides results in all four major research directions identified in Section 3.4.3: the interplay and distribution of decision makers (Chapter 4), the efficient schedule computation (Chapter 5), efficient scheduling regarding the memory hierarchy (Chapter 6) and energy-efficiency (Chapter 7).

The main result is the fast and efficient scheduling algorithm for malleable jobs, which can be used for the objective of minimizing the maximum as well as for the objective of minimizing the sum (see Section 5.2.3). The objectives are the sum or maximum of some resource-dependent functions for each job. Our scheduling algorithm finds the optimal resource distribution between these jobs. We provide a range of exemplary problems which can be solved by our scheduling algorithm including energy minimization and running-time minimization (see Section 5.2.5). Four conditions are identified which need to be fulfilled by the problem in order to use our scheduling algorithm. The first two of these conditions are restrictions of the problem class, whereas the second two require the existence and maximal running time of some supplementary algorithms. In case that one of the first two conditions is not met, we also present examples which result in NP-hard problems (in the end of Section 5.2.3 and Section 5.2.5). Hence these conditions seem to be justified in the sense that without them finding an optimal solution in polynomial time would not be possible (unless $P = NP$). We also show how our scheduling algorithm can be parallelized and thus present the first parallelized scheduling algorithm for parallel job scheduling (see Section 5.2.4).

A topic closely related to the scheduling of malleable jobs is the realization of malleable jobs. In this work we are able to present one of the first implementations of a malleable job: malleable sorting (see Section 4.6). An interesting part of malleable jobs is the interface to the operating system scheduler. We see this interface as part of the communication between different system components responsible for efficiency-related decisions. In Chapter 4 we take a look at the interplay of these decision makers and the role of malleable jobs. Our interest in this system-wide topic was fueled by our participation in InvasIC (see Section 4.4 and Section 2.7).

The main scheduling result is also used by us to solve a problem for the energy-efficient scheduling of malleable jobs (one of the examples in Section 5.2.5, using results from Section 7.1.1 and further enhancements in Section 7.1.2). For a set of jobs with common release time and deadline and speedup functions with some restrictions it is possible to compute a distribution of the available cores such that the energy consumption is minimized. This is one of the first results for energy-efficient scheduling of parallel jobs which computes the optimal solution to the given problem.

Memory operations become more and more relevant compared to computations regarding efficiency and speed as the increase of computing throughput and speed is faster than the increase of memory throughput and speed (see Section 2.2.1). Our measurements described in Section 7.2 also indicate that memory operations are highly relevant for power consumption. Hence future scheduling algorithms have to consider the properties of memory operations in order to produce good results. We present some measurements in this work that provide a basic understanding of the properties of current memory hierarchies (see Section 6.1). As an example of the relevance of memory operations we also demonstrate how a good memory scheduling can improve the performance of the LU-decomposition of dense matrices (see Section 6.2.2) even though the LU-decomposition is typically assumed to be a compute-bound application.

We also take a look into heuristic scheduling algorithms (see Section 5.3). Together with Felix Brandt and Markus Völker, the author of this work took part in the ROADEF/EURO challenge 2012 [1] which dealt with machine reassignment problems. Our team finished *second* in the final round among all teams in the junior category (11 teams in that category). Teams were placed in the junior category if no team member had a Ph. D. yet. Among all teams our team won the *fourth* place (28 participating teams in the final round) [1].

8.2 Outlook

In the future computing devices will spread into many more areas than today. Many of these computing devices will execute different applications while only using cheap chips and little energy. On the other side of the computing spectrum super-computers will become even more powerful. In both cases the performance and efficiency have to increase dramatically compared to today's devices of comparable cost and size. Also the physical limits for further performance improvements are coming closer (for example the Power-Wall as described in Section 2.2.1). These problems ask for an increased system efficiency, and an improved scheduling might contribute a reasonable share to that efficiency gain. On the other hand improvements through scheduling are often hindered by interfaces introduced a long time ago which hinder the coordination of scheduling decisions across system components (see Section 4.1). This is an example of path dependence as described in Section 2.3.

The number of produced and used computing devices with many parallel cores which work on parallel applications is likely to increase in the future. Such devices will include smart-phones, personal computers and servers which all already have parallel cores today. This will change the relative importance of development cost and the cost dependent on the usage of a single unit. User satisfaction (influenced for example through waiting times) and the energy usage of a single device are becoming more important, the development effort for improved scheduling solutions to improve these things is becoming less important. Hence there might be enough need and money to overcome the path dependence and to build new scheduling systems and adapt other system components according to them (this might also justify more complex models and scheduling algorithms, see Section 5.1).

As many new systems act in embedded applications, the users expect them to act reproducibly and thus to fulfill some performance guarantees. Hence scheduling algorithms with provable guarantees will become more important. This demands a further theory research in order to develop such scheduling algorithms. Especially the boundary between polynomially (efficiently) solvable problems and NP-hard problems has to be further investigated to decide which problems allow optimal solutions.

Memory operations become more and more important regarding performance as well as energy consumption, thus there is a need to include the memory/cache behavior into the models used for the development of scheduling algorithms. A long-term goal of this would be scheduling algorithms which consider the memory properties and provide performance guarantees for the system behavior.

Also many devices will be battery-powered and the general cost of electricity might also increase. Hence the importance of power consumption will also increase. This calls for scheduling algorithms which minimize the energy consumption. Regarding the power demand of memory operations, scheduling algorithms with power objectives should also include the management of memory and cache.

Even today different decision makers take part in the scheduling of a computing system (application-internal schedulers, OS scheduler, see also Section 4.3). In the future the number of these decision makers might increase and, what is even more important, the coordination between these decision makers will be much more relevant for the efficiency of the system (for example because of the usage of common resources like memory bandwidth or caches). Hence ways to coordinate different parts of the scheduling system have to be developed. Also the flexibility of applications to react to the system load or other system properties will be important for performance and efficiency. It might be too difficult for many programmers to write programs that can adapt flexibly to changing conditions and coordinate with the system scheduler, but already today many libraries for parallelization (OpenMP) or the efficient solution of expensive sub-problems (MCSTL) exist. We expect such libraries to be made flexible and to be used by most applications for expensive (in terms of power usage or computing effort) sub-problems.

8.3 Open Problems/Future Work

The outlook indicates that there are some areas which need future work. In each of these areas there is a lot of research necessary.

The importance of memory hierarchies both in terms of performance as well as power consumption shows that they should be included into the models used for scheduling and into scheduling algorithms.

Besides the inclusion of memory aspects, the theory of malleable jobs (or related flexible applications) also leaves many open problems for development. This ranges from various online problems, scheduling of a mixture between malleable, moldable and fixed-size jobs, approximation algorithms and the handling of a larger class of objective functions for the jobs, for example job functions that are not convex any more but nearly convex.

In order to implement the new scheduling results in practice, new flexible applications and appropriate operating systems are needed (maybe even special hardware support). The increased flexibility of the applications might be put into practice by flexible libraries for computation-intensive basic algorithms. These basic algorithms with resource flexibility mostly still have to be developed. Also new interfaces and the coordination between different performance- and efficiency-relevant decision makers need more development. Two important sub-problems are the prevention of overloaded decision makers by splitting up scheduling decisions and the handling of shared memory resources by new control mechanisms.

These new structures then might be introduced into an algorithm engineering cycle (see Section 2.6.1) of repeated modelling, analysis, (re-)implementation and experiments.

Bibliography

[1] H. Murat Afsar, Christian Artigues, Eric Bourreau, and Safia Kedad-Sidhoum. Machine reassignment problem: the ROADEF/EURO challenge 2012. *Annals of Operations Research*, 242(1):1–17, 2016.

[2] Anant Agarwal, John Hennessy, and Mark Horowitz. An Analytical Cache Model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[3] Ravindra K. Ahuja, Özlem Ergun, James B. Orlin, and Abraham P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1–3):75–102, 2002.

[4] Susanne Albers, Fabian Müller, and Swen Schmelzer. Speed Scaling on Parallel Processors. *Algorithmica*, 68(2):404–425, 2012.

[5] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Spring Joint Computer Conference*, pages 483–485. ACM, 1967.

[6] A. K. Amoura, E. Bampis, C. Kenyon, and Y. Manoussakis. Scheduling Independent Multiprocessor Tasks. *Algorithmica*, 32(2):247–261, 2002.

[7] W. Brian Arthur. *Increasing Returns and Path Dependence in the Economy*. The University of Michigan Press, 2011.

[8] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, 2006.

[9] Kenneth R. Baker and Dan Trietsch. *Principles of Sequencing and Scheduling*. John Wiley & Sons, 2009.

[10] Timo Bingmann. Parallel Memory Bandwidth Measurement and Benchmark, 2013. https://panthema.net/2013/pmbw/.

[11] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling Irregular Parallel Computations on Hierarchical Caches. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–366. ACM, 2011.

[12] Guy E. Blelloch and Phillip B. Gibbons. Effectively Sharing a Cache Among Threads. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 235–244. ACM, 2004.

[13] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[14] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple but Effective Techniques for NUMA Memory Management. In *Symposium on Operating Systems Principles (SOSP)*, pages 19–31. ACM, 1989.

[15] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[16] Stephen P. Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2008.

[17] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek. Reinventing Scheduling for Multicore Systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*. USENIX, 2009.

[18] Felix Brandt, Jochen Speck, and Markus Völker. Constraint-based large neighborhood search for machine reassignment. *Annals of Operations Research*, 242(1):63–91, 2014.

[19] James Bruno, Edward G. Coffman Jr., and Ravi Sethi. Scheduling Independent Tasks to Reduce Mean Finishing Time. *Communications of the ACM*, 17(7):382–387, 1974.

[20] Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. Malleable Invasive Applications. In *Working Conference on Programming Languages (ATPS)*, pages 123–126, 2015.

[21] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[22] Jacek Błażewicz, Mieczysław Drabowski, and Jan Węglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, C-35(5):389–393, 1986.

[23] Jacek Błażewicz, Klaus H. Ecker, Erwin Pesch, Günter Schmidt, and Jan Węglarz. *Handbook on Scheduling : From Theory to Applications*. Springer, 2007.

[24] Jacek Błażewicz, Mikhail Y. Kovalyov, Maciej Machowiak, Denis Trystram, and Jan Węglarz. Preemptable malleable task scheduling problem. *IEEE Transactions on Computers*, 55(4):486–490, 2006.

[25] Jacek Błażewicz, Jan Karel Lenstra, and AHG Rinnooy Kan. Scheduling Subject to Resource Constraints: Classification and Complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983.

[26] Jacek Błażewicz, Maciej Machowiak, Jan Węglarz, Mikhail Y. Kovalyov, and Denis Trystram. Scheduling Malleable Tasks on Parallel Processors to Minimize the Makespan. *Annals of Operations Research*, 129(1-4):65–80, 2004.

[27] Jacek Błażewicz, Jan Węglarz, and Mieczysław Drabowski. Scheduling independent 2-processor tasks to minimize schedule length. *Information Processing Letters*, 18(5):267–273, 1984.

[28] Michael Casey and Robert Hackett. The top 10 biggest R&D spenders worldwide. Fortune.

[29] Ho Leung Chan, Jeff Edmonds, and Kirk Pruhs. Speed Scaling of Processes with Arbitrary Speedup Curves on a Multiprocessor. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 1–10. ACM, 2009.

[30] Jian-Jia Chen, Heng-Ruey Hsu, Kai-Hsiang Chuang, Chia-Lin Yang, Ai-Chun Pang, and Tei-Wei Kuo. Multiprocessor energy-efficient scheduling with task migration considerations. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 101–108. IEEE, 2004.

[31] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos

Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling Threads for Constructive Cache Sharing on CMPs. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 105–115. ACM, 2007.

[32] Hsiang-Yun Cheng, Chung-Hsiang Lin, Jian Li, and Chia-Lin Yang. Memory Latency Reduction via Thread Throttling. In *International Symposium on Microarchitecture (MICRO)*, pages 53–64. IEEE, 2010.

[33] Richard Cole. Parallel Merge Sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[34] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[35] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 381–394. ACM, 2013.

[36] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[37] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Symposium on Communications Architectures & Protocols (SIGCOMM)*, pages 1–12. ACM, 1989.

[38] Danny Dolev, Eli Upfal, and Manfred K. Warmuth. The Parallel Complexity of Scheduling with Precedence Constraints. *Journal of Parallel and Distributed Computing*, 3(4):553–576, 1986.

[39] Maciej Drozdowski. On the Complexity of Multiprocessor Task Scheduling. *Bulletin of the Polish Academy of Sciences. Technical sciences*, 43(3):381–392, 1995.

[40] Jianzhong Du and Joseph Y-T. Leung. Complexity of Scheduling Parallel Task Systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, 1989.

[41] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Bandwidth Bandit: Quantitative Characterization of Memory Contention.

In *Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE, 2013.

[42] Dror G. Feitelson and Larry Rudolph. Distributed Hierarchical Control for Parallel Processing. *Computer*, 23(5):65–77, 1990.

[43] Dror G. Feitelson and Larry Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 1–26. Springer, 1996.

[44] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and Practice in Parallel Job Scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 1–34. Springer, 1997.

[45] Patrick Flick, Peter Sanders, and Jochen Speck. Malleable Sorting. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 418–426. IEEE, 2013.

[46] Kyle Fox, Sungjin Im, and Benjamin Moseley. Energy Efficient Scheduling of Parallelizable Jobs. In *Symposium on Discrete Algorithms, (SODA)*, pages 948–957. SIAM, 2013.

[47] Erich Frese. *Grundlagen der Organisation: Konzept - Prinzipien - Strukturen*. Gabler, 1998.

[48] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms*, 8(1):4:1–4:22, 2012.

[49] Michael R. Garey and Ronald L. Graham. Bounds for Multiprocessor Scheduling with Resource Constraints. *SIAM Journal on Computing*, 4(2):187–200, 1975.

[50] Michael R. Garey and David S. Johnson. “ Strong ” NP-Completeness Results: Motivation, Examples, and Implications. *Journal of the ACM*, 25(3):499–508, 1978.

[51] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[52] Haris Gavranović, Mirsad Buljubašić, and Emir Demirović. Variable Neighborhood Search for Google Machine Reassignment problem. *Electronic Notes in Discrete Mathematics*, 39:209–216, 2012.

[53] Gerhard Goos. *Vorlesungen über Informatik: Band 2: Objektorientiertes Programmieren und Algorithmen*. Springer, 2001.

[54] Ronald L. Graham. Bounds for Certain Multiprocessing Anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[55] Ronald L. Graham, Eugene L. Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[56] Albert G. Greenberg and Neal Madras. How Fair is Fair Queuing? *Journal of the ACM*, 39(3):568–598, 1992.

[57] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation : P-Completeness Theory*. Oxford University Press, 1995.

[58] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, 1988.

[59] Volker Hamscher, Uwe Schwiegelshohn, Achim Streit, and Ramin Yahyapour. Evaluation of Job-Scheduling Strategies for Grid Computing. In *Grid Computing — GRID*, pages 191–202. Springer, 2000.

[60] Tim Harris, Martin Maas, and Virendra J. Marathe. Callisto: Co-Scheduling Parallel Runtime Systems. In *European Conference on Computer Systems (EuroSys)*, pages 24:1–24:14. ACM, 2014.

[61] David Helmbold and Ernst Mayr. Fast scheduling algorithms on parallel computers. Technical Report STAN-CS-84-1025, Department of Computer Science, Stanford University, 1984.

[62] David Helmbold and Ernst Mayr. Two Processor Scheduling is in $\mathcal{NC}$. *SIAM Journal on Computing*, 16(4):747–759, 1987.

[63] Jörg Henkel, Andreas Herkersdorf, Lars Bauer, Thomas Wild, Michael Hübner, Ravi Kumar Pujari, Artjom Grudnitsky, Jan Heisswolf, Aurang Zaib, Benjamin Vogel, Vahid Lari, and Sebastian Kobbe. Invasive Manycore Architectures. In *Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 193–200. IEEE, 2012.

[64] John L. Hennessy and David A. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann, 2012.

[65] Sören Henning, Klaus Jansen, Malin Rau, and Lars Schmarje. Complexity and Inapproximability Results for Parallel Task Scheduling and Strip Packing. *arXiv:1705.04587 [cs]*, 2017. to appear at CSR 2018.

[66] Christian Hesse. *Angewandte Wahrscheinlichkeitstheorie: eine fundierte Einführung mit über 500 realitätsnahen Beispielen und Aufgaben*. Vieweg, 2003.

[67] Juraj Hromkovič. *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation and Heuristics*. Springer, 2004.

[68] Te C. Hu. Parallel Sequencing and Assembly Line Problems. *Operations Research*, 9(6):841–848, 1961.

[69] Jonathan F. Hutchings. *Project Scheduling Handbook*. Dekker, 2004.

[70] Intel Corporation - 2014 Annual Report.

[71] Improving Real-Time Performance by Utilizing Cache Allocation Technology, 2015. Intel Corporation.

[72] Tohru Ishihara and Hiroto Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In *International Symposium on Low Power Electronics and Design (ISLEPED)*, pages 197–202. ACM, 1998.

[73] Klaus Jansen. Scheduling Malleable Parallel Tasks: An Asymptotic Fully Polynomial Time Approximation Scheme. *Algorithmica*, 39(1):59–81, 2004.

[74] Klaus Jansen. A $(3/2+\varepsilon)$ Approximation Algorithm for Scheduling Moldable and Non-Moldable Parallel Tasks. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 224–235. ACM, 2012.

[75] Klaus Jansen and Felix Land. Scheduling Monotone Moldable Jobs in Linear Time. *arXiv:1711.00103 [cs]*, 2017. to appear at IPDPS 2018.

[76] Klaus Jansen and Lorant Porkolab. Linear-Time Approximation Schemes for Scheduling Malleable Parallel Tasks. *Algorithmica*, 32(3):507–520, 2002.

[77] Klaus Jansen and Lorant Porkolab. Computing optimal preemptive schedules for parallel tasks: linear programming approaches. *Mathematical Programming*, 95(3):617–630, 2003.

[78] Klaus Jansen and Malin Rau. Closing the gap for pseudo-polynomial strip packing. *arXiv:1712.04922 [cs]*, 2017.

[79] Klaus Jansen and Ralf Thöle. Approximation Algorithms for Scheduling Parallel Jobs. *SIAM Journal on Computing*, 39(8):3571–3615, 2010.

[80] Klaus Jansen and Hu Zhang. Scheduling malleable tasks with precedence constraints. *Journal of Computer and System Sciences*, 78(1):245–259, 2012.

[81] Berit Johannes. Scheduling parallel jobs to minimize the makespan. *Journal of Scheduling*, 9(5):433–452, 2006.

[82] Selmer Martin Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68, 1954.

[83] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[84] Christoph W. Kessler, Nicolas Melot, Patrick Eitschberger, and Jörg Keller. Crown Scheduling: Energy-Efficient Resource Allocation, Mapping and Discrete Frequency Scaling for Collections of Malleable Streaming Tasks. In *International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 215–222. IEEE, 2013.

[85] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. DistRM: Distributed Resource Management for On-Chip Many-Core Systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 119–128. ACM, 2011.

[86] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical Report CSE 2008-13, University of Notre Dame, 2008.

[87] Fanxin Kong, Nan Guan, Qingxu Deng, and Wang Yi. Energy-efficient Scheduling for Parallel Real-Time Tasks Based on Level-Packing. In *Symposium on Applied Computing (SAC)*, pages 635–640. ACM, 2011.

[88] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, 22(1):15–44, 2010.

[89] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and Comparison of the Task Graph Scheduling Algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[90] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[91] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[92] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. Complexity of Machine Scheduling Problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[93] Joseph Y.-T. Leung, editor. *Handbook of Scheduling : Algorithms, Models, and Performance Analysis*. Chapmann & Hall/CRC, 2004.

[94] Minming Li and Frances F. Yao. An Efficient Algorithm for Computing Optimal Discrete Voltage Schedules. *SIAM Journal on Computing*, 35(3):658–671, 2005.

[95] Giuseppe Lipari and Enrico Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2):257–269, 2005.

[96] Robert Love. *Linux Kernel Development : [A thorough guide to the design and implementation of the Linux kernel]*. Addison-Wesley, 2010.

[97] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux Scheduler: a Decade of Wasted Cores. In *European Conference on Computer Systems (EuroSys)*, pages 1:1–1:16. ACM, 2016.

[98] Walter Ludwig and Prasoon Tiwari. Scheduling Malleable and Nonmalleable Parallel Tasks. In *Symposium on Discrete Algorithms (SODA)*, pages 167–176. SIAM, 1994.

[99] Chris A. Mack. Fifty Years of Moore’s Law. *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202–207, 2011.

[100] Tobias Maier, Peter Sanders, and Jochen Speck. Locality Aware DAG-Scheduling for LU-Decomposition. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 82–92. IEEE, 2015.

[101] Loris Marchal, Bertrand Simon, Oliver Sinnen, and Frédéric Vivien. Malleable task-graph scheduling with a practical speed-up model. Research Report RR-8856, ENS de Lyon, 2016. <hal-01274099>.

[102] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wiley Publishing, 2008.

[103] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995.

[104] Robert McNaughton. Scheduling with Deadlines and Loss Functions. *Management Science*, 6(1):1–12, 1959.

[105] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 1965.

[106] Gregory Mounie, Christophe Rapine, and Denis Trystram. A 3/2-Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks. *SIAM Journal on Computing*, 37(2):401–412, 2007.

[107] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.

[108] Dirk W. Oetting. *Auftragstaktik : Geschichte und Gegenwart einer Führungskonzeption*. Report Verlag, 1993.

[109] David A. Padua, editor. *Encyclopedia of Parallel Computing*. Springer, 2011.

[110] Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. Design Principles for End-to-End Multicore Schedulers. In *Workshop on Hot Topics in Parallelism (HotPar)*. USENIX, 2010.

[111] PLASMA: Main Page. https://icl.bitbucket.io/plasma/ (accessed: 05.07.2017).

[112] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly, 2007.

[113] ROADEF/EURO Challenge 2012: Machine Reassignment. *http://challenge.roadef.org/2012/files/problem_definition_v1.pdf* (Problem Definition).

[114] ROADEF/EURO Challenge 2012: Machine Reassignment. *http://challenge.roadef.org/2012/en/reponse.php* (Submission Page, accessed: 30.09.2016).

[115] Peter Sanders. Algorithm Engineering – An Attempt at a Definition. In *Efficient Algorithms*, pages 321–340. Springer, 2009.

[116] Peter Sanders and Jochen Speck. Efficient Parallel Scheduling of Malleable Tasks. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1156–1166. IEEE, 2011.

[117] Peter Sanders and Jochen Speck. Energy Efficient Frequency Scaling and Scheduling for Malleable Tasks. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 167–178. Springer, 2012.

[118] Sandra Sattolo. An Algorithm to Generate a Random Cyclic Permutation. *Information Processing Letters*, 22(6):315–317, 1986.

[119] Jochen Seidel. Job-Scheduling in Main-Memory Based Parallel Database Systems. Diplomarbeit, KIT, 2011.

[120] Johannes Singler. *Algorithm Libraries for Multi-Core Processors.* PhD thesis, KIT, 2010.

[121] Johannes Singler, Peter Sanders, and Felix Putze. MCSTL: The Multi-core Standard Template Library. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 682–694. Springer, 2007.

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

[123] Ian Sommerville. *Software Engineering.* Pearson Education, 2007.

[124] Hans Stadtherr. *Work Efficient Parallel Scheduling Algorithms.* PhD thesis, Technische Universität München, 1998.

[125] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2009.

[126] Sivaprakasam Sunder and Xin He. Scheduling Interval Ordered Tasks in Parallel. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 100–109. Springer, 1993.

[127] David Tam, Reza Azimi, and Michael Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *European Conference on Computer Systems (EuroSys)*, pages 47–58. ACM, 2007.

[128] Andrew S. Tanenbaum and Todd Austin. *Rechnerarchitektur : von der digitalen Logik zum Parallelrechner*. Pearson, 2014.

[129] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson Education, 2015.

[130] Gecode Team. Gecode: Generic Constraint Development Environment, 2011. Available from *http://www.gecode.org*.

[131] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. Invasive Computing: An Overview. In *Multiprocessor System-on-Chip*, pages 241–268. Springer, 2011.

[132] Jürgen Teich, Jürgen Kleinöder, and Katja Lohmann, editors. *Invasive Computing Annual Report 2011*. DFG Transregional Collaborative Research Centre 89. https://invasic.informatik.uni-erlangen.de/publications/annual_report_2011.pdf.

[133] Jürgen Teich, Jürgen Kleinöder, and Katja Lohmann, editors. *Invasive Computing Annual Report 2012*. DFG Transregional Collaborative Research Centre 89. https://invasic.informatik.uni-erlangen.de/publications/annual_report_2012.pdf.

[134] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997.

[135] John Turek, Joel L. Wolf, and Philip S. Yu. Approximate Algorithms for Scheduling Parallelizable Tasks. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 323–332. ACM, 1992.

[136] Jeffrey D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[137] Martin L. Van Creveld. *Command in war*. Harvard University Press, 1985.

[138] Ward Van Heddeghem, Sofie Lambert, Bart Lannoo, Didier Colle, Mario Pickavet, and Piet Demeester. Trends in worldwide ICT electricity consumption from 2007 to 2012. *Computer Communications*, 50:64–76, 2014.

[139] David W. Wall. Limits of Instruction-Level Parallelism. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 176–188. ACM, 1991.

[140] Wolfgang Walter. *Analysis 1*. Springer, 2004.

[141] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[142] James M. Wilson. Gantt charts: A centenary appreciation. *European Journal of Operational Research*, 149(2):430–437, 2003.

[143] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[144] Jan Węglarz. Project Scheduling with Continuously-Divisible, Doubly Constrained Resources. *Management Science*, 27(9):1040–1053, 1981.

[145] Fatos Xhafa and Ajith Abraham, editors. *Metaheuristics for Scheduling in Distributed Computing Environments*. Springer, 2008.

[146] Fatos Xhafa and Ajith Abraham, editors. *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*. Springer, 2008.

[147] Huiting Xu, Fanxin Kong, and Qingxu Deng. Energy Minimizing for Parallel Real-Time Tasks Based on Level-Packing. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 98–103. IEEE, 2012.

[148] Frances Yao, Alan Demers, and Scott Shenker. A Scheduling Model for Reduced CPU Energy. In *Symposium on Foundations of Computer Science (FOCS)*, pages 374–382. IEEE, 1995.

[149] Asim Yarkhan, Jakub Kurzak, and Jack Dongarra. QUARK Users' Guide. *Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tennessee*, 2011.

[150] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013.

[151] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX, 2010.

[152] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 129–142. ACM, 2010.

[153] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *ACM Computing Surveys*, 45(1):4:1–4:28, 2012.

Deutsche Zusammenfassung

Lastverteilung/Scheduling in der Informatik bezeichnet das Finden eines Plans (Schedule) zur effizienten Ausführung eines Programms (wenn die Ausführung interner Teilaufgaben geplant wird) oder einer Gruppe von Programmen auf einem Computersystem. Dies beinhaltet das Zuweisen von benötigten Ressourcen zum Beispiel von Kernen oder Speicher/Cache genauso wie die Bestimmung anderer Parameter wie das Setzen der Taktrate. Natürlich muss der gefundene Plan umsetzbar und zulässig sein. Gute Pläne/Schedules sind wichtig für einen effizienten Betrieb des Computersystems im Hinblick auf Ziele wie hohen Durchsatz, geringen Energieverbrauch oder schnelle Fertigstellung aller Aufgaben. Genauso ist es wichtig, dass die Berechnung des Schedules nicht zu viel Zeit oder andere Ressourcen benötigt.

Heutige Computer sind üblicherweise Parallelrechner mit mehreren parallel arbeitenden Kernen. Auch werden die verwendeten Anwendungen (insbesondere jene mit hohem Rechenbedarf) heute immer öfter parallelisiert. Deshalb wird das Scheduling paralleler Anwendungen zu einem wichtigen Faktor der Effizienz und Leistungsfähigkeit heutiger Computersysteme.

Scheduling ist ein großes Gebiet, das in vielen verschiedenen Zusammenhängen und vielen verschiedenen Ansätzen untersucht wird. Zum Beispiel enthält das "Handbook of Scheduling"(zusammengestellt von Joseph Y-T. Leung) 1224 Seiten und zitiert etwa 2000 andere wissenschaftliche Arbeiten aus diesem Gebiet. Sogar das Scheduling von parallelen Anwendungen ist ein umfangreiches Gebiet. Trotz des hohen Forschungseinsatzes gibt es eine große Lücke zwischen der Forschung im Bereich der Schedulingtheorie und den in der Praxis oder in experimentellen Arbeiten verwendeten Lösungen.

Unser Ansatz ist es, die Lösungen aus Theorie und Praxis zu studieren, und das dabei gewonnene Wissen zu nutzen, um Resultate zu produzieren, die die Lücke verkleinern. Ein Grund für die Lücke zwischen Theorie und Praxis ist, dass die Modelle der Theorie mehr Wissen über die Anwendungen annehmen als in der

Praxis zur Verfügung steht. Zudem nehmen einige Modelle der Schedulingtheorie noch an, dass der System-Scheduler eine Möglichkeit hat, das Verhalten der Anwendungen zu ändern, welche in der Praxis noch nicht existiert. Aber es gibt Ansätze, die Anwendungen anpassbar an den Systemzustand zu machen und mehr Informationen für das Betriebssystem bereitzustellen. Ein Beispiel für die Entwicklungen mit dem Ziel, den Informationsaustausch zwischen Anwendungen und Betriebssystem und die Anpassbarkeit der Anwendungen zu verbessern, ist das InvasIC-Projekt. Der Autor dieser Arbeit war vier Jahre für das Projekt tätig und arbeitete in dieser Zeit mit Experten für viele verschiedene Bereiche moderner Computersysteme zusammen. Der verbesserte Informationsaustausch und die Anpassbarkeit von Anwendungen sind Systemdesign-Entscheidungen, die das System so verändern, dass die Ergebnisse der Schedulingtheorie besser anwendbar sind. Insbesondere das Modell der malleablen Jobs, welche ihre Ressourcennutzung während der Laufzeit ändern können, scheint ein passendes Gegenstück zu den InvasIC-Entwicklungen auf der Theorieseite zu sein.

Während unserer Arbeit am InvasIC-Projekt war es uns möglich vier Hauptforschungsrichtungen für das Scheduling von Parallelrechnern zu identifizieren (jede dieser Richtungen wird in einem eigenen Kapitel behandelt):

- *Die Verteilung der Informationen und Entscheidungen zwischen den verschiedenen Entscheidungsträgern im System.* In modernen Systemen gibt es verschiedene Entscheidungsträger wie zum Beispiel den Betriebssystemscheduler oder die anwendungsinternen Scheduler. Um gute Resultate zu erzielen, müssen diese Scheduler koordiniert werden.
- *Schnelles und effizientes Finden von guten Entscheidungen (das klassische Problem der Forschung im Bereich der Schedulingtheorie).* Die effiziente Lösung der oftmals komplizierten Schedulingprobleme bleibt der Kern jedes Schedulingsystems. Eine umfassende Basis dafür kann in der Schedulingtheorie gefunden werden.
- *Die effiziente Nutzung von Speicher und Caches.* Die effiziente Nutzung des Speichersystems ist eine zentrale Anforderung, um hohe Leistung und Effizienz zu erreichen.
- *Die Verringerung von Leistungsbedarf und Energieverbrauch.* Energieverbrauch und Hitzeentwicklung blockieren einfache Leistungsgewinne durch die Steigerung der Taktrate. Zudem wird dieser Bereich auch durch die Zunahme batteriebetriebener Geräte immer wichtiger.

Diese Arbeit stellt Resultate aus allen vier Bereichen dar. Das Hauptergebnis dieser Arbeit ist ein schneller Scheduling-Algorithmus für malleable Jobs, der optimale Schedules berechnet, falls das gegebene Problem einige Bedingungen erfüllt. Die Zielfunktionen, die durch unseren Hauptalgorithmus minimiert werden, sind entweder Maxima oder Summen von Jobeigenschaften. Wir zeigen auch, dass unser Hauptalgorithmus parallelisiert werden kann und dass die parallele Version einen optimalen Schedule in polylogarithmischer Zeit findet, falls die Anzahl verwendeter Kerne mindestens so groß ist wie die Anzahl der Jobs. Das macht diesen Algorithmus zum ersten parallelen Schedulingalgorithmus für parallele Jobs (soweit wir wissen). Einige Anwendungen des Hauptalgorithmus werden dargestellt, darunter auch die Minimierung des Energieverbrauchs einer Menge von malleablen Jobs. Soweit wir wissen ist die Anwendung auf die Energieminimierung der erste Algorithmus, der das Energieminimierungsproblem für malleable Jobs optimal löst. Neben dem Hauptergebnis beschreiben wir auch eine Heuristik für das schnelle und effiziente Scheduling, welche in einem Wettbewerb erfolgreich war.

Bezüglich der Koordination und des Informationsaustauschs zwischen verschiedenen Schedulern enthält diese Arbeit einige generelle Überlegungen aus anderen Bereichen und ein Beispiel für eine malleable Anwendung und dem entsprechenden Interface.

Das Speicher- und Cache-Verhalten moderner Computersysteme wird durch umfangreiche Experimente in dieser Arbeit untersucht. Zwei Fallstudien von Speicheroptimierungen für reale Anwendungen wurden mit unserer Beteiligung durchgeführt. Ihre Ergebnisse werden nach den Basisexperimenten zum Cache- und Speicherverhalten beschrieben. Wir betrachten auch den Leistungsbedarf von Speicheroperationen.