Lehrstuhl für Effiziente Algorithmen
der Technischen Universität München

Efficient Algorithms for On-Line Scheduling and Load Distribution in Parallel Systems

Stefan Bischof

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr.-Ing. E. Jessen

Prüfer der Dissertation:

1. Univ.-Prof. Dr. rer. nat. E. W. Mayr
2. Univ.-Prof. Dr. rer. nat. Chr. Zenger
3. Außerordentlicher Univ.-Prof.
   Dr. techn. G. J. Wöginger,
   Technische Universität Graz / Österreich
   (schriftliche Beurteilung)

Die Dissertation wurde am 21. 1. 1999 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 9. 9. 1999 angenommen.

Preface

In the day of internet hype everything needs a home page, and so has this thesis: `http://www.bischof-web.de`. If you have any comments or questions, please direct them to `stefan@bischof-web.de`

The hardcopy version (also called "book") of this thesis is available at your favorite book store or from "Libri Books on Demand" (ISBN: 3-89811-444-9) at a moderate price.

I would like to thank the B. G. Teubner Verlagsgesellschaft for the kind permission to reprint parts of the article "Classification and Survey of Strategies" [BE97] in Chapter 3.

Abstract

The efficient operation of parallel computing systems requires the best possible use of the resources that a system provides. In order to achieve an effective utilization of a parallel machine a smart coordination of the resource demands of all currently operating applications is necessary. Dynamic resource management is particularly essential for the parallel solution of irregular problems that arise frequently, for example, during numerical simulations. On-line scheduling and load distribution are widely used techniques for the assignment of resources in a dynamic fashion. This thesis provides a theoretical treatment of both approaches and presents efficient on-line scheduling and load distribution algorithms for a wide range of problems.

On-line scheduling of parallel job systems with the goal to minimize the total execution time is studied as a promising framework for the efficient execution of large-scale parallel applications. Under the assumption that the execution times of the jobs meet certain requirements, several efficient on-line scheduling algorithms for various network topologies are described. Thorough competitive analysis shows that almost all of the proposed on-line scheduling algorithms are optimal or near-optimal from a worst-case point of view. It is demonstrated that runtime restrictions improve the competitive performance achievable by on-line scheduling algorithms for parallel job systems, and therefore a satisfactory utilization of a parallel system can be guaranteed in this case.

Dynamic load distribution is used for partitioning problems for parallel execution. The only assumption is that a given class of problems has a certain bisection property. Such classes of problems appear, for example, in the context of distributed hierarchical finite element simulations. The sequential and parallel algorithms presented to tackle this load distribution problem yield provably good load balancing even in the worst case. Furthermore, under reasonable stochastic assumptions, it is shown that the average-case performance of these algorithms is sub-

stantially better than the worst-case bounds and surprisingly close to the optimum solution in some cases. Extensive simulation studies complement the mathematical analysis, and the results demonstrate that a satisfactory balancing quality can be achieved in this model by efficient algorithms. An integration of the sequential algorithm into existing finite element software already yielded a significant decrease of the total execution time.

Acknowledgments

This thesis would have never been written without the help and assistance of many people.

First of all, I would like to thank my advisor Ernst W. Mayr for his steady support and guidance, for introducing me to scheduling, load balancing, and the analysis of algorithms, for giving me indispensable motivation and the right ideas at the right time, and for his gorgeous treasure of bibliographic references.

I'm grateful to all my present and former colleagues at the Chair for Efficient Algorithms. The friendliness and helpfulness that I encounter each single day makes working here a real pleasure for me.

Furthermore, I thank my coauthors of several papers on load balancing, Angelika Steger, Ralf Ebner, Thomas Erlebach, and Thomas Schickinger for all the helpful discussions and inspiring ideas, for carefully reading the papers and most parts of this thesis, for their outstanding and successful cooperation, and for the large amount of their work they contribute to this thesis.

Valuable discussions with several other researchers improved this work considerably. In particular, I would like to thank Klaus Jansen, Gerhard Woeginger, Amos Fiat, Jiří Sgall, and Shang-Hua Teng for their stimulating ideas.

I would like to express my thankfulness to my parents for many things they taught me, for their help and understanding.

I'm grateful to my family for all their patience, support, and loving. Purring on my lap, my two cats Hägar and Attila were always willing to give me a break. My beloved children Daniela and Samuel make the nights shorter but my days much brighter. But most of all, I thank my marvelous wife Susanna for her encouragement and never-ending love.

Deut. 6 : 4–5 Matt. 22 : 37–39

To my wife Susanna

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The solution of large computational problems requires efficient algorithms as well as powerful computing and communication hardware. Challenging and important problems, whose solution involves complex calculations and huge amounts of data, originate from many different areas such as, for example, numerical simulation of physical processes, computational biology and chemistry, data mining in large databases, automotive and aerospace design, or financial analysis.

Despite the impressive advances in the performance of microprocessors, memory chips, and other hardware components, the solution of many large-scale computational problems in a reasonable amount of time requires the use of a parallel system that connects a large number of the most powerful processors via a high speed network. The aggregated processing speed, memory size, and network throughput of state of the art parallel computers is already enormous [DMS98], and will increase even more over the next years [MCP+98]. However, the peak performance of a supercomputer, which is achieved solving very regular problems such as dense systems of linear equations, is far from being reached when problems have to be solved that have fewer inherent parallelism or show an unpredictable dynamic behavior during execution.

Ideally, an efficient parallel algorithm would keep all computational nodes of a parallel system busy in order to minimize the overall execution time that is used to solve a particular problem. In many cases, however, it is difficult and time-consuming to partition a given problem into subproblems of almost equal size that can subsequently be processed in parallel. Consider, for example, the decomposition of an irregularly structured mesh of a finite element simulation.

A fact that complicates this situation even more is that for many parallel applications, e.g., adaptive numerical simulations, the number and workload of individual subproblems cannot be predicted with sufficient precision at compile-time or when the application starts computation. Moreover, in a large parallel system jobs might arrive without a full specification of their resource demands such as execution time or memory. Efficient and effective resource management is therefore a major issue that has be addressed in order to exploit the large potential of parallel systems as far as possible.

The above discussion makes it clear that a resource management facility is often forced to make decisions in a state of uncertainty since it has only partial knowledge about the future. The amount of knowledge might be even smaller when complete information about the current state of the parallel system is not available. It is a challenging task to devise efficient algorithms that cope with the handicap of incomplete knowledge and achieve solutions that are best possible under such circumstances.

Competitive analysis of on-line algorithms [BE98, FW98] is a frequently used framework to deal with incomplete information in a formalized manner from a worst-case point of view. An on-line algorithm learns it input piece by piece and has to generate valid partial solutions based on the knowledge of the past and the presence, but without secure information about the future. The goal is to optimize a certain objective function. Competitive analysis means that the quality of an on-line algorithm is measured by comparing its performance to that of an optimal algorithm that knows the whole input in advance. More precisely, an on-line algorithm ALG for a minimization problem is c-competitive for a $c \geq 1$ if for all inputs the value of the solution produced by ALG exceeds the value of an optimum solution by at most a factor of c. The competitive ratio of an on-line algorithm is the infimum over all c for which it is c-competitive (see also Section 4.1 on page 39). Therefore, competitive analysis is a worst-case measure that aims at comparing different on-line algorithms for the same problem.

Approximation algorithms and on-line algorithms are closely related. Both try to compute a "good" approximation for an optimization problem. However, the approximation algorithm has limited computational resources (e.g., only time polynomial in the length of the input may be permitted), whereas an on-line algorithm has limited knowledge of the input during the computation. Therefore, efficiency is an important additional characteristic of on-line algorithms in order to be practically

useful. But in many settings it turns out that algorithms that achieve the best competitive ratio are also computationally very simple and thus deserve the predicate *real time*.

The analysis of on-line algorithms frequently uses adversary arguments. In order to maximize the competitive ratio, the adversary constructs an input that is unfavorable for the on-line algorithm but comes in handy to an optimal off-line algorithm. An oblivious adversary has to fix the input in advance whereas an adaptive adversary may react to the decisions of the on-line algorithm. This distinction is often crucial for randomized on-line algorithms that try to improve their competitiveness by the use of random bits. However, the two adversary types are equally strong for deterministic on-line algorithms.

On-line scheduling is a popular and widely studied technique to tackle dynamic resource management problems where full knowledge about the input instance cannot be assumed. In general terms, scheduling is concerned with the problem of allocating resources over time to sets of jobs such that no given allocation restriction is violated in order to optimize some objective function [BEP+96].

Competitive analysis allows for a thorough mathematical analysis and a rating of different on-line scheduling algorithms (or on-line schedulers for short) from a worst-case point of view. However, it is assumed in this model that an on-line scheduler is a sequential algorithm that has a full record of past events and complete information about its environment. In the context of dynamic resource management, these assumptions might be too restrictive. A sequential on-line scheduling algorithm may be a bottleneck that might lead to an unpredictable performance decrease in certain situations.

Therefore, load distribution is also frequently used for resource management with incomplete information, although often in a less formal way. Load distribution can be defined [Sch97b] as the assignment of entities (objects to be processed) to targets (distribution units that provide resources) in order to meet given requirements. In addition, a load distribution algorithm may be able to subdivide certain entities into several smaller entities in order to distribute the workload among the targets. The decision process of a (dynamic) load distribution strategy uses load-state information about the parallel system accumulated at run-time (the term dynamic is often added to emphasize this fact, but we omit it for brevity) in order to find, for example, a "good" execution location for a new job, or to transfer some of the workload of an overloaded to an underloaded processor.

To provide scalability, these assignment decisions can be reached concurrently by a parallel or distributed algorithm.

We will study on-line scheduling and load distribution problems from a theoretical point of view. The main focus will be on the design and thorough worst-case or average-case analysis of efficient algorithms for these problems. Although this requires some simplifying assumptions regarding parallel systems and applications, our models certainly reflect the decisive characteristics of many resource management problems that are of practical relevance. Due to the generality of our approach, the proposed algorithms are widely applicable and provide provably good performance.

1.1 Synopsis of Previous and Related Work

Scheduling and load distribution problems have been studied intensively for decades. This fruitful and ongoing research is documented in a vast amount of scientific articles and numerous books. It is therefore beyond the scope of this thesis to give a complete account of the work that has been done in this field up to now. However, throughout this thesis, relevant previous and related work is surveyed and many pointers to the literature are provided.

In particular, Chapter 2 reviews the on-line machine load balancing model and a general framework for on-line resource management. In Section 3.2, selected load distribution strategies are described in order to reveal basic algorithmic techniques that have been developed to tackle the load distribution problem. Furthermore, Section 4.2 reviews results on non-preemptive on-line scheduling of parallel jobs and also gives some results regarding the computational complexity of the corresponding off-line problems.

In addition, a thorough introduction to scheduling theory and a good overview of results regarding multiprocessor scheduling as well as other important scheduling models can be found in [BEP+96]. SGALL [Sga98] provides an excellent and up-to-date survey of algorithms and results for on-line scheduling problems, and a taxonomy of on-line scheduling problems is given that comprises (among other criteria) a general problem definition, several different on-line paradigms (see also Chapter 2), possible objective functions, important job characteristics, and frequently used machine models.

1.2 Thesis Outline

In Chapter 2, *On-Line Scheduling and Load Distribution: A Comparison*, a comparative study of on-line scheduling and load distribution is made in order to expose the major differences of these techniques from our point of view. Then, Chapter 3, *Classification and Survey of Load Distribution Strategies*, presents a comprehensive classification scheme for load distribution strategies. From the vast body of literature on this topic a survey of selected load distribution strategies is given to exemplify basic algorithmic methods that were developed to tackle the load distribution problem.

On-line algorithms for scheduling parallel job systems are studied in Chapter 4, *On-Line Scheduling of Parallel Jobs with Runtime Restrictions*, under the assumption that the execution times of the jobs meet certain requirements. For parallel job systems with unit execution time it is shown that for an arbitrary interconnection topology of the parallel system no deterministic or randomized on-line scheduling algorithm can achieve a competitive ratio better than 2. It is also shown that this general lower bound can be improved for particular network topologies. A generic on-line scheduler for the unit execution time model is presented that achieves optimal or near optimal (up to small additive constants) competitive ratio for several network topologies. To investigate the entire bandwidth between unit and arbitrary running times of the jobs, the parallel job systems are classified according to their runtime ratio, i.e., the ratio between the longest and shortest execution time of any job. Again, for an arbitrary network topology, a general lower bound on the competitive ratio of any deterministic or randomized on-line scheduler is derived that depends only on the maximum runtime ratio of a class of parallel job systems. For parallel systems that support arbitrary allocation of processors to parallel jobs, an on-line scheduler with near optimal (up to a small additive constant) competitive ratio is proposed.

In Chapter 5, *Load Balancing for Problems with Good Bisectors*, a general load balancing approach using bisectors is described. A class of problems has α-bisectors if every problem in the class can be subdivided into two subproblems whose weight is not smaller than an α-fraction of the original problem for a fixed $\alpha > 0$. It is shown that the existence of α-bisectors for a class of problems allows good load balancing for a surprisingly large range of values of α. Algorithm HF, an efficient load balancing algorithm for this model, is proposed, and a tight worst-case upper bound on the ratio between the maximum load generated by this algorithm and the ideal

load (uniform partition) is derived that depends only on α. This bound implies performance guarantee 2 when $\alpha \geq 1/3$ and performance guarantee 3 when $\alpha \geq 1 - 1/\sqrt[4]{2}$ for this algorithm (an algorithm has performance guarantee ρ if the maximum load produced by the algorithm is at most a factor of ρ larger than the maximum load of an optimum solution). Two strategies to use Algorithm HF for load balancing distributed hierarchical finite element simulations and experimental results are presented. For this purpose, a certain class of weighted binary trees representing the load of such applications is shown to have 1/4-bisectors.

Based on the results on this sequential load balancing algorithm, parallel algorithms for this load balancing model are studied in Chapter 6, *Parallel Load Balancing for Problems with Good Bisectors*, in order to reduce the balancing overhead. First, a parallel implementation of Algorithm HF is derived that produces the same load distribution as the sequential algorithm at the expense of a substantial communication overhead. Therefore, a simpler and faster parallel load balancing algorithm, Algorithm BA, is introduced. The bound on the worst-case load imbalance for this algorithm, however, is shown to be worse than the corresponding bound for Algorithm HF. Algorithm BA-HF, an integration of Algorithm BA and Algorithm HF (or its parallelization), is shown to combine the main advantages of both approaches. The results of extensive simulation experiments regarding the load imbalance incurred by the three proposed parallel algorithms in the average case are reported.

The average-case balancing quality of Algorithm HF is analyzed in Chapter 7, *Average-Case Analysis of Load Balancing using Bisectors*. Assuming a natural and rather pessimistic distribution for the average case, it is shown that the maximum load generated by Algorithm HF does not exceed the ideal load (uniform partition) by a factor of $2 + \varepsilon$ with high probability. Moreover, ε is close to zero already for moderate numbers of processors. From this analysis faster and simpler variants of Algorithm HF and its parallelization are derived.

Finally, Chapter 8, *Conclusion*, summarizes and discusses the results derived and presented in Chapters 2–7. A number of open problems arising from these results are described, and possible directions for future work are given.

Many of the results described in this thesis have been obtained in joint work with Angelika Steger, Ralf Ebner, Thomas Erlebach, Ernst W. Mayr, and Thomas Schickinger. Preliminary versions of some of the results contained in Chapters 3–7 were accepted for presentation and publication

at various conferences, or were otherwise published. In particular, the classification scheme for load distribution strategies and a brief survey of selected strategies appears in [BE97]. Some of the results for on-line scheduling parallel jobs with runtime restrictions were presented at the International Symposium on Algorithms and Computation (ISAAC'98) [BM98a] and the Workshop on On-Line Algorithms in Udine (OLA'98). The article is accepted for a special issue on on-line algorithms of Theoretical Computer Science that will be published on the occasion of OLA'98. Preliminary results on this topic can also be found in [BM98b]. The results obtained for load balancing for problems with good bisectors from Chapter 5 were announced in part at the the Workshop on "Anwendungsbezogene Lastverteilung[1]" (ALV'98) [BEE98a] and the EURO-PAR Conference on Parallel Processing (EURO-PAR'98) [BEE98b]. A preliminary version of these results is contained in [BEE98c]. The parallelization of this approach given in Chapter 6 was be presented at the International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP'99) [BEE99]. Finally, the results regarding the average-case analysis of load balancing using bisectors were announced at the European Symposium on Algorithms (ESA'99) [BSS99].

[1] Application oriented load distribution

Chapter 2

On-Line Scheduling and Load Distribution: A Comparison

Solving irregular problems efficiently in parallel or sharing a parallel computer effectively among a set of users are challenging and important tasks. Not surprisingly, these and related problems have triggered intensive research, both theoretical and applied, that led to a variety of different approaches to tackle such tasks. Among these, on-line scheduling and load distribution have received considerable attention in the scientific literature. In this chapter we give a comparison of these two widely used techniques for dynamic resource management in parallel systems, and briefly review two related approaches.

The vast amount of articles published on on-line scheduling and load distribution (often using differing problem formulations and assumptions, as well as inconsistent and sometimes contradictory terminology) makes it difficult to rate the relative merits of alternative strategies. This comparison aims at exposing the main differences between on-line scheduling and load distribution from our point of view. In order to do so, we keep our discussion as informal as possible. A more detailed description of the properties of load distribution strategies is given in the next chapter.

The task of a scheduling algorithm for parallel systems, no matter whether off-line or on-line, is to compute a *schedule*, i.e., an assignment of processors (and possibly other resources) to jobs over time such that

1. at every moment in time each processor is assigned to at most one job,

2. each job is executed and eventually completed,
3. all constraints imposed by the particular scheduling model are obeyed.

The goal is to optimize or approximate a given objective function, for example the total schedule length. This corresponds directly to the efficiency requirements stated above. The performance of an on-line scheduler is measured by the competitive ratio, i.e., relative to an optimum solution with respect to the given objective function. This allows for a comparison of different on-line schedulers from a worst-case point of view.

There are several possibilities for the knowledge and abilities of an on-line scheduling algorithm (cf. [Sga98]). We describe two of them to illustrate the general practice in on-line scheduling:

- Scheduling jobs one by one. This variant is analogous to other "classical" on-line problems like list-update or paging. The jobs are ordered in a list and input one by one to the on-line scheduler in this order. Each arriving job has to be assigned consistently with previous decisions to a processor (or the requested number of processors in the case of parallel jobs), and must receive a start-time before the next job is presented. The assignment of a job is irrevocable, but all job characteristics are revealed as soon as a job is passed to the on-line scheduler.
- Unknown job characteristics. Often, a job can be scheduled without full knowledge of its characteristics. For example, the running time of a job may be unknown until the job finishes execution, or the number of successors might be unknown in the case of dependencies. The on-line scheduler is aware of all currently available jobs and may choose any of them to be scheduled next. Job assignments may be modified using preemptions or restarts if these activities are allowed.

We note that it is possible in both cases to *delay* the execution of a job for an arbitrary time-span.

Load distribution problems are often stated less formally. To capture the generality of the problem, the main purpose of load distribution can be defined as the assignment of *entities* (e.g., jobs) to *targets* (e.g., processors) in order to meet given requirements [Sch97b]. Furthermore, some

load distribution methods may provide the additional functionality of generating the entities dynamically by partitioning larger problems into subproblems. A load distribution strategy is potentially aware of all jobs currently in the system, but the knowledge about the job characteristics can vary from complete to none depending on the particular model.

There are two important subclasses of load distribution:

- *Load balancing* tries to keep the deviation of the current system load from the best possible distribution of that load (with respect to a given metric) as small as possible.
- *Load sharing* is less ambitious and the goal here is to keep all processors busy whenever possible.

The available means to reach such goals include the initial placement of newly created or arriving jobs, the reassignment of existing jobs that have not yet started execution, and the preemptive migration of running jobs.

A major difference between on-line scheduling and load distribution is that at any moment in time a load distribution algorithm may assign more than one job to a single processor for execution without specifying the order or mode in which these jobs have to be processed. Rather, this decision is frequently left to a local scheduling algorithm on each processor. According to the well-known classification scheme of CASAVANT and KUHL [CK88] load distribution as defined above would be classified as global dynamic scheduling since it is only concerned with the decision *where* to assign (and potentially execute) a job. However, we prefer to reserve the notion *scheduling* to those algorithms that explicitly generate a schedule.

A further distinction arises from the often more complex decision structures of load distribution strategies. On-line schedulers are usually sequential algorithms that act as a centralized control unit of the parallel system. In contrast, many load distribution strategies have a distributed or hierarchical decision structure. This means that several components may decide about load transfers independently or coordinated in some way (see Section 3.1.4 for a detailed discussion). The scalability of such approaches is one the most appealing features of load distribution. Although there is already some work about fault-tolerant distributed on-line algorithms [Asp98], it remains an important open problem to develop meaningful on-line paradigms and an appropriate definition of competitive performance for parallel or distributed on-line scheduling algorithms.

The competitive analysis of on-line scheduling algorithms gives worst-case upper bounds on the performance of these algorithms that are directly related to the intended use of these scheduling algorithms. For example, we might have the result that the length of an on-line schedule is never more than twice the length of an optimum schedule. Load distribution, however, is often applied as a heuristic to decrease, e.g., the running time of a parallel application, or to improve other performance measures. This makes the theoretical analysis of such approaches difficult or virtually impossible. Furthermore, if we resort to more restricted load distribution models that are amenable to thorough mathematical analysis (e.g., [XL97]), the results are usually limited to the load distribution strategy itself. The implications of such results for the running time of a parallel application, for example, are not immediately clear, and are often determined by experimental results only.

An interesting hybrid of the on-line paradigm and load distribution is the on-line machine load balancing model (a good survey by AZAR can be found in [Aza98]). It incorporates the possibility to assign multiple jobs to a single machine at any instant in time. The jobs have to be assigned one by one without delay. The assignment of a job increases the load of the target machine by the amount that is specified by the corresponding coordinate of its vector of load values for the duration of that job. The duration of a job may be known or unknown to the on-line algorithm. Possible goals are to minimize the maximum load of any machine or to minimize the average delay of any job. The precise definition of this resource management model allows the competitive analysis of a variety of on-line problems including, e.g., the assignment of bandwidth to communication requests in networks, or the assignment of queries to database servers.

A general framework for on-line resource management problems was proposed by LEONARDI and MARCHETTI-SPACCAMELA [LMS95, Leo96]. It is assumed that a sequence of jobs arrives one by one, each of which can be scheduled using an individual set of alternatives. Each alternative describes a possible combination of various kinds of resources that are sufficient to fulfill the demands of that job. Two different classes of on-line resource management problems can be distinguished with respect to the objective function:

- Benefit problems. The goal is to maximize the total benefit collected from the subsequence of jobs that can be scheduled without exhausting the available resources.

- Cost problems. The goal here is to minimize the amount of resources that is used in order to process all jobs presented to the on-line algorithm.

Benefit problems are modeled as linear programs which are solved in an on-line fashion, whereas methods similar to the on-line machine load balancing problem are used for cost problems. This framework was applied successfully to a variety of on-line scheduling and routing problems.

Chapter 3

Classification and Survey of Load Distribution Strategies

In this chapter we first present a comprehensive classification scheme for load distribution strategies. It is aimed at providing a better orientation in the confusing diversity of approaches that were proposed up to now. Furthermore, the scheme may serve as a guideline for future research since it states many important aspects of load distribution problems in a clear and understandable way. In Section 3.2 we give a brief survey of selected well-known and recently developed load distribution strategies.

3.1 Classification of Load Distribution Strategies

Classification schemes are helpful tools in making the tasks of comparison and evaluation easier. Several such schemes for the purpose of classifying load distribution have been designed [LMR91, SKS92, Lud93, RR96, DMP97, XL97] but so far none of them has been generally accepted. Furthermore, these schemes differ much in the extent to which technical details have been included. Therefore, in a joint effort, a team of more than a dozen researchers (from two research projects at Technische Universität München and an industrial partner) has developed several clear-cut and well-defined classification schemes for different, mainly orthogonal aspects of load distribution for parallel applications [SS97].

In this book [SS97], a general classification scheme regarding objectives, integration level, structure, and implementation of load distribution

is given [Sch97b]. In addition to the classification scheme for load distribution strategies presented in this chapter, a classification scheme for *load models* [Röd97] and *migration mechanisms* [Ste97] was proposed. A load model describes the properties and interpretation of the load index that is used to represent the "system load". Moreover, it comprises policies regarding the measurement and propagation of the load index and possibly the adaptivity of those policies. Migrations mechanisms are the tools that are necessary to realize the assignment decisions of the load distribution strategy. Generally, this includes any transfer of an entity to a different target. But it should be mentioned that the term *migration* is often used in the literature to describe the preemptive transfer of processes or threads. SCHNEKENBURGER [Sch97a] applied these classification schemes to a variety of load distribution approaches.

Over the past 30 years, numerous strategies for load distribution have been proposed by researchers working in a large number of different areas of computer science. The individual strategies have sometimes been evaluated by thorough mathematical analysis, sometimes by simulation, and sometimes by appealing to the intuition of the reader. As a consequence, it is extremely difficult to compare different load distribution strategies and to determine which strategy will perform best in a particular scenario.

A fact that complicates this task even further is that most load distribution strategies proposed in the literature rely on very specific assumptions about the system and the application at hand. Unfortunately, such assumptions are often only implicit in the presentation of the strategy, and must be made explicit in order to determine whether a strategy is appropriate for a particular load distribution problem.

Our classification scheme for load distribution strategies is intended to make comparison and evaluation of different strategies easier. We note that the technical details of a particular implementation do not play a major role for the presentation and understanding of a strategy. Usually, a strategy need not specify for which particular kind of entities and targets it was designed. The same strategy could, in principle, be used to assign different kinds of entities (e.g., processes, threads, or database queries) to different kinds of targets (e.g., workstations, multiprocessor nodes, or data base servers). In addition, a strategy is not concerned with measuring load or calculating load indices on the individual targets or with the technical mechanisms used for assigning entities. Furthermore, the local scheduling policy used for processing several entities on a single target is not considered part of the load distribution strategy. The classification

scheme is intended to extract only the basic underlying strategy of an approach to a load distribution problem.

Of course, we do not want to imply that somebody looking for an adequate load distribution strategy should evaluate different strategies independently from a particular problem. Instead, it is essential to choose a strategy that matches the involved system and application requirements well. Therefore, the classification scheme was designed to include those aspects of a strategy that are necessary in order to estimate the expected performance on a particular system with certain properties.

The strategy classification scheme we propose is hierarchical and includes criteria pertaining to system model, transfer model, information exchange, coordination, and algorithm. The scheme is sketched in Table 3.1. A detailed description follows.

3.1.1 System Model

The underlying system model is one of the strongest classification criteria for load distribution strategies. If the system at hand does not match the system for which the strategy was designed, it is highly questionable whether the strategy can be of any use for that system. The relevant components of a system as seen by a load distribution strategy are the entities that represent the load and the targets to which the entities can be assigned. In addition, it is in general quite helpful to know where the motivation for the design of a particular strategy came from. Therefore, we include the following criteria:

- **Model Flavor:**
  For several load distribution strategies it is obvious where the motivation for the selection of that particular strategy originated. Many strategies are based on the analogy with *physical* systems, e.g., the diffusion algorithm [Cyb89] or the gradient model algorithm [LK87]. Such strategies try to imitate physical systems where a kind of load distribution is accomplished as a consequence of the laws of nature.

  Other strategies originate from *combinatorial* models (e.g., see [GM94, GM96] and Section 3.2.3). Here, the load distribution problem is usually formalized as a discrete mathematical problem and tackled by employing results from graph theory, scheduling theory or related fields.

Table 3.1: Classification scheme for load distribution strategies

System Model	
Model Flavor	physical, combinatorial, microeconomic, random, fairness, none, ...
Target Topology	(heterogeneous) NOW, bus, mesh, fully connected, ...
Entity Topology	grid-like, tree-like, non-interacting entities, ...
Transfer Model	
Transfer Space	systemwide, long range, short range, neighborhood
Transfer Policy	preemptive, non-preemptive
Information Exchange	
Information Space	systemwide, long range, short range, neighborhood, central
Information Scope	partial, complete, none
Coordination	
Decision Structure	distributed, hierarchical, centralized
Decision Mode	autonomous, cooperative, competitive
Participation	global, partial
Algorithm	
Decision Process	static, dynamic
Initiation	sender, receiver, sender & receiver, central, timer-based, threshold-based
Adaptivity	fixed, adjustable, learning
Cost Sensitivity	none, low, partial, high
Stability Control	none, not required, partial, guaranteed

Economical systems have also inspired researchers to design analogous load distribution strategies [BP97]. This model is usually referred to as the *microeconomic* model.

If the target of a newly created or arriving entity is chosen at random, we classify the model as *probabilistic* or *random*. If a fair use of resources is the only obvious motivation for the strategy, the model flavor is *fairness*. (Note that our notion of *fair* should be understood in an intuitive sense, not related to any formal definitions of fairness that one can find in the context of distributed systems). Some heuristics are not related to any particular analogy, and are therefore classified as model flavor *none*.

It should be mentioned that the above terms do not form a complete list, because strategies with a model flavor different from the ones mentioned here will surely be encountered now and then.

- **Target Topology:**
  Parallel or distributed computer systems that are used in practice use a variety of different interconnection networks. There are bus-based shared memory multiprocessors, workstation clusters interconnected by ATM networks [ATM95], or distributed memory multiprocessors with static (hypercube, torus) or dynamic (IBM RS/6000 SP) network topologies [Lei92, AG94].

  Even though a strategy does not need to be restricted to any particular type of computer system, it still requires or assumes that the targets are interconnected in an appropriate way. This is due to the fact that a load distribution strategy must use the interconnection network to communicate load information and to transfer load objects.

  In practice, distributed computer systems are often simply networks of workstations interconnected by Ethernet or ATM. Besides, modern parallel computer systems like IBM RS/6000 SP or Cray T3E (see [AG94]) have abandoned the classical store-and-forward routing in favor of more efficient routing paradigms, e.g., wormhole or virtual-cut-through routing or reconfigurable networks. With these developments, the communication latency does not depend much on the distance of the communication partners in the network anymore. Nevertheless, it is still necessary in such networks to limit the contention on individual links or buses. Hence, the target topology remains an important characteristic of parallel systems.

  Another aspect of parallel systems that should be taken into account is whether they are homogeneous (all targets are the same type, and an executable file can be executed on any target) or heterogeneous. If a load distribution strategy can deal with heterogeneous systems, e.g., with heterogeneous NOWs (networks of workstations), the term *heterogeneous* is added to the entry for the criterion target topology in the classification scheme. Otherwise, it is assumed that the strategy deals only with homogeneous systems.

- **Entity Topology:**
  Independent from the target topology, typical parallel application programs have a specific communication structure as well. For example, traditional solvers for differential equations distribute the

data in a *grid-like* pattern and communicate intermediate results only among neighboring processes, whereas divide-and-conquer algorithms usually result in a *tree-like* communication pattern. Another possible scenario features *non-interacting entities* (abbreviated by *n.-i. entities*) that do not communicate with each other at all, e.g., sequential applications running on a parallel system.

Many load distribution strategies do not take the entity topology into account, i.e., they assume that the entities (load objects) are non-interacting. Obviously, such a strategy can cause severe problems if this assumption is not justified. It is likely to create high communication load by placing communicating entities at targets very distant from each other. Other strategies use information about the entity topology in order to keep communicating entities on neighboring targets.

3.1.2 Transfer Model

The basic task of a load distribution strategy is to transfer entities from heavily loaded targets to lightly loaded or idle targets. We refer to such transfers as *load transfers*. The following criteria specify which assumptions a strategy makes about the transfer model.

- **Transfer Space:**
  Whereas some load distribution strategies transfer entities from heavily loaded targets only to neighboring targets, other strategies do not impose such a limit and transfer entities over greater distances in the network. In the former setting, the transfer space is *neighborhood*. If entities are transferred to non-neighboring targets, the transfer space can be *short range* (more than the direct neighbors, but still only a rather small part of the network), *long range* (a substantial part of the network, but not the whole network), or *systemwide* (no restrictions). The distinction between long range and short range cannot be defined formally and remains intuitive in nature. In cases where such a distinction does not make sense, the term *restricted* can be used instead.

- **Transfer Policy:**
  A crucial distinction between different load distribution strategies is whether they transfer an entity from one target to another even if the entity is already being processed (e.g., if the entity is a running UNIX

process), or whether an entity is processed until completion on the target to which it has been assigned in the first place. This distinction between *preemptive* strategies employing migration of currently processed entities and *non-preemptive* strategies employing only initial mapping of entities to targets is ubiquitous in scheduling and load distribution theory.

3.1.3 Information Exchange

In addition to the communication overhead caused by the actual transfer of entities, a load distribution strategy increases the communication load of the network through the exchange of load-state information as well. It is desirable that this exchange of information uses up only a negligible amount of network resources, but it is also clear that good load distribution is difficult to achieve if the information available for evaluating the benefit of potential load transfers is outdated or incomplete.

- **Information Space:**
  Whereas transfer space expresses the distance over which entities are transferred by the load distribution strategy, information space is concerned with the transfer of load-state information. Similar to the transfer space, *systemwide* information space means that the load distribution strategy transfers load-state information without restrictions on the distance of communicating targets in the network, and *restricted* information space can be further divided into *long range*, *short range* or *neighborhood*. Furthermore, there are strategies which transfer load-state information to a single central manager. In this case, the information space is *central*. Finally, strategies which do not transfer any load-state information have *empty* information space.

- **Information Scope:**
  In addition to the information space, load distribution strategies also differ in the extent to which they collect load-state information before reaching a load-transfer decision. Bidding algorithms [SS84] (see Section 3.2.2), for example, typically take into account only the load-state information from a small subset of all targets. Hence, they are classified as having *partial* information scope. The more recently developed precomputation based load distribution algorithm, however, collects load-state information from the whole system (through

local communications) in order to determine the actual load transfers (see [BS96] and Section 3.2.4). Therefore, its information scope is *complete*. As mentioned in Section 3.1.1, there are load distribution strategies that use no load-state information from other targets at all. The information scope of such strategies is classified as *none*.

3.1.4 Coordination

Typically, load distribution problems arise in parallel or distributed systems made up of a large number of more or less independent components. Hence, different load distribution strategies can also be characterized by how they make sure that load-transfer decisions can be reached effectively and in a coordinated manner in such a distributed system.

- **Decision Structure:**
  Due to our limited intuition for parallel and distributed systems, load distribution strategies are understood most easily if they employ a central authority which gathers load-state information and makes all decisions regarding load distribution activities. Unfortunately, such a *centralized* decision structure is likely to create a bottleneck once the system grows larger.

  At the opposite end of the spectrum, there are load distribution strategies where each target in the system can make decisions about potential load transfers. Such a *distributed* decision structure is scalable, but if there is too little coordination among the targets the effects of different load transfers may cancel each other out or even worsen the load imbalance.

  A compromise between these two approaches is to employ a *hierarchical* decision structure. Such a strategy has many of the advantages of distributed decision making (including sufficient scalability) and avoids the bottleneck of a central decision maker by replacing it with a hierarchy of decision makers.

- **Decision Mode:**
  If the load distribution strategy allows targets to decide about potential load transfers independently from each other, we classify the decision mode of the strategy as *autonomous*. For example, a strategy where an overloaded target transfers load to another randomly picked target has this property. Usually, however, targets

cooperate in order to make sure that load transfers take place only if all involved targets agree. We call this latter alternative *cooperative* decision mode. A third possibility is *competitive* decision mode. Microeconomic strategies, for example, can usually be classified as competitive, because entities or targets compete for services mediated by brokers, without consideration for the needs of others.

- **Participation:**
  This criterion classifies a load distribution strategy with respect to the number of targets which participate in load distribution activity at the same time. Many proposed load distribution strategies assume that all targets in a system jointly stop executing tasks at certain times and participate in a global load distribution phase until the load is sufficiently balanced. Frequently, it is also assumed that the system is synchronized and that all targets execute the load distribution activities in lock-step.

  Whereas such strategies with *global* participation are often much easier to analyze, problems arise when they are to be applied in loosely coupled parallel computer systems where global synchronization is very time-consuming. Here, strategies with *partial* participation in load distribution activities seem much more appropriate. If the load on a certain subset of targets is unbalanced, only these targets participate in load transfers while the remaining targets can continue to execute their normal work load.

3.1.5 Algorithm

The classification criteria subsumed here are intended to capture general characteristics of the load distribution algorithm used to implement the strategy.

- **Decision Process:**
  The decision process of a load distribution algorithm is *static* if it does not depend on load-state information accumulated at runtime. Static load distribution algorithms include compile-time partitioning of parallel applications and mapping of tasks to predefined or random locations.

  Typically, however, the term "load distribution algorithm" already implies a *dynamic* decision process, whereas static load distribution algorithms are more commonly referred to as partitioning

algorithms, mapping algorithms, or embedding algorithms. This classification scheme is intended to be used for dynamic load distribution algorithms, and hence many of the other criteria do not apply to static algorithms.

- **Initiation:**
  The detection of load-imbalance in the system is a problem that must necessarily be addressed when one needs to employ a load distribution strategy in practice. Possible alternatives for strategies with partial participation are *sender* initiation with load distribution activity being initiated by overloaded targets, and *receiver* initiation with load distribution activity being initiated by underloaded or idle targets. These two can also be combined (*sender & receiver* initiation). Another possibility is to have a *central* component that initiates load distribution activity. In addition, it is possible to differentiate between *timer-based* and *threshold-based* initiation. The former refers to a strategy that performs load distribution activity after fixed time intervals, the latter to a strategy that starts load distribution as soon as the load or load imbalance exceeds a certain threshold.

- **Adaptivity:**
  An important aspect concerning the flexibility of a load distribution strategy is its adaptivity. A *fixed* strategy is independent of the current overall load level and of the characteristics of the entities present in the system. Since the required load distribution activity depends heavily on these dynamically changing factors, however, many researchers have designed strategies which are *adjustable* to the current load or other properties of the system. Another interesting approach are *learning* strategies that try to improve the effectiveness of their load distribution activities by learning from past experiences.

- **Cost Sensitivity:**
  A load distribution algorithm should not disregard the costs that are necessarily incurred by every kind of load distribution activity, i.e., communication overhead and migration costs. After all, load distribution is usually only a means to achieve the goal of short response-times or high through-put of a system. If a load distribution algorithm keeps the load in the system perfectly balanced at all times but slows down the overall system substantially by the overhead created, this is not a satisfactory solution. Therefore, algorithms should also be classified according to the extent to which they take the costs imposed by load distribution activities

into account. The cost sensitivity of a load distribution algorithm is classified as *none*, *low*, *partial* or *high*.

- **Stability Control:**
  One of the shortcomings of certain load distribution strategies is that they may lead to system instability. A load distribution algorithm is part of the overall system, and the gathering of load-information and the transfers of entities form one target to another contribute to the total load. Furthermore, the load situation may change substantially from time to time, and it is important that the load distribution algorithm does not increase the system load inadequately in such moments due to useless load transfers. In the worst case, it may happen that all resources of the system are occupied by load distribution activities, without any progress being made towards a more balanced state or a more lightly loaded system.

  Therefore, a load distribution algorithm should be designed to keep the system stable at any time. For some algorithms, stability control (i.e., particular steps taken in order to avoid system instability) is *not required*, because any system instability is made impossible by the inherent nature of the algorithm. Stability is *guaranteed* if the load distribution algorithm employs special means that keep the system stable under all conditions. For example, an algorithm can restrict the number of migrations of a single entity. *Partial* stability control adjusts certain parameters according to the current load-situation or according to the number of transfers of entities that happened in the recent past. But in the worst case these precautions might fail to keep the system stable. Finally, there are algorithms that do not take instability problems into account at all. These are classified as having stability control *none*.

Note that the individual classification criteria are not completely orthogonal to each other. In the following, we summarize the obvious dependencies. Classifying transfer space or information space as neighborhood, short range or long range makes sense only if the target topology is not bus-based or fully connected. If information scope is none, information space must be empty. If decision making is centralized, decision agreement must be classified as cooperative (viewing the fact that all targets agree to leave the load distribution decisions to the central authority as a kind of passive cooperation). If the decision process is static, the criteria pertaining to information exchange and to coordination, as well as

the criteria initiation, adaptivity and stability control do not seem to be applicable.

Finally, it should be noted that we have decided not to include the specific criterion which a strategy tries to optimize (e.g., system throughput or total execution time) in the classification scheme. The reason is that the exact optimization criterion is not even specified for many strategies. Furthermore, given a specific optimization criterion and complete advance knowledge about all application characteristics (task execution times, task dependencies, communication requirements), the optimization problem is $\mathcal{NP}$-hard in almost every scenario [GJ79]. Hence, load distribution strategies are usually heuristic approaches, and the quality of a strategy with respect to a certain criterion can frequently only be estimated by simulations.

3.2 A Survey of Load Distribution Strategies in Examples

Due to the vast amount of papers written about load distribution strategies, a comprehensive survey would be beyond the scope of this thesis. So we restrict this survey to four typical examples in order to reveal basic techniques for load distribution. The *diffusion approach* and the *bidding algorithm* are "classical" strategies that are used in quite a number of systems and applications. In addition, we describe two novel algorithms with interesting properties. Many other strategies are surveyed in [LMR91, WLR93, XL94a, FWM94, KGV94, SHK95, XL97, DMP97, Sch97a].

3.2.1 The Diffusion Approach

Diffusion is a well-known load distribution strategy (LDS for short) and has been evaluated theoretically [Cyb89] as well as in practical experiments [WLR93]. The diffusion LDS works like this: at the beginning of a load-distribution round every target collects the load values of all of its neighbors. Consider two neighboring targets i and j. If the load of i is greater than the load of j then i sends a certain part of this load-difference to j. Otherwise i receives some load from j. The amount of load that is transferred depends on the structure of the network. At the end of every load distribution round the targets update their load values. The diffusion

LDS performs a certain number of rounds necessary to achieve a nearly balanced state.

Formally, if we denote the load values of n targets at the beginning of round $t \geq 1$ by $(l_1^t, l_2^t, \ldots, l_n^t)$ the load exchange is given by the equation

$$l_i^{t+1} = l_i^t + \sum_{j \in \Gamma(i)} \alpha_{ij}(l_j^t - l_i^t), \quad 1 \leq i \leq n,$$

where $\Gamma(i)$ is the set of direct neighbors of i and $0 < \alpha_{ij} < 1$ is the diffusion parameter of i and j. As a target cannot give away more load than it possesses, we must have $1 - \sum_{j \in \Gamma(i)} \alpha_{ij} \geq 0$ for $1 \leq i \leq n$. CYBENKO [Cyb89] showed that this iteration converges against the uniform distribution if certain conditions hold.

Of course, the question arises how to set the diffusion parameters to obtain the fastest convergence. For example, it can be shown for mesh connected networks of dimension $d \geq 1$ that $\alpha_{ij} \equiv 1/(2d)$ is optimal [XL94b, XL97]. This means that the load of i (not on the border of the mesh) is just the average of the load of its neighbors after each round.

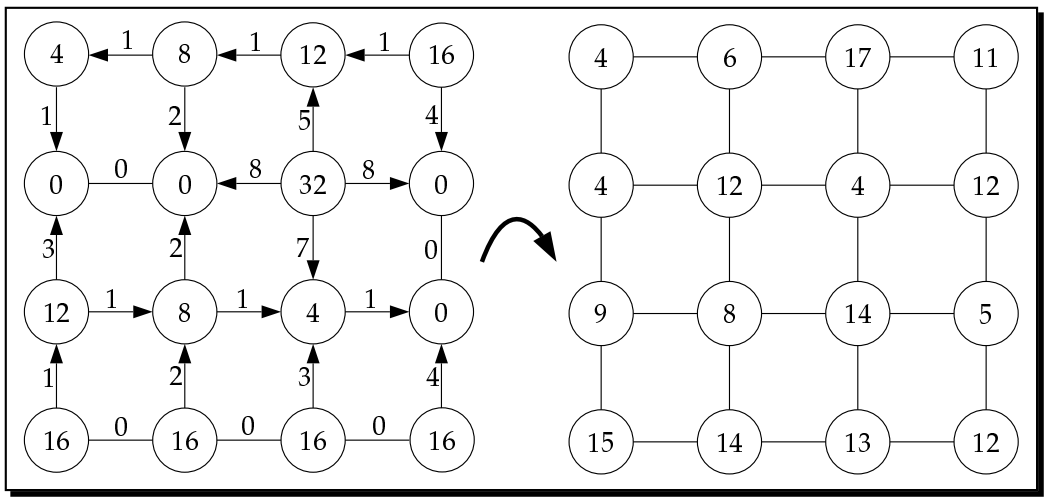

Figure 3.1: Diffusion in a 4×4 mesh

Figure 3.1 shows one iteration of the diffusion algorithm in 4×4 mesh using the optimal diffusion parameter $\frac{1}{4}$. Each target is labeled by its load value and the arrows indicate the amount and direction of the load transfer. Note that the average load is 10 and that the maximal deviation from the mean is reduced from 22 to 7.

The speed of convergence of load balancing algorithms of the diffusion type can be increased by keeping a limited memory of the past load transfers across each edge [GMS96]. More precisely, in each round, the amount of load transferred over an edge depends on the current gradient along this edge as well as the amount transferred in the previous round. However, the load of a target might be smaller than the total load that has to be transferred to its neighbors, and therefore a strategy to fulfill the unsatisfied demands in future rounds is needed.

A further step in this direction is proposed in [DFM98]. They describe a general framework for the analysis of diffusion type LDSs, and present a novel iteration scheme that converges within a number of rounds against the uniform distribution that depends only on the target topology and is independent of the initial load values. Their approach uses ideas similar to the precomputation based LDS described in Section 3.2.4.

In practice, the assumptions that all targets perform load distribution synchronously and the load is arbitrarily divisible are unrealistic, but it is possible to adapt the diffusion LDS for asynchronous systems and integer load values [WLR93, Son94].

3.2.2 Bidding and Balanced Allocations

One common procedure to obtain a fair price for a merchandise is to put it up for auction. Let us briefly describe a simple, no-minimum, one-round auction:

An auctioneer starts with a request for bids for a certain object to a set of bidders and then waits for bids. If any bids are made within a given period of time, the auctioneer selects the best one and knocks down the object to the highest bidder (ties are broken arbitrarily by the auctioneer).

This is the basic idea of bidding algorithms. To perform load distribution, there are two natural possibilities for the job of the auctioneer:

- **Overloaded targets** select some load for transfer to other, hopefully less loaded targets and send requests for bids for the selected entities to a certain number of targets (e.g., its direct neighbors). The targets evaluate the offered piece of load with regard to their own load situation and may choose to return a bid containing their rating of the entity or their current load index. If the initiating target receives any suitable bid in a certain amount of time and is still overloaded, it

transfers the entity to the chosen bidder. Otherwise another request for bids is sent to a greater or different set of potential bidders (see [SS84] for a detailed description of this strategy).

- **Underloaded targets** offer their unused capacity instead of burdensome load. This strategy was proposed by [NXG85] as a distributed "drafting" algorithm for load distribution.

If a target is allowed to make several bids in different auctions that take place concurrently, the strategy has to make sure that all transfers can be fulfilled and that the system remains stable. See [Rad96] for implementation details of the bidding strategy.

Since the analysis of the bidding strategy in a distributed system is very difficult, we now consider an allocation strategy [ABKU94] that is similar to the bidding strategy and admits a thorough analysis. Suppose that we place n balls (i.e., the entities are independent jobs of unit size) one by one into n bins (targets) by putting each ball into a randomly chosen bin. It can be shown that with high probability there are $\Theta(\ln n / \ln \ln n)$ balls in the fullest bin after placing all n balls (see [RS98] for a simple and tight analysis of this result).

However, if for each ball $d \geq 2$ bins are chosen independently and uniformly at random, and the ball is placed into the least full among these bins (ties are broken arbitrarily), then AZAR, BRODER, KARLIN, and UPFAL [ABKU94] showed that the number of balls in the fullest bin drops to $\ln \ln n / \ln d + \Theta(1)$ with high probability. Note that this strategy results in an exponential decrease of the maximum load in comparison to the "blind" random allocation process.

The selection of d bins is directly related to the request for bids of the auctioneer of the bidding strategy, and the chosen bins make bids by responding with their current load. Of course, there is a trade-off between the number of bins queried and the resulting maximum load. The problem of finding optimal values for d when placement costs are taken into account is studied in [DDLM95].

This simple randomized allocation strategy yields good load balancing with high probability using only little information exchange. However, it is a sequential strategy for a simplified load distribution scenario. This "shortcoming" has triggered a series of papers to extend the work of AZAR, et. al. in several directions. In particular, parallel and continuous allocation and load balancing processes, weighted jobs (balls), and other

objective functions such as the maximum waiting time have been considered.

First, it was shown [ACMR95] that this strategy can be enhanced to work in parallel and asynchronous systems by introducing several communication rounds between sending and receiving targets to resolve conflicts. STEMANN [Ste96] presented and analyzed an algorithm for the case that $m > n$ balls are allocated in parallel using the *collision protocol* that originated in randomized shared memory simulations [MSS95]. This work in turn was generalized in [BMS97] to weighted jobs (balls). Their analysis uses the *degree of uniformity* of the weights (i.e., the ratio between the average and the maximum weight) to bound the number of communication rounds that is necessary to obtain a maximum load that is optimal up to a constant factor.

In most systems the load information that is necessary to reach a placement decision has to be retrieved sequentially from the bins. If a fixed number $d \geq 2$ of queries is made to place a ball, even if the first chosen bin is empty, another $d - 1$ bins have to be accessed. CZUMAJ and STEMANN [CS97] therefore presented an adaptive variant of the sequential allocation process where the number of bins queried in order to place a ball depends on the loads of the bins previously inspected by that ball. They also analyzed the impact of a limited number of reassignments on the achievable maximum load.

Another important extension are infinite allocation and continuous load balancing processes. These scenarios are usually specified by a job arrival (generation) process and a deletion (service) process which both last for a potentially infinite time. The main focus of interest is on the long-term behavior of such processes. MITZENMACHER [Mit96a, Mit96b] introduced the "supermarket model" where a Poisson stream of customers (jobs) arrives, and the service time for a customer is exponentially distributed with mean 1. Each customer polls $d \geq 2$ servers and joins the shortest (FIFO) queue. An infinite parallel allocation process is analyzed in [ABS98] using the idea to line up arriving jobs at d queues simultaneously. This requires, however, to delete the unserviced replicates of a job as soon as the job has been serviced for the first time. Finally, a parallel continuous load balancing algorithm for this scenario is proposed and analyzed in [BFM98]. A load balancing step is locally initiated by a processor only if its load is above a certain threshold. If so, d potential balancing partners are randomly chosen. Again, a collision protocol is used to avoid "flooding" of an underloaded target.

3.2.3 Load Distribution by Random Matchings

Many parallel and distributed systems can be modeled as a graph $G = (V, E)$, where the nodes of the graph represent targets and the edges represent direct communication links between targets.

A matching of G is any subset of edges $M \subseteq E$ such that no two edges of M share an endpoint (see Figure 3.2).

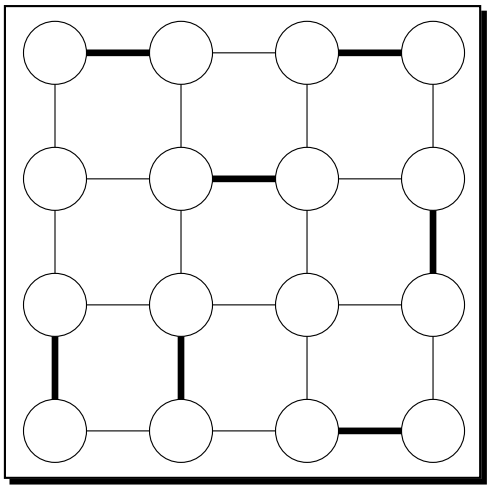

Figure 3.2: Matching (bold lines) in a 4×4 mesh

Matchings are useful to perform local load distribution in parallel: if a node i is an endpoint of an edge $e = \{i, j\}$ in the matching, it tries to reduce the load difference with the other endpoint j. Since no two edges of a matching are adjacent, any node is involved in at most one load transfer. This is a desirable feature due to limited communication capabilities of network interfaces.

The random matchings LDS [GM94, GM96] performs a certain number of synchronous rounds of the following form:

1. Generate a random matching M of G by a distributed algorithm.
2. Equalize the load of the endpoints of each edge in M.

The exact number of rounds necessary to achieve a nearly balanced load distribution with high probability depends on the degree of connectivity of the system.

To describe Step 1 more precisely, let d be the maximum number of neighbors of any node i and let S_i denote a set of edges maintained by each node i. All S_i are empty at the beginning of each round. The generation of a random matching proceeds in two steps executed in parallel by every node:

(a) For every incident edge e: insert e into S_i with probability $\frac{1}{8d}$.

(b) Resolve conflicts by executing an agreement protocol. This protocol makes sure that

- there is at most one edge in S_i and
- if $S_i = \{\{i, j\}\}$ then $S_j = \{\{j, i\}\}$ (and consequently the union of all S_i, $i \in V$, constitutes a matching of G).

An important characteristic of a random matching generated by the above algorithm is that the probability to be part of the matching is at least $1/8d$ for each edge.

To evaluate the effectiveness of an LDS we need a measure for the imbalance in the system. Commonly used measures are, for example, the difference between the highest and lowest load of any target, or the maximum deviation from the mean. In large systems, however, these criteria might not allow to judge the overall load situation correctly since a single target may signal a very bad load situation in an almost perfectly balanced system. Therefore, it is reasonable to define the *potential* of a vector of load values $l_1, l_2, \ldots, l_n$ in a system of n targets as $\Phi = \sum_{i=1}^{n} (l_i - \bar{l})^2$, where $\bar{l}$ denotes the average load. Clearly, the influence of a single target is reduced considerably according to this measure (see [AAG+95] for a detailed discussion of a wider class of imbalance measures).

It can be shown [GM94, GM96] that each round of the random matching LDS reduces the potential by a factor depending on the system topology if the potential is sufficiently large (otherwise load distribution isn't necessary anyway). Furthermore, Step 2 of the algorithm can be modified so that at most one unit of load is transferred in each round. This variant is analyzed in [GLM+95].

3.2.4 Precomputation Based Load Distribution

All LDSs presented so far reach their load transfer decisions on a small information base. Since most load transfers are very time-consuming

(especially when they are preemptive), it is only natural to study algorithms that gather more load information in order to avoid superfluous load transfers.

One way of reducing load transfers is to compute a load distribution scheme before any transfers take place. Let $G = (V, E)$ be a graph representing a parallel or distributed system. We replace each undirected edge $e = \{u, v\}$ in E by two directed edges $(u, v), (v, u)$ to indicate the direction of the load transfers. A load distribution scheme is a function δ that assigns a transfer value to each directed edge such that:

1. $\delta((u, v)) = -\delta((v, u)) \quad \forall \{u, v\} \in E$,
2. $l_v + \sum_{(u,v)} \delta((u, v)) = \bar{l} \quad \forall v \in V$,

where l_v is the load of node v and $\bar{l}$ denotes the average load. For simplicity, we assume that $\bar{l}$ and the δ-values are integers. Load distribution is performed by moving $\delta((u, v))$ load units from u to v if this value is positive. If $\delta((u, v)) < 0$ then u receives this much load from v. Hence, a load distribution scheme is like a road map that we can use to distribute the system load equally.

The precomputation based LDS presented in [BS96] (c.f. also [Böh96, p. 23–95]) is designed for tree-connected systems. The reason is that a load distribution scheme can be computed efficiently for trees. The algorithm consists of two phases:

1. Precompute a load distribution scheme. This requires three steps:
   (a) Starting from the leaves of the tree, every node v (with exception of the root r) calculates the total load of the subtree rooted at v and sends this value to its parent.
   (b) The root calculates the average load $\bar{l}$ and broadcasts $\bar{l}$ to all nodes.
   (c) After receiving the broadcast-message, every node $v \in V \setminus \{r\}$ computes $\delta(v, \text{parent}(v))$. This is accomplished by subtracting $\bar{l} * |\text{nodes\_in\_subtree}|$ from the total load of the subtree rooted at v. Finally, v sends this value to its parent.
2. Perform the actual load transfers according to the load distribution scheme. This is done in a number of rounds because a node might have to wait for some of the load it must transfer. The number of rounds, however, is bounded by the diameter of the tree.

This two phase approach is also employed by the diffusion type algorithm in [DFM98] (see Section 3.2.1). There, the second phase is formalized as a flow scheduling problem, and certain greedy heuristics for this problem are analyzed.

It is straightforward to generalize the precomputation based LDS to system-topologies that are cross products of trees. A grid (or mesh), for example, is the cross product of two linear arrays. First, the algorithm balances all rows in parallel and is then applied a second time to balance the columns of the grid.

Table 3.2: Classification of diffusion and bidding

	Diffusion	Bidding
System Model		
Model Flavor	physical	microeconomic
Target Topology	mesh	fully connected
Entity Topology	n.-i. entities	n.-i. entities
Transfer Model		
Transfer Space	neighborhood	systemwide
Transfer Policy	preemptive	non-preemptive
Information Exchange		
Information Space	neighborhood	systemwide
Information Scope	partial	partial
Coordination		
Decision Structure	distributed	distributed
Decision Mode	cooperative	competitive
Participation	global	partial
Algorithm		
Decision Process	dynamic	dynamic
Initiation	timer-based	sender & receiver
Adaptivity	fixed	fixed
Cost Sensitivity	none	none
Stability Control	none	none

In Table 3.2 and Table 3.3, four of the load distribution strategies discussed in this section are classified according to our classification scheme. It should be mentioned that the initiation of load distribution activities is not clearly specified for the diffusion strategy [Cyb89], for the strategy that employs random matchings [GM94, GM96], and for the precomputation based strategy [BS96]. These strategies can also be implemented using dif-

ferent initiation than given in the table. Furthermore, we note that stability control may be considered as *not required* for these three strategies, because they employ load distribution phases with global participation. These load distribution phases always result in a balanced system. Nevertheless, it is possible that load distribution phases are initiated at unfavorable moments and cause an unnecessary slowdown of the system, and hence we choose to classify stability control for these strategies as *none*.

Table 3.3: Classification of random matchings and precomputation based load distribution

	Matchings	PLB
System Model		
Model Flavor	combinatorial	combinatorial
Target Topology	arbitrary	tree
Entity Topology	n.-i. entities	n.-i. entities
Transfer Model		
Transfer Space	neighborhood	neighborhood
Transfer Policy	preemptive	preemptive
Information Exchange		
Information Space	neighborhood	neighborhood
Information Scope	partial	complete
Coordination		
Decision Structure	distributed	hierarchical
Decision Mode	cooperative	cooperative
Participation	global	global
Algorithm		
Decision Process	dynamic	dynamic
Initiation	timer-based	central
Adaptivity	fixed	fixed
Cost Sensitivity	none	none
Stability Control	none	none

Chapter 4

On-Line Scheduling of Parallel Jobs with Runtime Restrictions

In many situations the problem arises to find a schedule for a set of parallel jobs [FR95, FR96, FRS+97, BEP+96]. Such a set could be, for example, a parallel query execution plan generated by the query optimizer of a parallel database management system [Rah96, GI97]. In this chapter we propose on-line scheduling algorithms for such problems that generate satisfactory schedules if the individual running times of the jobs do not differ too much.

The scheduling model studied in this chapter assumes that each parallel job demands a fixed number of processors or a specified sub-system of a certain size and topology (depending on the underlying structure of the parallel machine considered) for its execution. It is not possible to run a parallel job on fewer processors than requested, and additional processors will not decrease the running time. This reflects the common practice that the decision on the number of processors is made before a job is passed to the scheduler based on other resource requirements like memory, disk-space, or communication intensity. The processors must be allocated exclusively to a job throughout its execution, and a job cannot be preempted or restarted later. This is a reasonable assumption because of the large overhead for these activities on parallel machines. Furthermore, there may be precedence constraints between the jobs. A job can only be executed if all of its predecessors have already completed execution. Most frequently, precedence constraints arise from data dependencies such that a job needs the complete input produced by other jobs before it can start computation.

We are concerned with on-line scheduling throughout this chapter to capture the fact that complete a priori information about a job system is rarely available. However, it has been shown [FKST98, Sga94] that the worst-case performance of any deterministic or randomized on-line algorithm for scheduling parallel job systems with precedence constraints and arbitrary running times of the jobs is rather dismal, even if the precedence constraints between the jobs are known in advance. Therefore, we study the case that there is some a priori knowledge about the execution times of the individual jobs but the dependencies are unknown to the scheduler.

Three different gradations for this additional knowledge are studied in this chapter. The first model of runtime restrictions requires that all job running times are equal and that this fact is known to the on-line scheduler. We give a level-oriented on-line algorithm for this problem that repeatedly schedules a set of available jobs using a packing algorithm and collects all jobs that arrive during a phase for execution in the next phase. For parallel systems that support arbitrary allocation of processors to jobs and 1-dimensional arrays we show that this algorithm is 2.7-competitive if the FIRST FIT BIN PACKING heuristic is used. Due to a lower bound of 2.691 on the competitive ratio for every deterministic on-line scheduler, our algorithm is almost optimal. For hypercube connected machines, we present an optimal on-line scheduling algorithm with competitive ratio 2. Further results are derived for 2-dimensional arrays and a general theorem for arbitrary interconnection topologies is given.

We then explore the entire bandwidth between unit and arbitrary execution times and capture the variation of the individual job running times by a characteristic parameter that we call **runtime ratio** (the ratio of the longest and shortest running time). The results for the proposed on-line schedulers demonstrate a smooth, linear transition of the competitive ratio from the case of unit execution times to unrelated execution times that is governed by the runtime ratio. Our second model postulates that the runtime ratio of a job system is reasonably small and that the on-line scheduler knows the shortest execution time (but not the runtime ratio itself). For any $T_R \geq 2$ a family of job systems with runtime ratio T_R is given that bounds the competitive ratio of any deterministic or randomized on-line scheduler by $(T_R+1)/2$ from below. This general lower bound holds for any network topology and remains valid even if the scheduler knows the actual runtime ratio in advance.

An on-line scheduler designated RRR (Restricted Runtime Ratio) for parallel systems supporting arbitrary allocations is described, and we

demonstrate that this algorithm is $(T_R/2 + 4)$-competitive for any job system with runtime ratio at most T_R. Therefore, the RRR Algorithm is nearly optimal up to a small additive constant. The assumption that the shortest execution time is known to the on-line scheduler can be dropped without much loss of competitive performance. We present a modified algorithm called RRR_ADAPTIVE for this third model, and show it to be $(T_R/2 + 5.5)$-competitive. Two main characteristics of these algorithms are discussed to show that they are essential to reach a competitive ratio that is close to the lower bound. Further results are given for other network topologies.

The remainder of this chapter is organized as follows. In Section 4.1 we introduce our scheduling model, some notation and definitions, as well as basic techniques for analyzing on-line scheduling algorithms. We then discuss previous and related work on on-line scheduling of parallel jobs in Section 4.2. Section 4.3 presents nearly optimal on-line schedulers for jobs with unit execution time, whereas in Section 4.4 we study job systems where the ratio of the running times of the longest and shortest job is bounded. We describe and analyze on-line scheduling algorithms for parallel systems supporting arbitrary allocations that are optimal up to small additive constants.

4.1 Preliminaries

Let N denote the number of processors of the parallel computer-system at hand. A *job system* is a non-empty set of jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_m\}$ where each job specifies the type and size of the sub-system that is necessary for its execution together with precedence constraints among the jobs in $\mathcal{J}$ given as a partial order $\prec$ on $\mathcal{J}$. If $J_a \prec J_b$, J_b cannot be scheduled for execution before J_a is completed. We define the *size* of a job as the number of processors it requests. A *task* is a job of size 1, i.e., it requires exactly one processor for execution. Note that tasks as well as jobs of size N can appear in job systems for any network topology. A job system that only contains tasks is a *task system*. A *schedule* for a job system $(\mathcal{J}, \prec)$ is an assignment of the jobs to processors and start-times such that:

- each job is executed on a sub-system of appropriate type and size,
- all precedence constraints are obeyed,
- each processor executes at most one job at any time,
- jobs are executed non-preemptively and without restarts.

The interconnection topology of the parallel computer-system may impose serious restrictions on the *job types* that can be executed efficiently on a particular machine. On a hypercube, for example, it is reasonable to execute jobs only on subcubes of a certain dimension rather than on an arbitrary subset of the processors. On the other hand, a number of interconnection networks do not restrict the allocation of processors to parallel jobs. For example, the CLOS-network of the very popular *IBM RS/6000 SP* system, which uses an oblivious buffered wormhole routing strategy, justifies the assumption that the running time of a job only weakly depends on a specific processor allocation-pattern (see [AG94, p. 512] for a short description of this system and [SSA+94] for in-depth information on its interconnection network). Therefore, the various types of interconnection networks have to be treated separately.

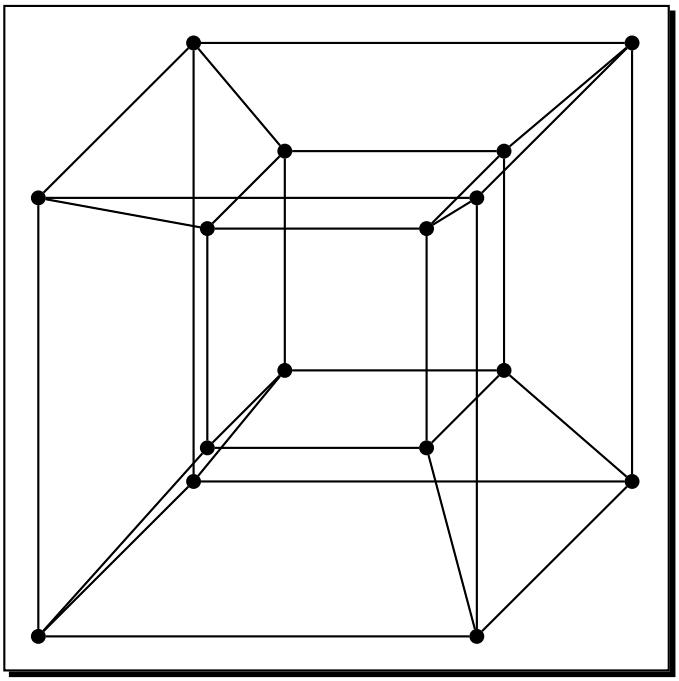

Figure 4.1: 4-dimensional hypercube

The *complete model* assumes that a job J_a requests n_a processors ($1 \leq n_a \leq N$) for execution and any subset of processors of size n_a may be allocated. The terminology has been chosen in analogy to a complete graph on N nodes. This model was called PRAM model in [Sga94, FKST93], but since the results for this model are not restricted (cf. the above discussion) to this abstract parallel machine model we prefer to use the name 'complete model'.

The *r-dimensional hypercube* (see Figure 4.1) consists of $N = 2^r$ processors, labeled from 0 to $N-1$, and has $r2^{r-1}$ point-to-point communication links. Two processors are connected iff the binary representations of their labels (an r-bit string) differ in exactly one bit. As a consequence, each processor is directly connected to $r = \log_2 N$ other processors (see [Lei92] for properties of hypercubes). A job J_a can only request a d_a-dimensional subcube ($0 \leq d_a \leq r$) for its execution.

Another topology frequently used for parallel computing is the *r-dimensional array* with side-lengths $(N_1, N_2, \ldots, N_r)$, $N_i \geq 2$ for $i = 1, 2, \ldots, r$ (also called r-dimensional grid or mesh). The label of a processor is an r-dimensional vector $x = (x_1, x_2, \ldots, x_r)$ with $0 \leq x_i < N_i$ for $i = 1, 2, \ldots, r$. Two processors x and y are connected iff $\sum_{i=1}^{r} |x_i - y_i| = 1$. Note that hypercubes form the subclass of arrays with side-length 2 in every dimension. Eligible job types are sub-arrays with side-lengths $(N'_1, N'_2, \ldots, N'_r)$, $1 \leq N'_i \leq N_i$. The dimension of a job can be less than r if one or more of the N'_i are equal to 1.

It is always possible to transform a job system $(\mathcal{J}, \prec)$ into a directed acyclic graph $D = (\mathcal{J}, E)$ with $(J_a, J_b) \in E \Leftrightarrow J_a \prec J_b$. Removing all transitive edges from D we obtain the *dependency graph* induced by $(\mathcal{J}, \prec)$ (see Figure 4.4 on page 53 for an example). We call two jobs J_a and J_b *dependent* if $J_a \prec J_b$ or $J_b \prec J_a$, and *independent* otherwise. We shall use the terms *dependency* and *precedence constraint* interchangeably. The *length of a path* in the dependency graph induced by $(\mathcal{J}, \prec)$ is defined as the sum of the running times of the jobs along this path. A path is called *critical* if its length is maximum among all paths in the dependency graph induced by $(\mathcal{J}, \prec)$.

Assume that all jobs have running time 1 and let P be a longest path in D ending at job J. Then *depth*(J) is defined as the number of nodes in P. If we partition a schedule for such a unit execution time (*UET*) job system into timesteps of length 1, the depth of a job indicates the earliest possible timestep (*EPT*) in which J can be scheduled. The i'th *level* of D is the set of (independent) jobs $\{ J \in \mathcal{J} \mid \text{depth}(J) = i \}$. Motivated by the above observation, a level of D is often referred to as an EPT level of the corresponding job system $(\mathcal{J}, \prec)$.

A job is *available* if all predecessors of this job have completed execution. An on-line scheduling algorithm is only aware of available jobs and has no knowledge about their successors. We assume that the on-line scheduler receives knowledge about a job as soon as the job becomes available. This event, however, may depend on earlier scheduling decisions.

The *work* of a job is defined as the number of requested processors, multiplied by its running time. A schedule preserves the work of a job if the processor-time product for this job is equal to its work. The *efficiency* of a schedule at any time t is the number of busy processors at time t divided by N. In general, the running time of a job is also unknown to the on-line scheduler and can only be determined by executing a job and measuring the time until its completion. In Section 4.3, though, we study the case of unit execution times and therefore restrict the on-line model there to the case of unknown precedence constraints.

Throughout the chapter we use the notations shown in Table 4.1 (cf. [Sga94, FKST98]) for a given job system $(\mathcal{J}, \prec)$. To simplify our presentation, we do not attach the job system or schedule as arguments to the notations in Table 4.1. The relationships should always be clear from the context. Further notation is introduced when needed.

Table 4.1: Frequently used notations

T_{opt}	Length of an optimum off-line schedule for $(\mathcal{J}, \prec)$
T_{ALG}	Length of a schedule for $(\mathcal{J}, \prec)$ generated by algorithm ALG
$T_{\max}$	Maximal length of any path in the dependency graph induced by $(\mathcal{J}, \prec)$
$t_{\min}$	Minimal running time of any job in $\mathcal{J}$
$t_{\max}$	Maximal running time of any job in $\mathcal{J}$
$\|S\|$	Length of a schedule S
$T_{<\alpha}$	Total time of a schedule for $(\mathcal{J}, \prec)$ when the efficiency is less than α, $0 \leq \alpha \leq 1$

Our goal is to generate schedules with *minimum makespan*, i.e., to minimize the completion time of the job finishing last. We evaluate the performance of our on-line scheduling algorithms by means of competitive analysis [ST85, BE98, FW98]. We call a deterministic on-line scheduling algorithm ALG *c-competitive* if for **all** N: $T_{\text{ALG}} \leq c \cdot T_{\text{opt}}$ for **all** job systems that can be executed on N processors. The infimum of the values $c \in [1, \infty]$ for which this inequality holds is called the *competitive ratio* of ALG.

It is possible to define the competitiveness of an on-line scheduler as a function of N rather than for arbitrary N. We call ALG *$f(N)$-competitive for N processors*, if $T_{\text{ALG}} \leq f(N) \cdot T_{\text{opt}}$ for all job systems that can be executed on N processors, where $f : \mathbb{N} \to \{ x \in \mathbb{R} \mid x \geq 1 \}$. However, for technical

reasons, we do not define a competitive ratio in this case. There are two advantages of the latter definition. First, it can provide more detailed information about the competitiveness of an algorithm for particular choices of N. Second, it allows to distinguish between on-line algorithms that have an asymptotically unbounded competitive ratio. We will use the first definition almost exclusively since $f(N)$ is often quite complicated and we will encounter only algorithms with constant competitive ratio.

For randomized algorithms working against an oblivious adversary we have to modify these definitions slightly. Let RALG be a randomized on-line scheduler. We call RALG c-competitive if for **all** N: $\mathbb{E}[T_{\text{RALG}}] \leq c \cdot T_{\text{opt}}$ for **all** job systems that can be executed on N processors, where the expectation is taken over all random choices made by RALG. The infimum of the values $c \in [1, \infty]$ for which this inequality holds is called the *(expected) competitive ratio* of RALG. As above it is possible to modify this definition such that the competitiveness of RALG is a function of N.

The competitive ratio clearly is a worst-case measure. It is intended to compare the performance of different on-line algorithms that solve the same problem, since it is in general impossible to compute an optimum solution without complete knowledge of the problem instance. An *optimal* on-line algorithm is one with a best possible competitive ratio. See [BE98, FW98] for a thorough treatment of the fundamental concepts in on-line computation and competitive analysis.

The following two lemmata provide useful tools for the competitive analysis of our scheduling algorithms.

Lemma 4.1 *Let S be a schedule for a job system $(\mathcal{J}, \prec)$ such that the work of each job is preserved. Let $0 \leq \alpha_1 \leq \alpha_2 \leq 1$ and $\beta \geq 0$. Suppose that the efficiency of S is at least α_1 at all times and $T_{<\alpha_2} \leq \beta T_{\text{opt}}$. Then*

$$|S| \leq \left(\beta + \frac{1 - \alpha_1\beta}{\alpha_2}\right) T_{\text{opt}}.$$

See [Sga94] for a proof of this lemma.

Lemma 4.2 *Consider a schedule for a job system $(\mathcal{J}, \prec)$. Then there exists a path of jobs in the dependency graph induced by $(\mathcal{J}, \prec)$ such that whenever there is no job available to be scheduled, some job of that path is running.*

This lemma is due to GRAHAM [Gra66, Gra69]. The proof given there still holds for parallel jobs since it uses only the structure of the dependency graph.

4.2 Previous and Related Work

Extensive work on non-preemptive on-line scheduling of parallel jobs with or without precedence constraints was done by FELDMANN, KAO, SGALL, and TENG [FKST98, Sga94, FST94]. However, these results for general job systems are bad news for users of parallel computers since they show that no deterministic on-line scheduler for N processors can have competitive ratio better than N. That is, the competitive ratio is asymptotically unbounded, and even randomization cannot improve this unsatisfactory situation substantially.

One possibility to improve the performance is to restrict the maximum job size to λN processors, $0 < \lambda < 1$. Given this restriction it has been shown that the GREEDY algorithm is optimal for the complete model with competitive ratio $1 + \frac{1}{1-\lambda}$. Setting $\lambda = 1/2$, for example, yields a 3-competitive algorithm.

Another alternative is the use of *virtualization*. This means that a parallel job J_a which requests n_a processors is executed on a smaller number of processors n'_a by the use of simulation techniques with a predetermined increase in running time. Under the assumption of proportional slowdown (the running time of a job is enlarged by the factor n_a/n'_a) it can be shown that there is an optimal on-line scheduler for the complete model with competitive ratio $1 + \Phi$, where $\Phi = (1 + \sqrt{5})/2$ is the golden ratio. This improves a previous off-line result of WANG and CHENG [WC92] with asymptotic performance guarantee 3. For the hypercube, an $O(\log N / \log\log N)$-competitive algorithm has been given, and similar results [FKST98, Sga94] hold for arrays. The two approaches just described can be combined to yield an optimal on-line scheduler with competitive ratio $2 + \frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$ for the complete model.

Both approaches, though, have a severe drawback that arises due to the memory requirements of parallel jobs. Restricting the maximum size of a job to λN processors can thus severely restrict the problem size that can be solved on a particular machine. This is often unacceptable in practice because solving large problems is the main reason for the use of parallel computers besides solving problems fast. Virtualization may be impossible or prohibitively expensive if such memory limitations exist.

The job systems used in the lower bound proofs in [FKST98, Sga94] for the general case reveal an unbounded ratio of the running times of the longest and shortest job. Therefore, we think it necessary to study the

influence of the individual running times on the competitive ratio of on-line schedulers for our scheduling problem. To gain insight into this relationship it is only natural to start with unit execution times as is done in Section 4.3. It turns out that the problem becomes manageable with small constant competitive ratio even if nothing is known about the precedence constraints.

To fill the gap between these two extremes — totally unrelated running times versus unit execution times — we identify the runtime ratio (the ratio of the running time of the longest and shortest job) as the distinctive parameter of a job system for the achievable competitive ratio. The importance of this parameter has also been demonstrated recently in [CM96] for **off-line** scheduling of jobs with multiple resource demands, both malleable (allowing for virtualization with proportional slowdown) and non-malleable.

Although we are interested in on-line scheduling, it might be appropriate to briefly mention some complexity results for the corresponding off-line problems. Not surprisingly, almost any variant of these scheduling problems is $\mathcal{NP}$-hard. BŁAŻEWICZ, DRABOWSKI, and WĘGLARZ [BDW86] have proved that it is strongly $\mathcal{NP}$-hard to compute optimum schedules for job systems with unit execution time and no dependencies if N is part of the problem instance. For any fixed N they showed that the problem can be solved in polynomial time. Furthermore, it is known [GJTY83] that the problem is $\mathcal{NP}$-hard for task systems with precedence constraints that are the disjoint union of an in-forest and and an out-forest. The scheduling problem for job systems with arbitrary job running times and without dependencies is strongly $\mathcal{NP}$-hard for every fixed $N \geq 5$ [DL89]. If precedence constraints consisting of a set of chains are involved, the problem of computing an optimum 2-processor schedule for a job system is also strongly $\mathcal{NP}$-hard [DL89].

4.3 Jobs with Unit Execution Time

In this section, we restrict our model to the case where all jobs have the same execution time. When the dependency graph is known to the scheduler this problem has been intensively studied by GAREY, GRAHAM, JOHNSON and YAO [GGJY76]. We show that similar results hold in an on-line environment, where a job is available only if all its predecessors have completed execution.

4.3.1 A General Lower Bound

We will show in this section that for any network topology no deterministic or randomized on-line scheduling algorithm can achieve a competitive ratio better than 2. In the randomized case we will assume an oblivious adversary.

We assume a parallel system with $N \geq 2$ processors. To prove the claim for deterministic algorithms, we use the following job system. This job system has $N-1$ levels with $N+1$ tasks on each level. Again, the dependencies are assigned dynamically by the adversary according to the decisions of the deterministic on-line scheduler. The task from level i, $1 \leq i \leq N-2$, scheduled last by the on-line scheduler is designated to be predecessor of all tasks on level $i+1$. Therefore any deterministic on-line scheduler ALG needs at least 2 timesteps to schedule all tasks of one level. In an optimum schedule, on each level the task with dependencies is scheduled first together with $N-1$ other tasks from the same level. The $N-1$ remaining tasks are scheduled in timestep N. This gives the desired lower bound on the competitive ratio of deterministic algorithms:

$$\frac{T_{\text{ALG}}}{T_{\text{opt}}} \geq \frac{2(N-1)}{N} = 2 - \frac{1}{N}\,.$$

To derive the lower bound for randomized on-line schedulers working against an oblivious adversary, we need only a slightly more complicated job system. It now consists of N levels with $N+\sqrt{N}$ tasks on each level (for convenience we may assume w.l.o.g. $\sqrt{N} \in \mathbb{N}$). N tasks on level i, $1 \leq i \leq N-1$, are selected by the adversary in advance to be predecessors of all tasks on level $i+1$. We refer to the tasks without successors in this job system as terminal tasks. The optimum solution first schedules all non-terminal tasks level by level. This takes time $N-1$. Then, the $(N-1)\sqrt{N} + N + \sqrt{N} = N\sqrt{N} + N$ remaining terminal tasks are scheduled in additional $\sqrt{N}+1$ timesteps. Thus, $T_{\text{opt}} = N + \sqrt{N}$.

The randomized algorithm RALG receives $N+\sqrt{N}$ independent tasks from level 1 in the beginning or if a new level becomes available. Only after termination of the N non-terminal tasks on level i, $i \leq N-1$, the $N+\sqrt{N}$ tasks on the next level become available. We define the length of the partial schedule for level i, $1 \leq i \leq N-1$, as the time that elapses from the moment when level i becomes available to the moment when level $i+1$ becomes available. Clearly, the sum of these partial schedule

lengths is a lower bound on the total length of the on-line schedule. If RALG schedules any one of the $\sqrt{N}$ terminal tasks on level i before a non-terminal task, the length of the partial schedule for level i will be at least 2. Therefore, we have to bound the probability that RALG schedules all N non-terminal tasks first. In this case the length of the partial schedule will be at least 1. Since there is only one choice for this event to happen out of $\binom{N+\sqrt{N}}{N}$ possible choices, we can upper bound the probability that the length of the partial schedule is in the interval $[1, 2)$ by

$$\binom{N+\sqrt{N}}{N}^{-1} \leq \left(\frac{N+\sqrt{N}}{N}\right)^{-N} = \left(1+\frac{1}{\sqrt{N}}\right)^{-N} < 2^{-\sqrt{N}}.$$

By linearity of expectation we have:

$$\mathbb{E}[T_{\text{RALG}}] \geq (N-1)(2^{-\sqrt{N}} \cdot 1 + (1-2^{-\sqrt{N}}) \cdot 2) \geq 2N - 4.$$

Thus,

$$\frac{\mathbb{E}[T_{\text{RALG}}]}{T_{\text{opt}}} \geq \frac{2N-4}{N+\sqrt{N}} = \frac{2-4/N}{1+1/\sqrt{N}} \quad \xrightarrow[N\to\infty]{} \quad 2.$$

Summarizing, we have the following

Theorem 4.3 *For an arbitrary network topology the competitive ratio of any deterministic or randomized on-line algorithm for scheduling UET job systems is at least* 2.

The job systems used in the proof of Theorem 4.3 contain no parallel jobs. Therefore the derived lower bound is valid for task systems as well as job systems. This result has been obtained independently by EPSTEIN [Eps98].

Interestingly, the above lower bound for deterministic algorithms is identical to the lower bound proved by SHMOYS, WEIN, and WILLIAMSON [SWW95] for task systems with arbitrary running times but without precedence constraints.

4.3.2 Complete Model

In this section we present a generic on-line scheduler for UET job systems. We will show that a certain variant of this algorithm is optimal for the complete model up to a very small additive constant.

The LEVEL Algorithm (see Figure 4.2) collects all jobs that are available from the beginning. Since available jobs are independent we can easily transform the problem of scheduling these jobs to the BIN PACKING problem: the size of a job divided by N is just the size of an item to be packed, and the timesteps of the schedule correspond to the bins (see [CGJ96] for a survey on BIN PACKING). Let PACK be an arbitrary BIN PACKING algorithm. We parameterize the LEVEL Algorithm with PACK to express the fact that a schedule for a set of independent jobs is generated according to PACK. Thereafter, the available jobs are executed as given by this schedule. Any jobs that become available during this execution phase are collected by the algorithm. After the termination of all jobs of the first level a new schedule for all available jobs is computed and executed. This process repeats until there are no more jobs to be scheduled. Note that the LEVEL Algorithm works for any interconnection topology if an appropriate packing algorithm is used as a subroutine. We will exploit this observation in the following sections.

```
Algorithm LEVEL(PACK):
begin
    while not all jobs are finished do
        begin
        A := { J ∈ J | J is available };    // next EPT level
        schedule all jobs in A according to PACK;
        wait until all scheduled jobs are finished;
        end;
end.
```

Figure 4.2: The LEVEL(PACK) Algorithm

First, we use the Next-Fit (NF) bin-packing heuristic for scheduling on each level. NF packs the items in given order into a so-called active bin. If an item does not fit into the active bin, the active bin is closed and never used again. A previously empty bin is opened and becomes the next active bin.

Theorem 4.4 LEVEL(NF) *is* 3*-competitive.*

Proof: The number of iterations of the while-loop is exactly the length of a critical path in the dependency graph. There are two possibilities for each level:

1. The partial schedule for this level has length 1. Let T_1 denote the number of levels of this type.

2. The partial schedule for this level has length ≥ 2. By the packing rule of NF it is clear that the average efficiency of 2 consecutive timesteps in such a partial schedule is $> 1/2$. From this we conclude that the average efficiency of all timesteps except possibly the last one is $> 1/2$. Let T_2 denote the number of final timesteps with efficiency $< 1/2$ in partial schedules for levels of this type.

Since $T_1 + T_2 \leq T_{\max} \leq T_{\text{opt}}$ we can apply Lemma 4.1 with $\alpha_1 = 1/N$, $\alpha_2 = 1/2$, $\beta = 1$, yielding:

$$T_{\text{LEVEL(NF)}} \leq \left(3 - \frac{2}{N}\right) T_{\text{opt}} .$$ □

Since NF can be implemented to run in linear time (in the number of items to be packed) the scheduling overhead is very low when NF is used to compute partial schedules. Now we use the First-Fit (FF) bin-packing heuristic instead of NF to achieve a better result with only a modest increase of the scheduling overhead. FF in contrast to NF considers all partially filled bins as possible destinations for the item to be packed. An item is placed into the first (lowest indexed) bin into which it will fit. If no such bin exists, a previously empty bin is opened and the item is placed into this bin. It has been shown [Joh74] that FF has time-complexity $\Theta(n \log n)$ for a list of n items.

Theorem 4.5 LEVEL(FF) *is* 2.7*-competitive.*

The proof of this theorem uses the weighting function from [GGJY76]. Let $W : [0, 1] \to [0, 8/5]$ be defined as follows (cf. Figure 4.3):

$$W(\alpha) = \begin{cases} \frac{6}{5}\alpha & \text{for} \quad 0 \leq \alpha \leq \frac{1}{6}, \\ \frac{9}{5}\alpha - \frac{1}{10} & \text{for} \quad \frac{1}{6} < \alpha \leq \frac{1}{3}, \\ \frac{6}{5}\alpha + \frac{1}{10} & \text{for} \quad \frac{1}{3} < \alpha \leq \frac{1}{2}, \\ \frac{6}{5}\alpha + \frac{4}{10} & \text{for} \quad \frac{1}{2} < \alpha \leq 1. \end{cases}$$

We also need the following results from [GGJY76]:

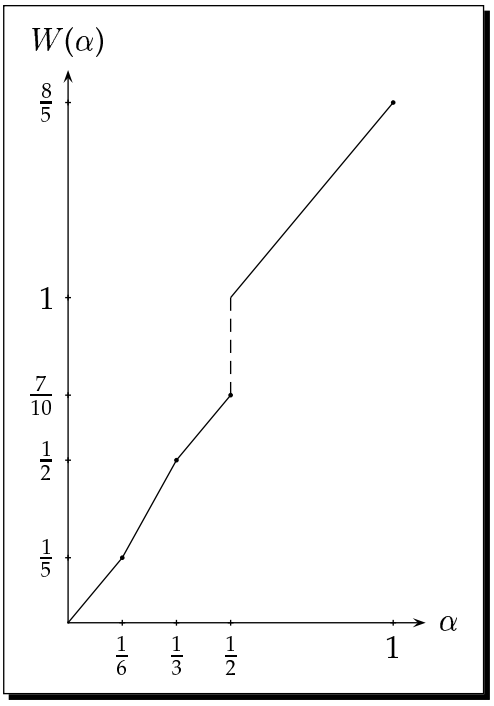

Figure 4.3: Weighting function for the analysis of BIN PACKING algorithms

Lemma 4.6 *Let B denote a set of items with total size ≤ 1. Then*

$$\sum_{b \in B} W(\text{size}(b)) \leq \frac{17}{10}.$$

If all sizes are $\leq 1/2$,

$$\sum_{b \in B} W(\text{size}(b)) \leq \frac{3}{2}.$$

Theorem 4.7 *If L is a list of items with sizes ≤ 1,*

$$\text{FF}(L) < \sum_{x \in L} W(\text{size}(x)) + 1.$$

Together with the above lemma this theorem provides the best known upper bound for the number of bins used by first-fit. If L^* is the number of bins used in an optimum packing of L, first-fit uses at most $\lceil (17/10)L^* \rceil$ bins. Now we are ready to prove Theorem 4.5:

Proof: Let $\mathcal{J}$ be a job system with unit execution time and arbitrary precedence constraints. We define

$$\overline{W}(\mathcal{J}) = \sum_{j \in \mathcal{J}} W(\text{size}(j)/N).$$

Thus $\overline{W}(\mathcal{J})$ is the total weight of all job sizes. Let l be the number of levels of the job system. For $1 \leq i \leq l$ let U_i be the set of jobs of each level. By Theorem 4.7 we can upper bound the length of the partial schedule for each level i, $1 \leq i \leq l$, generated by LEVEL(FF):

$$T_{\text{LEVEL(FF)}}(U_i) < \overline{W}(U_i) + 1.$$

We can think of an optimum packing of $\mathcal{J}$ with the dependencies removed as a partition of $\mathcal{J}$ into $\mathcal{J}^*$ sets each of which has total size ≤ 1. Applying Lemma 4.6 yields $\overline{W}(\mathcal{J}) \leq \frac{17}{10}\mathcal{J}^*$. Together with the fact that the length of the optimum schedule for $\mathcal{J}$ without dependencies cannot be longer than the length of the optimum schedule for $\mathcal{J}$ we conclude:

$$\begin{aligned} T_{\text{LEVEL(FF)}} &= \sum_{i=1}^{l} T_{\text{LEVEL(FF)}}(U_i) \\ &< \sum_{i=1}^{l} (\overline{W}(U_i) + 1) \\ &= \overline{W}(\mathcal{J}) + l \leq 1.7\, T_{\text{opt}} + l. \end{aligned}$$

Since $l = T_{\max} \leq T_{\text{opt}}$, the result follows. □

The result for the LEVEL(FF) Algorithm is nearly optimal. To show this, we give an asymptotic lower bound of 2.691 on the competitive ratio of each deterministic on-line scheduling algorithm. For the sake of clarity, we first prove a slightly weaker lower bound of $2.69\, T_{\text{opt}} - 4$ on the length of a schedule generated by a deterministic on-line scheduler. Using Salzer numbers we refine this construction to derive the asymptotic lower bound.

Fix $N \in \mathbb{N}$, $N \geq 7 \cdot 1806$, and let

$$A := \left\lfloor \frac{N}{2} \right\rfloor + 1, \qquad B := \left\lfloor \frac{N}{3} \right\rfloor + 1,$$
$$C := \left\lfloor \frac{N}{7} \right\rfloor + 1, \qquad D := N - A - B - C - 1.$$

The job system (see Figure 4.4) consists of $l \geq 4$ levels with one chain of $l-4$ tasks and l jobs of size A, $l-1$ jobs of size B, $l-2$ jobs of size C, $l-3$ jobs of size D.

Additional dependencies are assigned dynamically by the adversary depending on which parallel job of each level is scheduled last by the on-line algorithm. This is possible because the on-line scheduler cannot distinguish between the parallel jobs on the same level. The optimum schedule has length l and is shown in Figure 4.5. Here, the parallel job with successors is scheduled first on each level. Contrary to the optimum solution, the on-line scheduler is forced to schedule and execute all jobs on one level to make the jobs on the next level available. The schedule generated by LEVEL(FF) is thus the best possible on-line schedule (see Figure 4.6) and has length

$$l + \left\lceil \frac{l-1}{2} \right\rceil + \left\lceil \frac{l-2}{6} \right\rceil + \left\lceil \frac{l-3}{42} \right\rceil + (l-4) > 2.69\, l - 4,$$

if $2 \nmid (l-1)$, $6 \nmid (l-2)$, and $42 \nmid (l-3)$. It is easy to see that any $l \in \mathbb{N}$ with $42 \mid l$ fulfills the above conditions.

The following sequence $(t_i)_{i \in \mathbb{N}}$ was investigated by SALZER [Sal47]:

$$\begin{aligned} t_1 &= 2, \\ t_{i+1} &= t_i(t_i - 1) + 1, \quad \text{for} \quad i \geq 1. \end{aligned}$$

The first five numbers of this sequence are $2, 3, 7, 43, 1807$. Closely related is the following series:

$$h_\infty = \sum_{i=1}^{\infty} \frac{1}{t_i - 1} > 1.69103. \tag{4.1}$$

There are two basic relations for the Salzer numbers that can be derived inductively from their definition:

$$\sum_{i=1}^{k} \frac{1}{t_i} + \frac{1}{t_{k+1} - 1} = 1, \qquad \prod_{i=1}^{k} t_i = t_{k+1} - 1.$$

Let $A_i = \lfloor N/t_i \rfloor + 1$, $1 \leq i \leq k$, be the sizes of the parallel jobs on the first k levels. Setting $A_{k+1} = N - \sum_{i=1}^{k} A_i - 1$, we can conclude that

$$A_{k+1} < \frac{N}{t_{k+1} - 1} - 1, \qquad A_{k+1} \geq \frac{N}{t_{k+1} - 1} - (k+1).$$

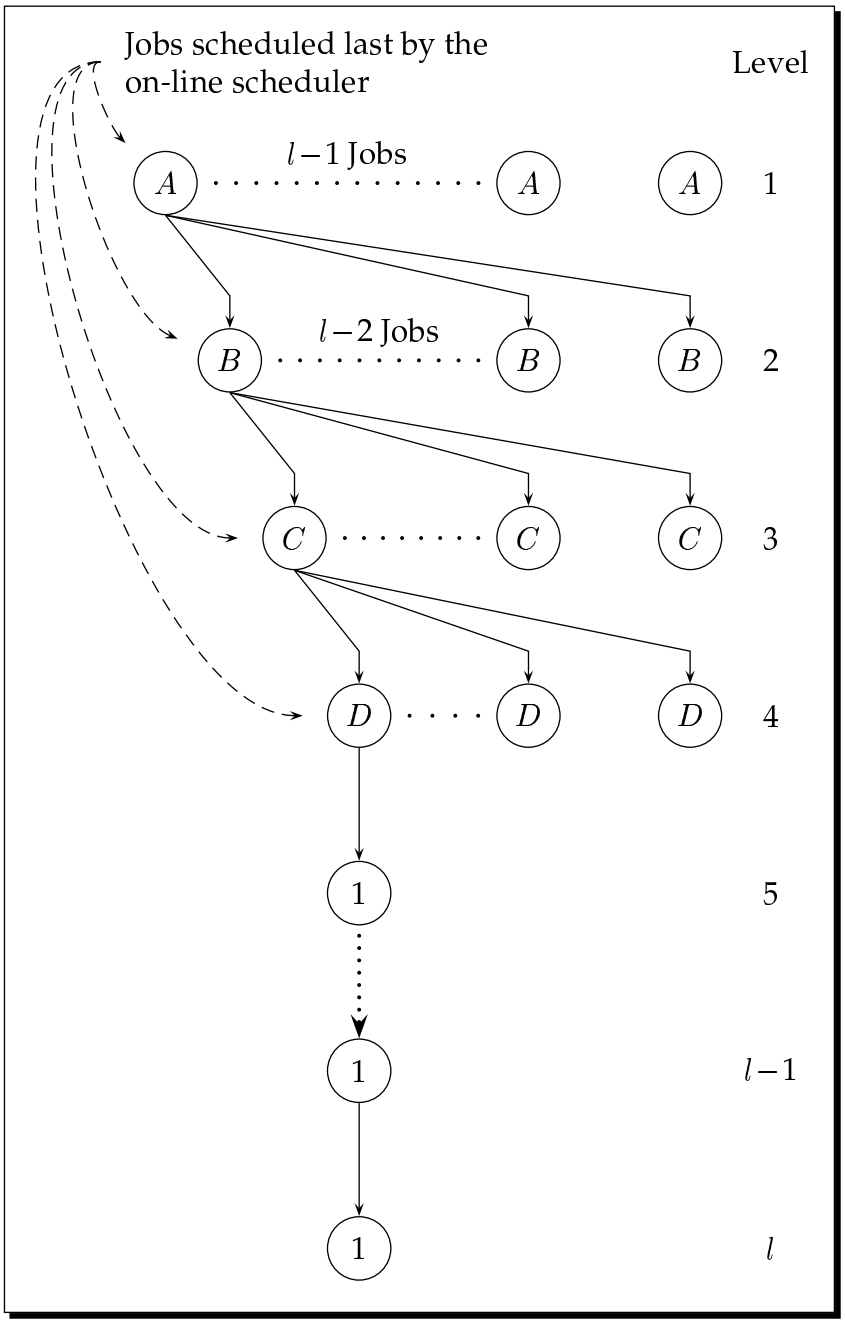

Figure 4.4: Job system used in lower bound proof

It is easy to see that $t_{k+1} - 1$ jobs of size A_{k+1} can be scheduled in one timestep on $N - 1$ processors. To ensure that no more than $t_{k+1} - 1$ jobs of size A_{k+1} can be co-scheduled on N processors we choose $N > (k + 1)(t_{k+2} - 1)$. The job system again consists of $l \geq k + 1$ levels with one chain

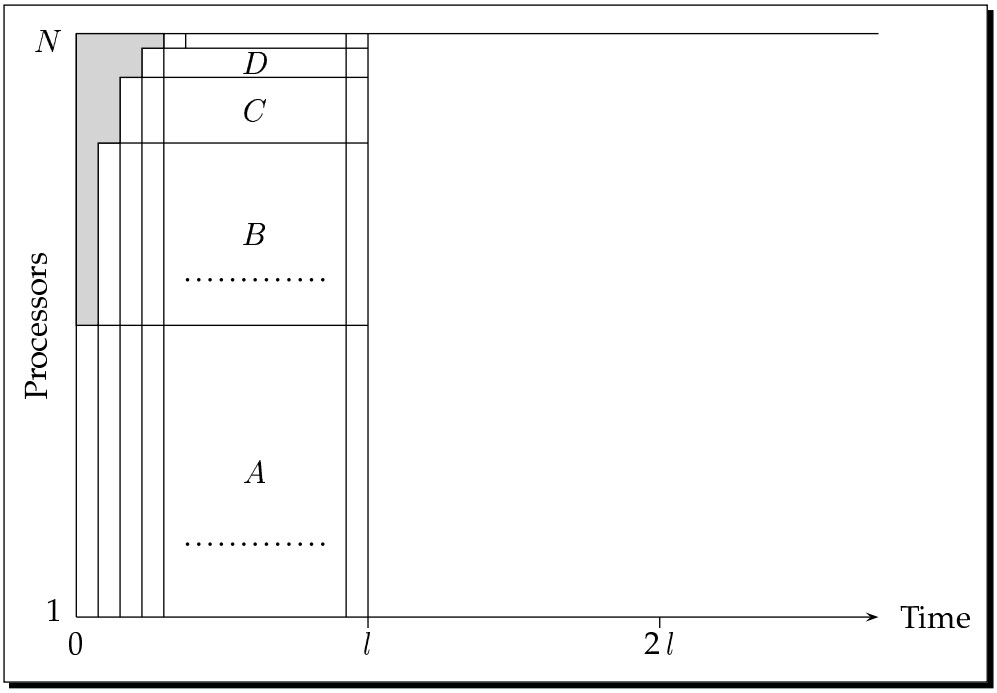

Figure 4.5: Optimum schedule

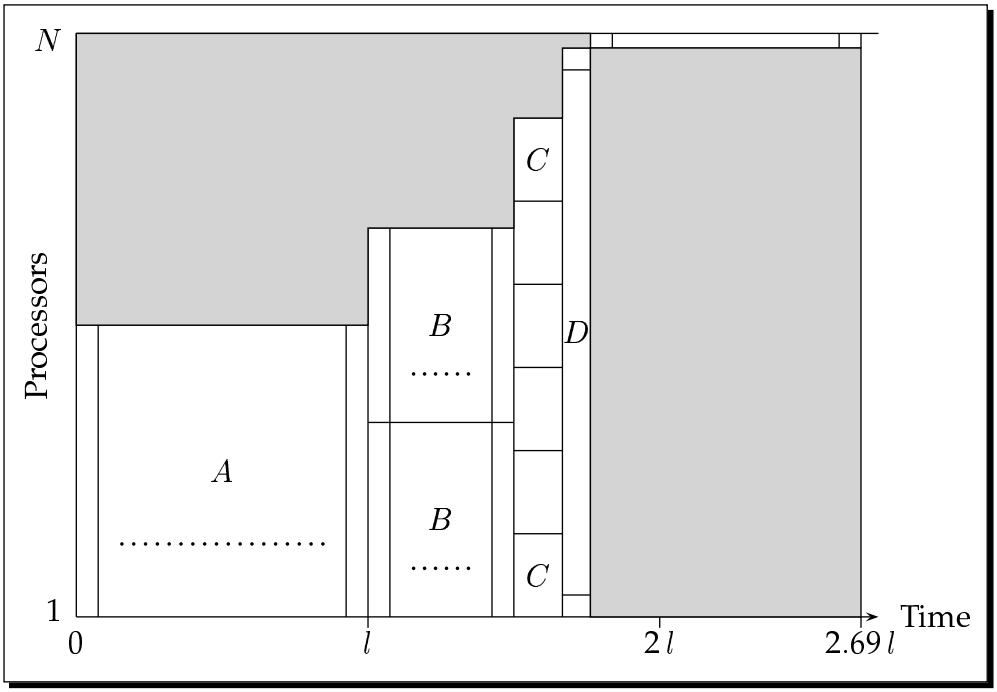

Figure 4.6: On-line schedule generated by LEVEL(FF)

of $l-(k+1)$ tasks and $l-(i-1)$ jobs of size A_i, $1 \leq i \leq k+1$. Dependencies are assigned dynamically as above. The length of the optimum schedule is l, whereas every schedule generated by a deterministic on-line scheduler has length at least

$$\sum_{i=1}^{k+1} \left\lceil \frac{l-(i-1)}{t_i - 1} \right\rceil + l - (k+1).$$

From this and (4.1) we see that the competitive ratio can be brought arbitrarily close to $1 + h_\infty$ for $k \to \infty$, $l = \omega(k)$.

The competitive ratio of LEVEL(FF) is in the interval $[1 + h_\infty, 2.7]$. This can be improved if the maximum size of a job is restricted to $\lfloor N/2 \rfloor$:

Theorem 4.8 LEVEL(FF) *is* 2.5*-competitive, if no job requests more than half of the total number of processors.*

Proof: Analogous to the proof of Theorem 4.5 using the second inequality of Lemma 4.6. □

The same bound holds if the Next-Fit-Decreasing (NFD) bin-packing heuristic (presort the items in non-increasing order, then use NF) is used instead of FF. This follows easily form the fact that the average efficiency of 2 consecutive timesteps in a partial schedule for a level generated by NFD is $> 2/3$ in this case.

Similarly to the unrestricted case, an asymptotic lower bound > 2.4 on the competitive ratio of any deterministic on-line scheduler for this problem can be derived. Further restrictions of the maximum job size might yield somewhat better competitive ratios for the LEVEL Algorithm, but this situation is already handled well by the GENERIC Algorithm in [FKST93, Sga94] which achieves competitive ratio $1 + 1/(1 - \lambda)$, if no job requests more than λN, $0 < \lambda < 1$, processors. For example, $\lambda = 1/2$ yields competitive ratio 3 for the GENERIC Algorithm that is valid for job systems with arbitrary execution times.

We also remark that the results of this section remain valid if we assume a 1-dimensional array of length N as interconnection topology instead of using the complete model, since the BIN PACKING algorithms assign consecutive processors to the jobs and the assignments in different timesteps are independent from each other.

4.3.3 Hypercube

In this section the problem of on-line scheduling job systems with arbitrary precedence constraints and unit execution times for hypercube connected parallel computers is studied. We will show that the LEVEL Algorithm, using a suitable packing algorithm for hypercubes, achieves competitive ratio 2 which matches the lower bound of Theorem 4.3 exactly. Hence, this algorithm is optimal.

It is not difficult to schedule a set of independent parallel jobs each of which requests a subcube of a certain dimension. First, we sort the jobs by size in non-increasing order. To avoid fragmentation, we use only normal subcubes for job execution:

Definition 4.9 *A k-dimensional subcube is called* normal, *if the labels of all its processors differ only in the last k positions.*

For each timestep of our schedule we allocate jobs from the head of the sorted list to normal subcubes while there are unscheduled jobs left and the hypercube is not completely filled. If the timestep is full, we have to add a new timestep to our schedule (if there are any unscheduled jobs left). It is easy to see that the efficiency of this schedule for independent jobs is 1 in all timesteps except possibly the last. We refer to this strategy as PACK_HC. The algorithm for job systems with arbitrary dependencies is just the LEVEL algorithm using PACK_HC instead of a 1-dimensional BIN PACKING algorithm.

Theorem 4.10 LEVEL(PACK_HC) *is an optimal deterministic on-line scheduler with competitive ratio* 2.

Proof: The number of iterations of the while-loop is exactly the length of a critical path in the dependency graph. Thus $T_{<1} \leq T_{\max} \leq T_{\text{opt}}$. Since the efficiency of the schedule is at least $1/N$ all the time, we have by Lemma 4.1:

$$T_{\text{LEVEL\_HC}} \leq \left(1 + \frac{1 - 1/N}{1}\right) T_{\text{opt}} = \left(2 - \frac{1}{N}\right) T_{\text{opt}}.$$

Since Theorem 4.3 implies the optimality of LEVEL(PACK_HC) the claim of the theorem follows. □

4.3.4 2-Dimensional Array

We have seen in Section 4.3.2 that the problem of computing an on-line schedule for a UET job system with dependencies for a 1-dimensional array can be solved by using BIN PACKING as a subroutine. In this section we generalize these results for 2-dimensional arrays. Let (N_1, N_2), $N_i \geq 2$, $i = 1, 2$, denote the vector of side-lengths of the 2-dimensional array, and let (j_1, j_2), $1 \leq j_i \leq N_i$, denote the vector of side-lengths of a parallel job J to be scheduled. If we normalize the side-lengths to fit into the unit square $[0, 1]^2$, i.e., $(x, y) := (j_1/N_1, j_2/N_2)$, the problem of scheduling a set of independent jobs transforms into the 2-DIMENSIONAL GEOMETRIC PACKING [GW95] problem which is defined as follows:

INSTANCE: List of items $L = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, $0 < x_i, y_i \leq 1$, $1 \leq i \leq n$.

SOLUTION: A packing of L into a sequence of unit squares such that

1. each item is entirely contained in exactly one unit square with all sides parallel to the sides of the bin,
2. no two items contained in the same bin overlap,
3. the orientation of each item is the same as the orientation of the unit square in which it is contained (i.e., items must not be rotated).

MEASURE: The number k of unit squares used for the packing.

GOAL: Minimize k.

The generalization of this problem to higher dimensions is straightforward.

As already seen in Sections 4.3.2 and 4.3.3 the main difficulty is to find a "good" packing algorithm. We adapt a result of LI and CHENG [LC90] for 3-DIMENSIONAL GEOMETRIC PACKING. Although this packing algorithm is asymptotically inferior to the Hybrid First-Fit (HFF) Algorithm [CGJ82] it yields a better upper bound on the competitive ratio for the LEVEL Algorithm due to a smaller additive constant in the performance estimates.

The basic building blocks of this packing algorithm are the following two algorithms called L^x and L^y. Both algorithms try to pack a given list of items into **one** unit square by packing certain subsets of items into layers. A *layer* (or *shelf*) of a packing in direction of the x-axis is a rectangle of width 1 such that

1. the interior of each item is either completely inside or completely outside of the rectangle,
2. each item inside the rectangle intersects the bottom of the rectangle,
3. every vertical line through the rectangle intersects at most one item.

The height of a horizontal layer is determined by the maximum height of all the items it contains. Vertical layers are defined analogously.

Algorithm L^x first packs all items with $x_i \geq 1/2$ and puts each such item in a separate horizontal layer starting at the origin of the unit square. All remaining items are packed in their relative order into layers using the Next-Fit Algorithm in direction of the x-axis: Whenever an item does not fit into the active layer, the active layer is closed and never used again. A new layer is created atop the previous layer (if possible) and becomes the active layer. Algorithm L^y is identical to Algorithm L^x except that it works in the direction of the y-axis.

A very similar packing algorithm (assuming that rotation of items is allowed) was described by MEIR and MOSER [MM68]. The following lemma is an immediate consequence of Theorem 4 in their article:

Lemma 4.11 *Let $L = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, $x_i, y_i \leq 1$, $1 \leq i \leq n$, be a list of items such that:*

1. $x_i \geq y_i$, $1 \leq i \leq n$,
2. $y_1 \geq y_2 \geq \cdots \geq y_n$,
3. $\sum_{i=1}^{n} x_i y_i \leq 7/16$.

Then L can be packed into a unit square using Algorithm L^x.

An analogous statement holds for Algorithm L^y. It is shown in [LC90] that if any one of the three conditions in the above lemma is violated, the algorithm may not be able to generate a legal packing.

To generate a packing for an arbitrary list of items we split L into 2 sublists $L_x := \{(x_i, y_i) \in L \mid x_i \geq y_i\}$ and $L_y := \{(x_i, y_i) \in L \mid y_i > x_i\}$. Items with size $x_i y_i > 7/32$ are packed into separate unit squares. The remaining sublists are sorted by the appropriate coordinate and then split further into sublists with total size between $7/32$ and $7/16$ that can now be packed by Algorithm L^x or L^y respectively. We refer to this algorithm as PACK_2D. The next lemma is obvious from the description of PACK_2D:

Lemma 4.12 *If* $L = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ *is a list of items, then*

$$\text{PACK\_2D}(L) \leq \frac{32}{7} \sum_{i=1}^{n} x_i y_i + 2,$$

where PACK_2D(L) *denotes the number of unit squares used by* PACK_2D *to pack* L.

To compute an on-line schedule for UET job systems with dependencies for 2-dimensional arrays we use the LEVEL Algorithm in combination with PACK_2D.

Theorem 4.13 LEVEL(PACK_2D) *is* $46/7$*-competitive.*

Proof: For each EPT level in the dependency graph there are at most 2 time steps in the on-line schedule generated by LEVEL(PACK_2D) with efficiency less than $7/32$. Let $N := N_1 N_2$. If $N \geq 5$ we can apply Lemma 4.1 with $\alpha_1 = 1/N$, $\alpha_2 = 7/32$, and $\beta = 2$. This yields:

$$T_{\text{LEVEL(PACK\_2D)}} \leq \left(2 + \frac{1 - 2/N}{7/32}\right) T_{\text{opt}} = \left(\frac{46}{7} - \frac{64}{7N}\right) T_{\text{opt}}.$$

For $N < 5$ the claim follows from the fact that the efficiency of the on-line schedule is always at least $1/N$. □

To obtain a lower bound for deterministic on-line scheduling algorithms we adapt the job system of the lower bound construction for the complete model (cf. Section 4.3.2). The only change occurs in the definition of the job side-lengths. Let

$$\begin{aligned}
A &:= \left(\left\lfloor \frac{N_1}{2} \right\rfloor + 1, \left\lfloor \frac{N_2}{2} \right\rfloor + 1\right), \\
B &:= \left(\left\lfloor \frac{N_1}{2} \right\rfloor + 2, N_2 - \left\lfloor \frac{N_2}{2} \right\rfloor - 1\right), \\
C &:= \left(N_1 - \left\lfloor \frac{N_1}{2} \right\rfloor - 2, \left\lfloor \frac{N_2}{2} \right\rfloor + 1\right), \\
D &:= \left(N_1 - \left\lfloor \frac{N_1}{2} \right\rfloor - 1, N_2 - \left\lfloor \frac{N_2}{2} \right\rfloor - 1\right).
\end{aligned}$$

It is easy to see that jobs of type A cannot be scheduled together. Furthermore, at most 2 jobs of type B, at most 2 jobs of type C, and at most 4 jobs of type D can be executed in one timestep if $N_i \geq 13$, $i = 1, 2$.

The optimum solution needs time l because it is possible to schedule one job of type A–D together with a task from the chain (see Figure 4.7).

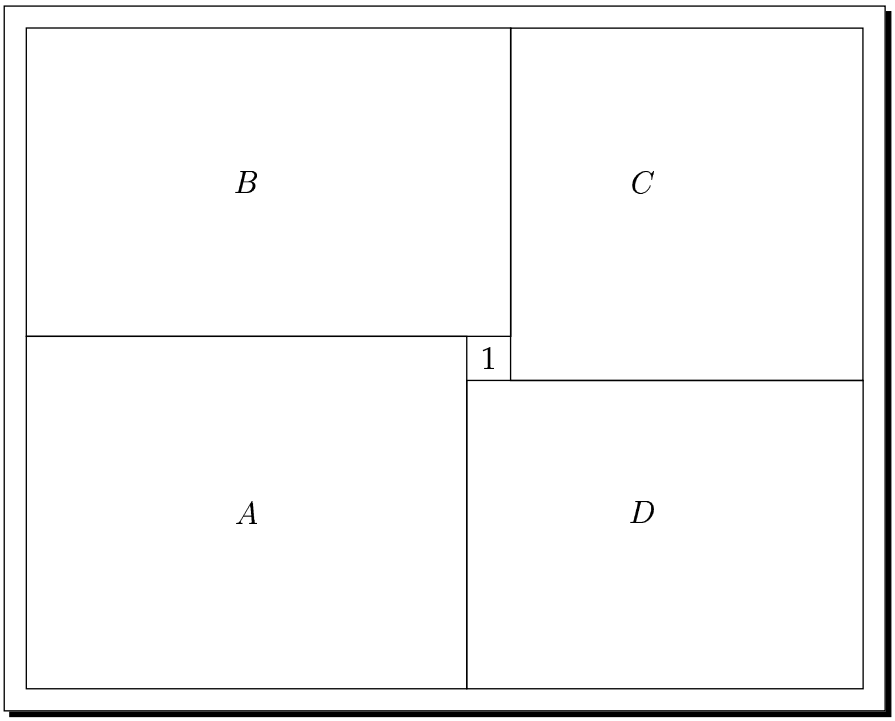

Figure 4.7: Packing of jobs in the optimum schedule

Again, the on-line scheduler can be forced to schedule the job system level by level and therefore the schedule length is at least

$$l + \left\lceil \frac{l-1}{2} \right\rceil + \left\lceil \frac{l-2}{2} \right\rceil + \left\lceil \frac{l-3}{4} \right\rceil + (l-4) > 3.25\, l - 5.5,$$

if $l \geq 6$ is even.

Thus, we have an asymptotic lower bound of 3.25 on the competitive ratio of any deterministic on-line scheduler for 2-dimensional arrays with side-lengths at least 13.

4.3.5 Other Topologies

We have seen so far that the LEVEL Algorithm performs quite well for several interconnection topologies that are frequently used in parallel systems. For these results the existence of an efficient packing algorithm that

guarantees a reasonable utilization of the resources is fundamental. The following theorem gives a general result for arbitrary networks:

Theorem 4.14 *Let* PACK *be a packing algorithm for a given network T, let $(\mathcal{J}, \prec)$ be a job system with job types suitable for T, $\mathcal{J}' \subset \mathcal{J}$ an arbitrary subset of independent jobs, and $w : \mathcal{J} \to \mathbb{R}^+$ a weighting function. If* $\text{PACK}(\mathcal{J}') \leq \sum_{j \in \mathcal{J}'} w(j) + c$ *for a constant $c \in \mathbb{N}_0$, and if $\sum_{j \in \mathcal{J}} w(j) \leq a \cdot T_{\text{opt}}$ for a constant $a \in \mathbb{R}^+$, it holds that*

$$T_{\text{LEVEL(PACK)}} \leq (a + c) T_{\text{opt}} .$$

The proof of this theorem is analogous to the proof of Theorem 4.5. It is not hard to see that all upper bounds on the competitive ratio of our on-line algorithms in this section could be derived using the above theorem. However, since the weighting function w is very simple in all but one case, we preferred to give the more direct proofs using Lemma 4.1. This approach has also the advantage of yielding somewhat better bounds on the competitive ratio for small numbers of processors.

4.4 Job Systems with Restricted Runtime Ratio

We have shown in the preceding section that for various network topologies on-line scheduling of parallel jobs with unit execution time and precedence constraints is possible with small constant competitive ratio. On the other hand, if execution times are arbitrary, there exists no on-line scheduling algorithm with acceptable competitive performance for this model [FKST98, Sga94]. It is therefore only natural to explore the case that job runtimes are restricted by some criterion other than unit execution time in order to achieve a satisfactory competitive ratio.

For a set of jobs $\mathcal{J}$ we therefore define the *runtime ratio* $\text{RR}(\mathcal{J}) := t_{\max}/t_{\min}$. In this section we study the problem of on-line scheduling job systems with dependencies where the runtime ratio is bounded from above by a parameter $T_R \geq 1$ which is not known to the on-line scheduler. This problem often arises in practice when a priori estimates of the maximal and minimal running time of any job are available but the actual running times are unknown. This situation also makes it clear that the parameter T_R cannot be used as additional information for scheduling decisions by the on-line scheduler and is therefore not part of the problem

instance. Indeed, our results show that this knowledge is not necessary for the on-line scheduler to achieve a near optimal competitive ratio that depends only on T_R .

4.4.1 A General Lower Bound

First, we give a lower bound of $(T_R + 1)/2$ on the competitive ratio of any deterministic on-line scheduler that holds for any interconnection topology.

For simplicity we normalize the running time of the shortest job to 1. The job system used in this lower bound argument is very simple (see Figure 4.8) and consists of $N \geq 2$ layers with two tasks and one parallel job of size N on each layer. The parallel job depends on one of the tasks on the same layer and is predecessor of both tasks of the following layer. The task scheduled first by the on-line scheduler is assigned running time T_R and the remaining task runs for 1 unit of time and is predecessor of the parallel job which also needs 1 unit of time for execution.

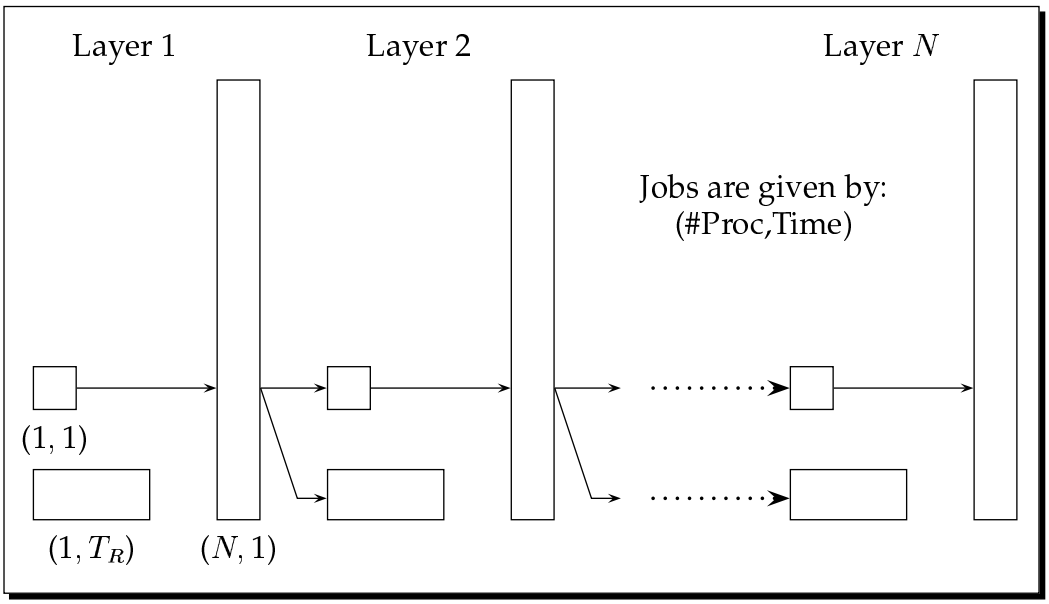

Figure 4.8: Difficult job system for RRR-scheduling

Clearly, the makespan of any schedule generated by an on-line scheduler is at least $N(T_R + 1)$. If T_R is sufficiently large (e.g., $T_R \geq 2$), the optimum solution first schedules the critical path which has length $2N$

followed by the tasks of length T_R in parallel. The competitive ratio of any deterministic on-line scheduler is thus lower bounded by

$$\frac{N(T_R+1)}{2N+T_R} \quad \xrightarrow[N\to\infty]{} \quad \frac{T_R+1}{2}.$$

Next we show that the lower bound of $(T_R+1)/2$ also holds for randomized on-line scheduling algorithms working against an oblivious adversary. To show this, we modify the job system slightly. Each layer of the job system now has $\lceil\sqrt{N}\rceil$ tasks instead of two. Exactly one of these $\lceil\sqrt{N}\rceil$ tasks is predecessor of the parallel job of size N. This task and the parallel job have running time 1 whereas all remaining tasks run for time T_R. The parallel job is predecessor of all tasks of the following layer (if any). All running times and dependencies are assigned in advance and do not depend on the decisions and random choices of the randomized on-line scheduler.

We can lower bound the length of the partial schedule for each layer generated by the on-line algorithm as follows. In the best case the on-line scheduler picks only one task for execution and this task is the predecessor of the parallel job on the same layer. Then the length of the partial schedule for this layer is at least 2. In any other case the length of the partial schedule will be at least T_R+1 because a task with running time T_R has to be executed ahead of the parallel job. From this we can readily compute a lower bound on the expected schedule length for any randomized on-line scheduler RALG. Since no on-line algorithm can distinguish between the tasks of one layer, the probability that the length of the partial schedule is in the interval $[2, T_R+1)$ is at most $1/\lceil\sqrt{N}\rceil$ which is the probability to choose the "right" task. Using linearity of expectation we conclude:

$$\begin{aligned} E[T_{\mathrm{RALG}}] &\geq N\left(1/\lceil\sqrt{N}\rceil\cdot 2+\left(1-1/\lceil\sqrt{N}\rceil\right)\cdot(T_R+1)\right)\\ &= N(T_R+1)+o(N). \end{aligned}$$

As before it is easy to see that the optimum schedule length is $2N+(\lceil\sqrt{N}\rceil-1)T_R$. This yields the desired lower bound on the competitive ratio of any randomized on-line scheduling algorithm against an oblivious adversary:

$$\frac{N(T_R+1)+o(N)}{2N+(\lceil\sqrt{N}\rceil-1)T_R} \quad \xrightarrow[N\to\infty]{} \quad \frac{T_R+1}{2}.$$

We summarize the preceding discussion in the following

Theorem 4.15 *For an arbitrary network topology the competitive ratio of any deterministic or randomized on-line algorithm for scheduling any job system* $(\mathcal{J}, \prec)$ *with* $\mathrm{RR}(\mathcal{J}) \leq T_R$ *on that topology cannot be smaller than* $(T_R + 1)/2$.

4.4.2 Complete Model

We now describe an algorithm for the complete model designated RRR (see Figure 4.9) that is $(T_R/2 + 4)$-competitive. A key feature of this algorithm is the distinction between *big* jobs that request more than half of the total number of processors and *small* jobs with size $\leq \lfloor N/2 \rfloor$.

Let $\alpha := \alpha(t)$ denote the efficiency at time t. The RRR Algorithm tries to keep the efficiency at least $1/2$ whenever possible. There are two reasons that hinder the RRR Algorithm from achieving this goal. First, there might be no job available and second, there might not be enough processors available to schedule a big job.

The second case is much more severe than the first one which can be handled by the GRAHAM argument (cf. Lemma 4.2) without much loss of performance. Therefore, the RRR Algorithm must prevent big jobs from being delayed too long in order to bound the fraction of the total schedule length with low efficiency. This is done by occasionally stopping to schedule small jobs, if all big jobs request more processors than currently available and the efficiency is below $1/2$.

We present two versions of the RRR Algorithm. The first one assumes that $t_{\min}$ is a known quantity. Again, we normalize the running time of the shortest job to 1 and a *unit of time* refers to this normalized time quantum. In the second version we remove this assumption and employ an adaptive waiting strategy to maintain a comparable competitive ratio. The RRR Algorithm maintains two sets, L_1 and L_2, containing the available big and respectively small jobs. We assume that any job that becomes available is immediately inserted into the appropriate set, and we will not state this activity explicitly in the pseudo-code description of our algorithms.

Theorem 4.16 *The RRR Algorithm is* $(T_R/2+4)$*-competitive for any job system* $(\mathcal{J}, \prec)$ *with* $\mathrm{RR}(\mathcal{J}) \leq T_R$.

Proof: We partition the schedule generated by the RRR Algorithm into 3 different kinds of phases:

```
Algorithm RRR
begin
    while L_1 ≠ ∅ do
        schedule a big job exclusively;
    while not all jobs are finished do
        begin
        while L_2 ≠ ∅ do
            schedule small jobs greedily;
        if L_1 ≠ ∅ then
            if a big job can be scheduled then
                do it;
            else
                if α ≥ 1/2 then
                    wait for a scheduled job to finish;
                else   // start of a delay phase
                    collect small jobs that become available
                    during the next 2 units of time;
                    schedule those jobs greedily and
                    then wait for all scheduled jobs to finish;
                    while L_1 ≠ ∅ do
                        schedule a big job exclusively;
                fi;
            fi;
        else
            wait for next available job;
        fi;
        end;
end.
```

Figure 4.9: The RRR Algorithm

1. Efficiency is at least $1/2$.
2. Efficiency is below $1/2$ and there is no job available.
3. Efficiency is below $1/2$ and the algorithm waits for the termination of all jobs.

We refer to the third type as a *delay phase* and denote the total time of each kind by $T_{\geq 1/2}$, T_{nojob}, and T_{delay} respectively. The total time of the RRR schedule that is spent in phases of type 1 and 2 can easily be bounded by

$3\,T_{\text{opt}}$, because we have $T_{\geq 1/2} \leq 2\,T_{\text{opt}}$ by a straightforward area-argument and $T_{\text{nojob}} \leq T_{\max} \leq T_{\text{opt}}$ by Lemma 4.2.

It remains to show that $T_{\text{delay}} \leq (T_R/2+1)T_{\text{opt}}$. We define a *delayed job* as a big job that was available at the beginning of a delay phase. Let t_i denote the start time of delay phase i. First, we bound the length of a delay phase by $T_R + 2$. If no small jobs become available during the first two units of time after the beginning of a delay phase, no more jobs are scheduled until all currently running jobs terminate. Since the running time of any job is no more than T_R, such a delay phase lasts at most time T_R.

On the other hand, if small jobs become available during the first two units of time, these are collected and scheduled greedily at time $t_i^s = t_i + 2$ (resp. $t_i^s < t_i + 2$ if all jobs running at time t_i terminate before two units of time have elapsed) in addition to those jobs still running at time t_i^s. If the total size of these small jobs (i.e., the total number of processors that all these small jobs request) is no more than the number of idle processors at time t_i^s, they can be scheduled immediately. Clearly, the length of a delay phase is bounded by $T_R + 2$ in this case. Should the total size of the small jobs exceed the number of idle processors at time t_i^s we can schedule enough small jobs to raise the efficiency above 1/2 as long as small jobs that were collected during the interval $[t_i, t_i^s]$ are available. The time-span while the efficiency is at least 1/2 is, of course, a phase of type 1 and not part of the delay phase. Obviously, the length of the second part of a delay phase is bounded by T_R and therefore the length of a delay phase is always bounded by $T_R + 2$.

Let d denote the number of delay phases in a schedule generated by the RRR Algorithm. We distinguish two cases:

1. $d = 1$: We have to show that the optimum solution needs at least time 2. This follows immediately from the fact that each delayed job must have a predecessor in the job system because otherwise it would have been scheduled earlier.

2. $d > 1$: This case will be proven by constructing a chain of jobs in the dependency graph with total execution time at least $2d$. From that we have $T_{\text{opt}} \geq 2d$ and together with $T_{\text{delay}} \leq d(T_R + 2)$ the claim follows.

The construction of this chain proceeds as follows: Starting with an arbitrary delayed job that is scheduled after delay phase d we observe that

there must be a small job that is ancestor of this delayed job and is available immediately after the delayed jobs of delay phase $d-1$ (i.e., without having a small job as direct predecessor that is itself scheduled after the delayed jobs of delay phase $d-1$) because otherwise this delayed job would have been scheduled earlier. We add such a small job at the front of the chain.

To augment the chain, we state the possibilities for the direct predecessor of a small job that is scheduled by the RRR Algorithm immediately after the delayed jobs of delay phase i:

Type 1: Delayed job of delay phase i or
big job that is successor of a delayed job of delay phase i,

Type 2: Small job collected during delay phase i,

Type 3: Small job running from the beginning of delay phase i.

This is due to the fact that the RRR Algorithm schedules all small jobs that are available by time t_i^s before the delayed jobs of delay phase i.

We continue the construction inductively according to these three possibilities. If there is a direct predecessor of Type 1 of the small job that is currently head of the list, we can repeat the initial construction step of the chain and add a delayed job and its small ancestor at the front of the chain. When there is no direct predecessor of Type 1 but a direct predecessor of Type 2, we add 2 more jobs at the front of the chain: the Type 2 job and a direct predecessor of this job that was running at the beginning of the delay phase during which this Type 2 job was collected. Finally, if there is only a direct predecessor of Type 3, we add this job at the front of the chain. The inductive construction stops as soon as the head of the chain is a small job that is scheduled before the delayed jobs of the first delay phase.

To complete the proof, we show that the total execution time of the jobs along this chain is at least $2d$. The construction of the chain starts with 2 jobs, a delayed job and its small ancestor. Since the minimum running time of any job is 1, these 2 jobs need at least 2 units of time for execution in any schedule. If the construction proceeds by adding a Type 1 job, the same argument applies. Continuing with a Type 2 job means that again 2 more jobs were added to the chain. If a Type 3 job is encountered, we know that this job must have execution time at least 2 because it is direct predecessor of a small job that is scheduled immediately after the delayed jobs of the

delay phase the Type 3 job belongs to. Thus, for each delay phase in the schedule generated by the RRR Algorithm, the above construction adds jobs with total execution time at least 2 to the chain. □

The assumption that $t_{\min}$ is known to the RRR Algorithm can be dropped by employing an adaptive waiting strategy without much loss in competitive performance. We describe this adaptive version separately in order to keep our presentation modular. The modifications of the RRR Algorithm are as follows (see also Figure 4.10): Since $t_{\min}$ is now unknown the RRR_ADAPTIVE Algorithm does not collect small jobs during the first delay phase. In all following delay phases (if any), the algorithm calculates $t^i_{\min}$, the minimum execution time of any finished job up to the start of delay phase i. The duration during which small jobs are collected is now limited by $2\,t^i_{\min}$ (and, of course, by $t_{\max}$ since the collection of jobs ends as soon as all scheduled jobs finish their execution).

Theorem 4.17 *The RRR_ADAPTIVE Algorithm is* $(T_R/2 + 5.5)$*-competitive for any job system* $(\mathcal{J}, \prec)$ *with* $\mathrm{RR}(\mathcal{J}) \leq T_R$.

Proof: With the notation of the proof of Theorem 4.16 we conclude analogously that the above theorem holds for $d = 1$. If $d > 1$, we have

$$T_{\text{delay}} \leq d \cdot t_{\max} + 2 \sum_{i=2}^{d} t^i_{\min}\,.$$

First, we show that $2\sum_{i=2}^{d} t^i_{\min} \leq 2\,T_{\text{opt}} - 2\,t_{\min}$. To see this, we observe that after delay phase i, $1 \leq i \leq d$, at least one delayed job has to be scheduled. Let $t^{d+1}_{\min} := t_{\min}$. The running time of such a delayed job is at least $t^{i+1}_{\min}$, since this job is executed before the start of delay phase $i+1$ (if $i < d$). Even in an optimum schedule all delayed jobs must be scheduled sequentially because they require more than half of the available processors for execution. Therefore:

$$2\,T_{\text{opt}} \geq 2 \sum_{i=2}^{d+1} t^i_{\min} = 2 \sum_{i=2}^{d} t^i_{\min} + 2\,t_{\min}\,. \tag{4.2}$$

As in the proof of Theorem 4.16 we can construct a chain of jobs in the dependency graph with total execution time at least $(2d-1)t_{\min}$. The only difference in the construction is that there is no collection of small jobs during the first delay phase and therefore a Type 3 job might only

```
Algorithm RRR_ADAPTIVE
begin
    i := 0;   // i counts the number of delay phases
    while L_1 ≠ ∅ do
        schedule a big job exclusively;
    while not all jobs are finished do
        begin
        while L_2 ≠ ∅ do
            schedule small jobs greedily;
        if L_1 ≠ ∅ then
            if a big job can be scheduled then
                do it;
            else
                if α ≥ 1/2 then
                    wait for a scheduled job to finish;
                else   // start of a delay phase
                    if i > 0 then
                        i := i + 1;
                        t^i_min := current minimum execution time;
                        collect small jobs that become available
                        for time ≤ 2 t^i_min;
                        schedule those jobs greedily
                        and then wait for all scheduled jobs to finish;
                    else
                        i := i + 1;
                        wait for all scheduled jobs to finish;
                    fi;
                    while L_1 ≠ ∅ do
                        schedule a big job exclusively;
                fi;
            fi;
        else
            wait for next available job;
        fi;
        end;
end.
```

Figure 4.10: The RRR_ADAPTIVE Algorithm

run for time $t_{\min}$ in this delay phase. This yields another lower bound on the optimum schedule length:

$$T_{\text{opt}} \geq (2d-1)t_{\min}\,. \tag{4.3}$$

From (4.2) and (4.3) we conclude:

$$\begin{aligned} T_{\text{delay}} &\leq d \cdot t_{\max} + 2\sum_{i=2}^{d} t_{\min}^{i} \\ &\leq d \cdot t_{\max} + 2\,T_{\text{opt}} - 2\,t_{\min} \\ &\leq \frac{(d-1/2)T_R}{2d-1}T_{\text{opt}} + \frac{t_{\max}}{2} + 2\,T_{\text{opt}} \\ &\leq \left(\frac{T_R}{2} + \frac{5}{2}\right) T_{\text{opt}}\,. \end{aligned}$$

If the number of delay phases of a schedule is less than $(T_R+1)/2$, we can derive a better upper bound:

$$T_{\text{delay}} \leq (d+2)\,T_{\text{opt}}\,.$$

However, this bound is useful for a posteriori analysis only, since the number of delay phases can be arbitrarily large. Since the total schedule time that is spent in phases of type 1 and 2 (cf. proof of Theorem 4.16) is bounded by $3\,T_{\text{opt}}$, the proof is complete. □

Clearly, both algorithms can easily compute the runtime ratio $RR(\mathcal{J})$ for any scheduled job system $\mathcal{J}$. From this, we can bound the actual performance for the generated schedules:

$$\begin{aligned} T_{\text{RRR}} &\leq (RR(\mathcal{J})/2+4)\,T_{\text{opt}}\,, \\ T_{\text{RRR\_ADAPTIVE}} &\leq (RR(\mathcal{J})/2+5.5)\,T_{\text{opt}}\,. \end{aligned}$$

For practical purposes it is desirable to have tools that allow to control the performance of a scheduler in addition to worst-case guarantees such as the competitive ratio. Let T_{big} be the sum of the execution times of all big jobs in $\mathcal{J}$, and let W_{total} denote the total work of all jobs. Then we have the following lower bound on the length of an optimum schedule:

$$T_{\text{opt}} \geq \max\left\{W_{\text{total}}/N, T_{\max}, T_{\text{big}}\right\}.$$

Again, our on-line algorithms can compute W_{total} and T_{big} during the scheduling process. Assuming that the on-line scheduler has knowledge

of the predecessor/successor relationships (which usually will be the case after all jobs have been scheduled), $T_{\max}$ can be computed by searching a longest path in the dependency graph. The quotient of the length of the on-line schedule and the above lower bound is then an upper bound for the performance of our on-line schedulers.

4.4.3 Characteristics of the RRR Algorithm

Both algorithms presented in the previous section have two characteristics that we study in more detail in this section:

- **Waiting**. If not all processors for a big job to be scheduled are available, the algorithms wait until all currently executing jobs terminate.
- **Collecting**. Although the algorithms decide to wait they first collect and then jointly execute small jobs that become available during a specified time-span after a delay phase has started.

We show that waiting is (in a restricted sense to be made precise) necessary for deterministic on-line schedulers for any interconnection topology to achieve a competitive ratio that is close to the general lower bound of Section 4.4.1. Furthermore, we argue that collecting is essential for the RRR Algorithm.

To this aim, we say that an on-line scheduler is *greedy* if some available job is scheduled whenever this is possible. Clearly, the RRR and RRR_ADAPTIVE algorithms are not greedy in the sense of this definition. The next theorem shows that a greedy algorithm cannot reach the lower bound:

Theorem 4.18 *The competitive ratio of a deterministic greedy on-line scheduler for job systems with runtime ratio no more than T_R on an arbitrary network is unbounded.*

Proof: As in the proof of the general lower bound we construct a job system that consists of $N \geq 5$ layers (see Figure 4.11). Each layer has a task that is predecessor of a parallel job of size N on the same layer. Furthermore, there is a chain of tasks of length $k \geq 2$ and k additional tasks numbered from 1 to k. Task $j \geq 2$ is successor of the $(j-1)$'th task of the chain. Consequently, each layer starts with 3 independent tasks. The

parallel job is predecessor of these 3 task on the next layer (if any). The task on each layer scheduled first by the on-line scheduler is chosen as Task 1 with running time T_R by the adversary. The next one will be the head of the chain with running time $T_R - \varepsilon$ with $\varepsilon > 0$. The remaining task runs for 1 unit of time.

We normalize the beginning of each partial schedule to time 0. Since the algorithm is greedy the parallel job becomes available at time 1 but cannot be scheduled. At time $T_R - \varepsilon$ the first task of the chain finishes and two more tasks become available. These tasks are immediately scheduled since there are enough processors available. Again, the task from the chain is assigned running time $T_R - \varepsilon$ whereas Task 2 runs for time T_R. This pattern is repeated until all tasks of this layer finish their execution. Finally, the greedy on-line scheduler is able to schedule the job of size N. The length of each partial schedule is thus at least kT_R for ε sufficiently small.

The optimum solution schedules the UET jobs first followed by the N chains in parallel. The remaining kN tasks are scheduled in parallel at the end of the optimum schedule. It is easy to see that the length of the optimum solution is less than $2(N + kT_R)$.

From this we can instantly conclude that the competitive ratio of a deterministic greedy on-line scheduler cannot be less than $kT_R/2$. Choosing k as a function of N such that $k(N) = \omega(1)$ but $k = o(N)$ yields the claim of the theorem. □

After having shown the importance of waiting we now study the influence of collecting. Since non-greedy on-line schedulers may employ very different waiting strategies we only consider the RRR Algorithm in the complete model. We modify the RRR Algorithm by omitting the collection of small jobs during the first 2 units of time of a delay phase. Furthermore, we drop the restriction that big jobs have to be scheduled exclusively in certain situations. We refer to this modified algorithm as RRR_SIMPLE.

We show now that the performance of the RRR_SIMPLE Algorithm is substantially worse than the performance of the RRR Algorithm. The adversary again uses a job system that consists of $N \geq 5$ layers. Each layer has 5 tasks. A job of size $N - 2$ requiring one unit of time for execution is added to each layer but the first one. This job depends on one task from the preceding layer. The first 3 tasks of each layer scheduled by the RRR_SIMPLE Algorithm are assigned running time T_R. The fourth one is chosen as the predecessor of the parallel job and runs for 1 unit of

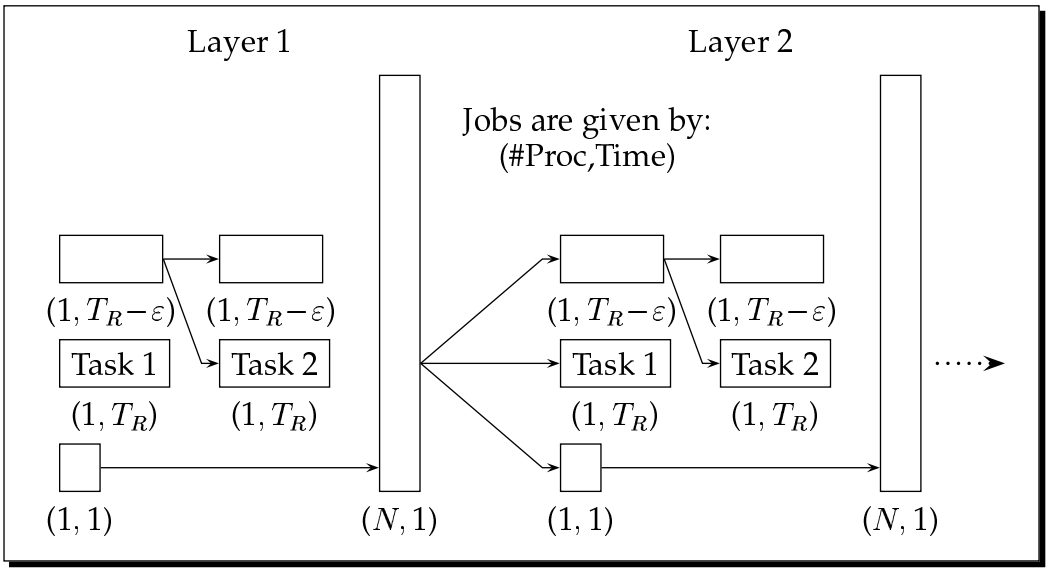

Figure 4.11: Bad job system for deterministic greedy on-line schedulers

time. Finally, the fifth task scheduled needs time $1 + \varepsilon$, $\varepsilon > 0$, for execution and is predecessor of all tasks on the next layer (if any).

This timing makes sure that the job of size $N - 2$ becomes available to the on-line scheduler before the tasks of the same layer. Thus, the RRR_SIMPLE Algorithm enters a delay phase and this prevents the available tasks from execution until all tasks from the preceding layer have finished. But even then only 2 tasks can be scheduled in parallel with the job of size $N - 2$. Therefore, it is not hard to see that for ε sufficiently small the competitive ratio of the RRR_SIMPLE Algorithm cannot be smaller than $T_R + 1$.

4.4.4 Other Topologies

The RRR and RRR_ADAPTIVE algorithms rely heavily on the complete model assumption which avoids fragmentation of the idle processors. On other topologies like hypercubes and arrays fragmentation is likely to occur frequently when jobs with unknown execution times are scheduled in parallel.

SGALL [Sga94] has shown that no deterministic or randomized on-line algorithm can be $o(\log N / \log \log N)$-competitive for N processors for scheduling job systems with arbitrary running times on a 1-dimensional

array even when virtualization is allowed. The proof can be modified for job systems $(\mathcal{J}, \prec)$ with $RR(\mathcal{J}) \leq T_R$ to show that no deterministic on-line scheduler for 1-dimensional arrays can have competitive ratio better than T_R.

To obtain upper bounds on the competitive ratio for arbitrary network topologies we use the LEVEL Algorithm. It is understood that the job packings computed by the packing subroutine are executed sequentially. The following theorem is a straightforward generalization of Theorem 4.14:

Theorem 4.19 *Let* PACK *be a packing algorithm for a given network T, let $(\mathcal{J}, \prec)$ be a job system with job types suitable for T and $RR(\mathcal{J}) < T_R$, $\mathcal{J}' \subset \mathcal{J}$ an arbitrary subset of independent jobs, and $w : \mathcal{J} \to \mathbb{R}^+$ a weighting function. If* $\text{PACK}(\mathcal{J}') \leq \sum_{j \in \mathcal{J}'} w(j) + c$ *for a constant $c \in \mathbb{N}_0$, and if $\sum_{j \in \mathcal{J}} w(j) \leq a \cdot T_{\text{opt}}$ for a constant $a \in \mathbb{R}^+$, it holds that*

$$T_{\text{LEVEL(PACK)}} \leq (a + c)T_R \cdot T_{\text{opt}}.$$

Proof: Let $\mathcal{J}_1$ denote the set of jobs when all running times of the jobs in $\mathcal{J}$ are reduced to 1, and let S and S_1 denote the schedules generated by the LEVEL algorithm for both job systems. Clearly, $|S| \leq T_R\,|S_1|$. Since $T_{\text{opt}}(\mathcal{J})$ cannot be smaller than $T_{\text{opt}}(\mathcal{J}_1)$ the claim of the theorem follows. □

For hypercubes the above theorem yields that the LEVEL(PACK_HC) Algorithm is $(2T_R)$-competitive. This result can be improved if we drop the assumption that the individual packings are executed sequentially. Rather, we use a modification of PACK_HC to generate a partial schedule for the whole level. In the beginning the jobs of a level are sorted by size in non-increasing order as before. Then, jobs are scheduled from the head of the sorted list to normal subcubes while there are idle processors or no more jobs left. Whenever a job finishes, PACK_HC is able to reuse all the processors if there are enough unscheduled jobs. Therefore, the efficiency of a partial schedule is 1 except possibly for time at most T_R at the end of the partial schedule. With Lemma 4.1 we have that this algorithm is $(T_R + 1)$-competitive.

Chapter 5

Load Balancing for Problems with Good Bisectors

In this chapter we study load balancing for a very general class of problems. The only assumption we make is that all problems in the class have a certain bisection property. Such classes of problems arise, for example, in the context of distributed hierarchical finite element simulations. We show that a satisfactory load balancing quality can be achieved even in the worst case using a quite simple strategy. The tight worst-case upper bound on the maximum load generated by this algorithm depends only on the particular bisection property of the class of problems under consideration. Although our approach may appear sequential in nature at first sight, we provide efficient parallel algorithms for our load balancing model in the next chapter. Furthermore, we show how our general results can be applied to numerical applications in several ways.

The remainder of the chapter is structured as follows. In Section 5.1 we present and analyze a very general algorithm that computes a good load distribution for classes of problems with α-bisectors. Section 5.2 briefly explains distributed finite element simulations with recursive substructuring. Two strategies for applying the algorithm from Section 5.1 to these applications are discussed. Section 5.3 shows that certain weighted trees, which model the load of applications in numerical simulations like the one discussed in Section 5.2, have 1/4-bisectors. This implies that the maximum load generated by our novel load balancing algorithm for these applications exceeds the ideal load (uniform distribution) by at most 9/4. In Section 5.4 we classify the new algorithm according to the scheme presented in Chapter 3.

5.1 Using Bisectors for Load Balancing

In many applications a computational problem cannot be divided into many small problems as required for an efficient parallel solution directly. Instead, a strategy similar to *divide and conquer* is used repeatedly to divide problems into smaller subproblems. We refer to the division of a problem into two smaller subproblems as *bisection*. Assuming a weight function w that measures the resource demand, a problem p cannot always be bisected into two subproblems p_1 and p_2 of equal weight $\mathsf{w}(p)/2$. For many classes of problems, however, there is a bisection method that guarantees that the weights of the two resulting subproblems do not differ too much. The following definition captures this concept more precisely.

Definition 5.1 *Let $0 < \alpha \leq \frac{1}{2}$. A class $\mathcal{P}$ of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$* has α-bisectors *if every problem $p \in \mathcal{P}$ can be efficiently divided into two problems $p_1 \in \mathcal{P}$ and $p_2 \in \mathcal{P}$ with $\mathsf{w}(p_1) + \mathsf{w}(p_2) = \mathsf{w}(p)$ and $\mathsf{w}(p_1), \mathsf{w}(p_2) \in [\alpha\mathsf{w}(p), (1-\alpha)\mathsf{w}(p)]$.*

For a class of problems $\mathcal{P}$ that has α-bisectors we refer to α as the *bisection parameter* of $\mathcal{P}$. When a problem $p \in \mathcal{P}$ is bisected into p_1 and p_2 such that $\mathsf{w}(p_1) \leq \mathsf{w}(p_2)$ we let $\hat{\alpha} := \mathsf{w}(p_1)/\mathsf{w}(p)$ and call $\hat{\alpha}$ the *actual bisection parameter* of that bisection.

The above definition characterizes classes of problems that have α-bisectors in a very abstract way. In a particular application, problems might correspond to subdomains of a numerical computation, to parts of a simulated system, to parts of the search space for an optimization problem (cf. [KZ93]), or to program execution dags.

Note that this definition requires, for sake of simplicity, that all problems in $\mathcal{P}$ can be bisected, whereas in practice this is not the case for problems whose weight is below a certain threshold. We assume, however, that the problem to be divided among the processors is large enough to allow further bisections until the number of subproblems is equal to the number of processors. This is a reasonable assumption for most relevant parallel applications.

A definition very similar to ours (using the term α-splitting) is used by KUMAR, GRAMA, and RAO [KV87, KGV94] [KGGK94, pp. 315–318] under the assumption that the weight of a problem is unknown to the load balancing algorithm.

5.1.1 Tight Analysis of Algorithm HF

```
Algorithm HF(p,N)
begin
    P := {p};
    while |P| < N do
        begin
        q := a problem in P with maximum weight;
        bisect q into q1 and q2;
        P := (P ∪ {q1, q2}) \ {q};
        end;
    return P;
end.
```

Figure 5.1: Algorithm HF (Heaviest Problem First)

Figure 5.1 shows Algorithm HF, which receives a problem p and a number N of processors as input and divides p into N subproblems by repeated application of α-bisectors to the heaviest remaining subproblem. Using a priority queue for the subproblems, we can implement Algorithm HF to run in time $\Theta(N \log N)$ provided that a bisection of a problem takes constant time. Subsequently, the subproblems have to be sent to the processors adding communication costs depending on the topology of the network.

A perfectly balanced load distribution on N processors would be achieved if a problem p of weight $\mathsf{w}(p)$ was divided into N subproblems of weight exactly $\mathsf{w}(p)/N$ each. The following theorem gives a worst-case bound on the ratio between the maximum weight among the N subproblems produced by Algorithm HF and this *ideal weight* $\mathsf{w}(p)/N$. The case that more than N subproblems are generated is discussed in Section 5.1.3.

Theorem 5.2 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ that has α-bisectors. Given a problem $p \in \mathcal{P}$ and a positive integer N, Algorithm HF uses $N - 1$ bisections to partition p into N subproblems $p_1, \ldots, p_N$ such that*

$$\max_{1 \le i \le N} \mathsf{w}(p_i) \le \frac{\mathsf{w}(p)}{N} \cdot r_\alpha ,$$

where

$$r_\alpha = \left\lfloor \frac{1}{\alpha} \right\rfloor \cdot (1-\alpha)^{\left\lfloor \frac{1}{\alpha} \right\rfloor - 2} .$$

Proof: It is obvious that Algorithm HF uses $N-1$ bisections to partition p into N subproblems. In the following we show that the stated inequality regarding the maximum weight among these subproblems holds.

We introduce the *bisection tree* T_p^N to represent the run of the algorithm on input p and N. The root of T_p^N is the problem p. If the algorithm bisects a problem q into q_1 and q_2, nodes q_1 and q_2 are added to T_p^N as children of node q. In the end, T_p^N has N leaves, which correspond to the N subproblems computed by the algorithm, and all problems that were bisected by the algorithm appear as internal nodes with exactly two children. We observe that the run of any bisection-based load balancing algorithm corresponds to a certain bisection tree. Figure 5.2 gives an example of a bisection tree for a problem of weight 44 from a class of problems with $\frac{1}{7}$-bisectors. We follow the convention of drawing the node with greater weight among two children of the same parent as the left child of that parent.

The following properties hold for bisection trees arising from classes of problems with α-bisectors. Let the leaves of the tree be $p_1, \ldots, p_N$ and let $m := \max_{1 \le i \le N} \mathsf{w}(p_i)$.

(a) $\mathsf{w}(q) \ge m$ for all internal nodes q

(b) $\mathsf{w}(q) \ge \frac{1}{1-\alpha}\mathsf{w}(q')$ if q' is a child of q

(a) holds because the algorithm always bisects a subproblem of maximum weight; since one of the p_i has weight m, there must have been at least one subproblem of weight $\ge m$ during the whole run of the algorithm, and thus the algorithm never bisected a problem of weight $< m$. (b) follows directly from $\mathsf{w}(q') \le (1-\alpha)\mathsf{w}(q)$, which holds because the algorithm uses α-bisectors.

Now remove from the bisection tree all internal nodes which are not parent of a leaf. This partitions the bisection tree into a number of disjoint branches, whose shape can be that of a *leaf-branch* (one of the internal nodes of the branch has two leaf children) or that of an *internal branch* (all the internal nodes of the branch have exactly one leaf child). The branches obtained from the example tree of Figure 5.2 are shown in Figure 5.3. Our goal is to derive a lower bound on the average weight of the leaves in each branch.

Consider a leaf-branch with k internal nodes, $k \ge 1$. Denote its internal nodes by $v_1, v_2, \ldots, v_k$ such that v_{i+1} is the parent of v_i for $1 \le i \le k-1$.

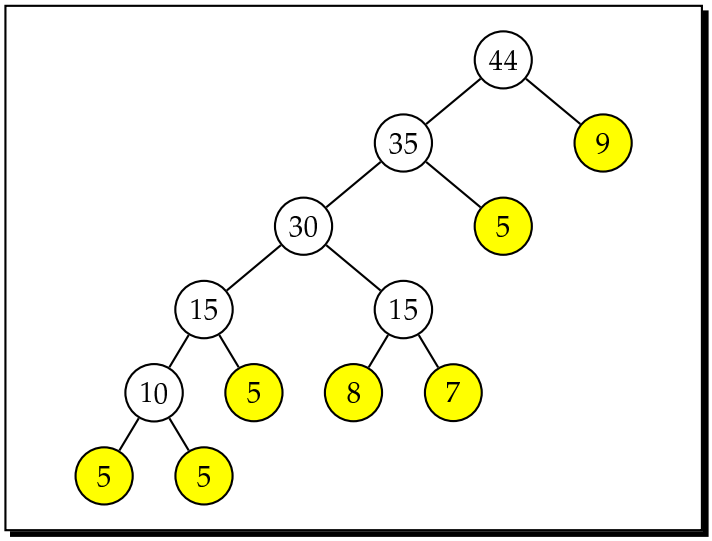

Figure 5.2: Example of a bisection tree

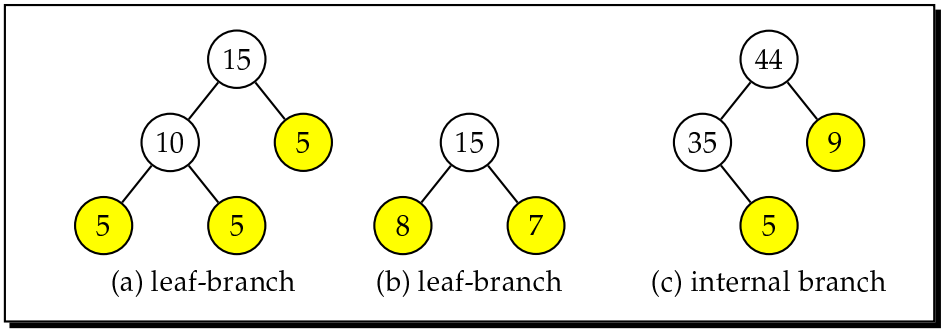

Figure 5.3: Branches obtained from the example tree

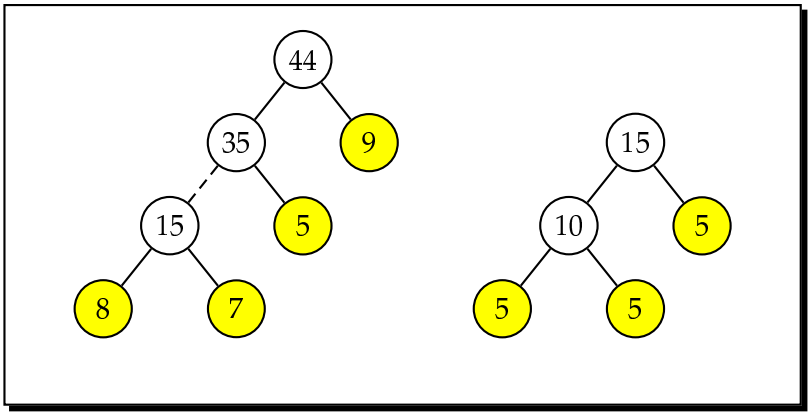

Figure 5.4: Composed leaf-branches obtained from the example tree

Furthermore, let c_i denote the leaf child of v_i for $2 \leq i \leq k$, and let c_0 and c_1 denote the leaf children of v_1. As (a) implies $\mathsf{w}(v_1) \geq m$, we have by (b) $\mathsf{w}(v_i) \geq \left(\frac{1}{1-\alpha}\right)^{i-1} m$ for $1 \leq i \leq k$ and $\mathsf{w}(c_i) \geq \alpha \left(\frac{1}{1-\alpha}\right)^{i-1} m$ for $1 \leq i \leq k$. The average weight of the leaves $c_0, \ldots, c_k$ can now be bounded from below as follows:

$$\begin{aligned}
\frac{1}{k+1}\sum_{i=0}^{k} \mathsf{w}(c_i) &\geq \frac{1}{k+1}\left(m + \sum_{i=2}^{k} \alpha \left(\frac{1}{1-\alpha}\right)^{i-1} m\right) \\
&= \frac{1}{k+1}\left(m(1-\alpha) + \alpha m \sum_{i=0}^{k-1} \frac{1}{(1-\alpha)^i}\right) \\
&= \frac{1}{k+1}\left(m(1-\alpha) + m(\alpha-1)\left(1 - \frac{1}{(1-\alpha)^k}\right)\right) \\
&= \frac{m}{(k+1)(1-\alpha)^{k-1}}\,.
\end{aligned}$$

If there are internal branches, we do not deal with them separately but instead attach them to leaf-branches. For example, one can consider the leaf-branches one by one and attach to each leaf-branch all internal branches that intersect the path from the leaf-branch to the root of the bisection tree and that have not been attached to a different leaf-branch before. Here, attaching an internal branch to a leaf-branch means making the root of the leaf-branch a child of the bottom-most internal node of the internal branch, resulting in a new leaf-branch. We call the leaf-branches obtained by attaching zero or more internal branches to an original leaf-branch *composed leaf-branches*. Observe that conditions (a) and (b) are satisfied for these composed leaf-branches as well. Hence, the lower bound above also pertains to the average weight of the leaves in such a composed leaf-branch. The bisection tree from Figure 5.2 contained two leaf-branches and one internal branch as illustrated in Figure 5.3. Attaching the internal branch to one of the leaf-branches gives the composed leaf-branches shown in Figure 5.4.

As every leaf of the bisection tree appears in exactly one composed leaf-branch, we conclude that $\min_{k\in\mathbb{N}} \frac{m}{(k+1)(1-\alpha)^{k-1}}$ is a lower bound on the average weight of all leaves in the bisection tree. Therefore, we obtain

$$\mathsf{w}(p) = \sum_{i=1}^{N} \mathsf{w}(p_i) \geq Nm \min_{k\in\mathbb{N}} \frac{1}{(k+1)(1-\alpha)^{k-1}}\,. \tag{5.1}$$

Besides, we observe that

$$\min_{k\in\mathbb{N}} \frac{1}{(k+1)(1-\alpha)^{k-1}} = \frac{1}{\max_{k\in\mathbb{N}}\left((k+1)(1-\alpha)^{k-1}\right)}, \tag{5.2}$$

and we claim that $(k+1)(1-\alpha)^{k-1}$ as a function of $k \in \mathbb{N}$ is maximized for $\hat{k} = \lfloor 1/\alpha \rfloor - 1$. To see this, let $f(k) = (k+1)(1-\alpha)^{k-1}$ and consider the ratio $f(k)/f(k-1) = (1-\alpha)(k+1)/k$. We obtain:

$$\frac{f(k)}{f(k-1)} = \begin{cases} > 1 & \text{for } k < \frac{1}{\alpha} - 1 \\ = 1 & \text{for } k = \frac{1}{\alpha} - 1 \\ < 1 & \text{for } k > \frac{1}{\alpha} - 1 \end{cases}.$$

For a fixed value of α, $f(k)$ is monotone increasing from $k = 1$ to $k = \lfloor 1/\alpha \rfloor - 1$ and monotone decreasing for larger values of k. If $1/\alpha$ is not an integer greater than 2, $f(k)$ is maximum only for $k = \lfloor 1/\alpha \rfloor - 1$. If $1/\alpha$ is an integer greater than 2, $f(k)$ is maximum for $k = \lfloor 1/\alpha \rfloor - 1$ and for $k = \lfloor 1/\alpha \rfloor - 2$. In any case we have $\max_{k\in\mathbb{N}}\left((k+1)(1-\alpha)^{k-1}\right) = f(\lfloor 1/\alpha \rfloor - 1) = r_\alpha$, and the theorem follows with (5.1) and (5.2). □

For some values of α, Table 5.1 gives worst-case bounds on the ratio between $\max_{1\le i\le N} \mathsf{w}(p_i)$ and $\frac{\mathsf{w}(p)}{N}$ as well as a value of k for which $(k+1)(1-\alpha)^{k-1}$ is maximized. These bounds show that the worst-case deviation from the ideal load distribution, in which $\mathsf{w}(p_i) = \frac{\mathsf{w}(p)}{N}$ for all $1 \le i \le N$, is bounded by a small constant for a wide range of α. We observe that r_α is equal to 2 for $\alpha \ge 1/3$, below 3 for $\alpha \ge 1 - 1/\sqrt[4]{2} \approx 0.159$, and below 10 for $\alpha \ge 0.04$. Hence, Algorithm HF achieves provably good load balancing for classes of problems with α-bisectors for a surprisingly large range of α. Note that in many cases an ideal load distribution cannot be achieved by any algorithm.

Corollary 5.3 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ that has α-bisectors. Given a problem $p \in \mathcal{P}$ and a positive integer N, Algorithm HF uses $N-1$ bisections to partition p into N subproblems $p_1, \ldots, p_N$ such that*

$$\max_{1\le i\le N} \mathsf{w}(p_i) \le \frac{\mathsf{w}(p)}{N} \cdot \frac{1}{\mathrm{e}(1-\alpha)^2 \ln \frac{1}{1-\alpha}}.$$

Proof: In the proof of Theorem 5.2 it was shown that

$$\max_{1\le i\le N} \mathsf{w}(p_i) \le \frac{\mathsf{w}(p)}{N} \cdot \max_{k\in\mathbb{N}}((k+1)(1-\alpha)^{k-1}). \tag{5.3}$$

Table 5.1: Worst-case ratio of Algorithm HF for different values of α

α	k	ratio
0.02	49	18.96
0.04	24	9.78
0.06	15	6.73
0.08	11	5.21
0.10	8	4.30
0.12	7	3.72
0.14	6	3.29
0.16	5	2.99
0.18	4	2.76
0.20	3	2.56

α	k	ratio
0.21	3	2.50
0.22	3	2.43
0.23	3	2.37
0.24	3	2.31
0.25	2	2.25
0.26	2	2.22
0.27	2	2.19
0.28	2	2.16
0.29	2	2.13
0.30	2	2.10

α	k	ratio
0.31	2	2.07
0.32	2	2.04
0.325	2	2.025
0.33	2	2.01
0.331	2	2.007
0.332	2	2.004
0.333	2	2.001
0.334	1	2.00
0.40	1	2.00
0.50	1	2.00

Observe that the term maximized on the right hand side of this inequality is a differentiable function of k. Therefore, we define $f : \mathbb{R}^+ \to \mathbb{R}^+$ by $f(k) = (k+1)(1-\alpha)^{k-1}$. The derivative of f is:

$$f'(k) = (1-\alpha)^{k-1} \cdot ((k+1)\ln(1-\alpha) + 1)\,.$$

The derivative is zero for

$$(k+1)\ln(1-\alpha) = -1\,,$$

which is the case only for $\hat{k} = \frac{-1}{\ln(1-\alpha)} - 1$.

Substitution yields $f(\hat{k}) = \left(\mathrm{e}(1-\alpha)^2 \ln \frac{1}{1-\alpha}\right)^{-1}$, and this is the global maximum of f. Hence,

$$\max_{k\in\mathbb{N}}((k+1)(1-\alpha)^{k-1}) \leq \frac{1}{\mathrm{e}(1-\alpha)^2 \ln \frac{1}{1-\alpha}}\,,$$

and the corollary follows directly from inequality (5.3). □

In Figure 5.5 the worst-case bound on the ratio between $\max_{1\leq i\leq N} \mathsf{w}(p_i)$ and $\mathsf{w}(p)/N$ from Theorem 5.2 as well as the continuous approximation of this bound $\left(\mathrm{e}(1-\alpha)^2 \ln \frac{1}{1-\alpha}\right)^{-1}$ from Corollary 5.3 are plotted for $0.08 \leq \alpha \leq 0.5$. It turns out that the continuous approximation of the bound matches the discrete bound (cf. Table 5.1) almost exactly for $\alpha \leq 0.3$. To complete this part of our analysis, we observe that the exact upper bound on the ratio between $\max_{1\leq i\leq N} \mathsf{w}(p_i)$ and $\mathsf{w}(p)/N$ is 2 for $\alpha \geq 1/3$.

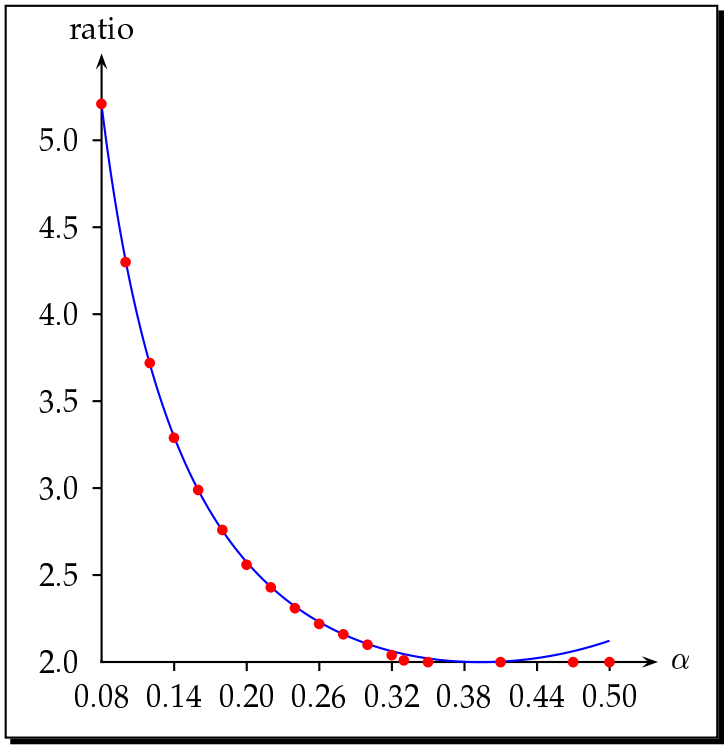

Figure 5.5: Plot of discrete (dotted) and continuous worst-case bounds

Now we give a lower bound on the worst-case ratio between the maximum weight subproblem generated by Algorithm HF and the ideal value given by a uniform partition. This will show that the upper bound from Theorem 5.2 is tight.

Theorem 5.4 *For each* $0 < \alpha \leq \frac{1}{2}$ *there exists a class of problems* $\mathcal{Q}^\alpha$ *that has* α*-bisectors and contains a family of problems* $(q^l)_{l\in\mathbb{N}}$ *such that*

$$\lim_{l\to\infty} \frac{\max_{1\leq i\leq N_l} \mathsf{w}(q_i^l)}{\frac{\mathsf{w}(q^l)}{N_l}} = r_\alpha \,,$$

where $N_l = \lfloor 1/\alpha \rfloor \cdot 2^l - 1$ *and* $q_1^l, q_2^l, \ldots, q_{N_l}^l$ *are the subproblems generated by Algorithm HF on input* q^l *and* N_l*.*

Proof: Let $\mathcal{Q}^\alpha$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ and the following properties:

(a) each $q \in \mathcal{Q}^\alpha$ with $\mathsf{w}(q) > 1$ can only be partitioned into 2 subproblems of weight $\mathsf{w}(q)/2$ each

(b) each $q \in \mathcal{Q}^\alpha$ with $\mathsf{w}(q) \leq 1$ can only be partitioned into 2 subproblems of weight $(1-\alpha)\mathsf{w}(q)$ and $\alpha\mathsf{w}(q)$

(c) for every $l \in \mathbb{N}$ there is a problem $q^l \in \mathcal{Q}^\alpha$ of weight $\mathsf{w}(q^l) = 2^l$

Clearly, $\mathcal{Q}^\alpha$ has α-bisectors according to Definition 5.1.

Let $k = \lfloor 1/\alpha \rfloor - 2$. For a given $l \in \mathbb{N}$, choose a problem $q^l \in \mathcal{Q}^\alpha$ of weight 2^l and let $N_l = (k+2)2^l - 1$. On input q^l and N_l, Algorithm HF proceeds as follows. After the first $2^l - 1$ bisections, there are 2^l subproblems of weight 1 each. We assign level 0 to these problems and call them active. As the weight of each active problem is ≤ 1, the next 2^l bisections performed by the algorithm subdivide all active problems on level 0 and generate subproblems of weight $1-\alpha$ and α, which are assigned level 1. Now the 2^l problems on level 1 with weight $1-\alpha$ become active. This process is repeated such that in phase i, $i \geq 0$, the algorithm subdivides all 2^l problems of weight $(1-\alpha)^i$ on level i. At the end of phase i there are $(i+2)2^l$ subproblems altogether. The subdivision process is finished when exactly one active subproblem on level k of weight $(1-\alpha)^k$ remains.

To ensure that the algorithm indeed subdivides the active problems on level i in phase i for all $0 \leq i \leq k$ and not the heaviest inactive problem, which has weight α, it is required that $(1-\alpha)^k \geq \alpha$. This is obvious for $k = 0, 1$. For $k \geq 2$, recall that the series $(1-1/(k+1))^k$ is strictly decreasing. It converges to e^{-1} from above. Note that $k = \lfloor 1/\alpha \rfloor - 2$ implies $\alpha \leq 1/(k+2) \leq 1/4$. Hence,

$$(1-\alpha)^k \geq \left(1 - \frac{1}{k+2}\right)^k \geq \left(1 - \frac{1}{k+1}\right)^k \geq \frac{1}{\mathrm{e}} \geq \frac{1}{4} \geq \alpha\,.$$

In the end, Algorithm HF has generated $(k+2)2^l - 1$ subproblems and a maximum weight of $(1-\alpha)^k$. Thus,

$$\max_{1 \leq i \leq N_l} \mathsf{w}(q_i^l) = \frac{\mathsf{w}(q^l)}{N_l} \cdot (1-\alpha)^k (k+2-2^{-l})\,,$$

and the assertion of the theorem follows by substituting $k = \lfloor 1/\alpha \rfloor - 2$ and taking into account $\lim_{l\to\infty} 2^{-l} = 0$. □

We can use the same construction to show that Algorithm HF is *optimal* from a worst-case point of view.

Corollary 5.5 *Let A be a deterministic or randomized bisection-based load balancing algorithm for classes of problems that have α-bisectors. Assume furthermore that A is restricted to perform at most $N-1$ bisections. Then, the worst-case upper bound on the (expected) ratio between the maximum weight subproblem generated by A and the ideal weight cannot be smaller than r_α.*

Proof: For a problem p in a class of problems that has α-bisectors the *complete bisection tree* $\overline{T_p}$ is defined as the complete binary tree of infinite height with root p that results if each subproblem is bisected recursively.

With the notations of the preceding theorem it is a straightforward calculation to show that for $l \in \mathbb{N}$ $\overline{T_{q^l}}$ contains exactly N_l nodes of weight at least $(1-\alpha)^k$. Thus, after $N_l - 1$ bisections there will be a subproblem of weight at least $(1-\alpha)^k$, no matter how A works. We conclude from the preceding theorem that the asymptotic lower bound r_α also holds for A. □

5.1.2 A Better Bound for Small N

Note that the bound of Theorem 5.2 is independent of N, the number of desired subproblems. Although we have shown that this bound is tight asymptotically, it is possible to obtain a better bound if N is sufficiently small. Again, we will show that this improved bound is tight. To establish this result, we need the following

Lemma 5.6 *Let $\alpha \leq 1/5$, $2 \leq k \leq 1/\alpha$. Then, with the assumptions of Theorem 5.2, for any leaf-branch of a bisection tree with k leaves $p_1, p_2, \ldots, p_k$ and root p:*

$$\max_{1 \leq i \leq k} \mathsf{w}(p_i) \leq \mathsf{w}(p)(1-\alpha)^{k-1}.$$

Proof: If all bisections are exact α-bisections we conclude that the maximum weight subproblem generated by Algorithm HF has weight $\mathsf{w}(p)(1-\alpha)^{k-1}$ since $\alpha < (1-\alpha)^{k-1}$ for $k \leq 1/\alpha$ and $\alpha \leq 1/5$. Clearly, the upper bound remains valid if the maximum weight subproblem is the leftmost leaf of the leaf-branch.

Therefore, we consider the case that the maximum weight subproblem does not result from the last bisection step. Let $m := \max_{1 \leq i \leq k} \mathsf{w}(p_i)$. The combined weight of the maximum weight leaf and the 2 leaves generated in the last bisection step is at least $2m$. The total weight of the remaining

leaves can be bounded from below by

$$m\left(\left(\frac{1}{1-\alpha}\right)^{k-3}-1\right)$$

using the same argument as in the proof of Theorem 5.2. Thus,

$$m \leq \frac{\mathsf{w}(p)}{\left(\frac{1}{1-\alpha}\right)^{k-3}+1},$$

and it remains to show that the right hand side of this inequality is no more than $\mathsf{w}(p)(1-\alpha)^{k-1}$. But since $\alpha \leq 1/5$ and $k \leq 1/\alpha$ we have

$$\begin{aligned} 1 &\leq 0.64 + e^{-1} \\ &\leq (1-\alpha)^2 + (1-\alpha)^{k-1} \\ &\leq \left(\left(\frac{1}{1-\alpha}\right)^{k-3}+1\right)(1-\alpha)^{k-1}. \end{aligned}$$

□

Theorem 5.7 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ that has α-bisectors, and assume $\alpha \leq 1/5$. Given a problem $p \in \mathcal{P}$ and a positive integer $N \leq 1/\alpha$, Algorithm HF uses $N-1$ bisections to partition p into N subproblems $p_1, \ldots, p_N$ such that*

$$\max_{1\leq i\leq N} \mathsf{w}(p_i) \leq \mathsf{w}(p)(1-\alpha)^{N-1}.$$

Proof: We will show that the worst-case bisection tree is a single leaf-branch if the assumptions of the theorem hold. The claim then follows immediately from the previous lemma.

Let us assume that the bisection tree generated by the run of Algorithm HF is not a single leaf-branch. Consequently, the bisection tree has internal nodes which are not parent of a leaf. We call these nodes *cut-nodes*. Let $m := \max_{1\leq i\leq N} \mathsf{w}(p_i)$. Observe that a cut-node has weight at least $2m$. If there are 2 or more cut-nodes we distinguish two cases. First, assume that there are 2 cut-nodes such that one is neither an ancestor nor a descendant of the other. Then their combined weight is at least $4m$ and thus $m \leq (1/4)\mathsf{w}(p)$. But we have $1/4 < e^{-1} < (1-\alpha)^{N-1}$ by the assumptions of the theorem. If there are 2 cut-nodes c_1 and c_2 such that c_1 is an ancestor of c_2, we know that c_1 is the parent of an internal node not on the path between c_1 and c_2 and thus the weight of c_1 is at least $3m$. We conclude that $m \leq (1/3)\mathsf{w}(p) < e^{-1}\mathsf{w}(p)$ in this case.

Now consider the case that there is exactly one cut-node c. If the maximum weight leaf is not in the subtree rooted at c we conclude $\mathsf{w}(p) \geq 3m$ and finish the proof for this case as above. Otherwise, let the children of c be x and y and assume without loss of generality that the maximum weight leaf is contained in the leaf-branch rooted at x. Denote the number of bisection steps in the leaf-branch rooted at x (y) by d_x (d_y), and let N' denote the number of leaves in the subtree rooted at c. Observe that $N' \geq 4$, $1 \leq d_x, d_y \leq N' - 3$ and $d_x + d_y = N' - 2$. As we have $\alpha \leq 1/5$ and the maximum weight leaf is contained in the leaf-branch rooted at x, we conclude $\mathsf{w}(x) \geq m(\frac{1}{1-\alpha})^{d_x}$ using Lemma 5.6. Since any internal node in the leaf-branch rooted at y has weight at least m, we have $\mathsf{w}(y) \geq m(\frac{1}{1-\alpha})^{d_y - 1}$ as in the proof of Theorem 5.2. Combining these two bounds and substituting $d_y = N' - 2 - d_x$ yields:

$$\mathsf{w}(x) + \mathsf{w}(y) \geq m\left(\left(\frac{1}{1-\alpha}\right)^{d_x} + \left(\frac{1}{1-\alpha}\right)^{N'-3-d_x}\right) \geq m\left(\frac{1}{1-\alpha}\right)^{N'-1},$$

where the last inequality is equivalent to $(1-\alpha)^{N'-1-d_x} + (1-\alpha)^{d_x+2} \geq 1$. This can be shown to hold for $\alpha \leq 1/5$ by a straightforward calculation using analytic techniques. Hence, using $\mathsf{w}(c) = \mathsf{w}(x) + \mathsf{w}(y)$ we have

$$\mathsf{w}(c) \geq m\left(\frac{1}{1-\alpha}\right)^{N'-1}. \tag{5.4}$$

Assume that there are d_z nodes, $d_z \geq 0$, above c on the path from c to the root p of the bisection tree. Observe that

$$\mathsf{w}(p) \geq \mathsf{w}(c)\left(\frac{1}{1-\alpha}\right)^{d_z}. \tag{5.5}$$

As $N = N' + d_z$, Equations (5.4) and (5.5) imply $m \leq \mathsf{w}(p)(1-\alpha)^{N-1}$. □

It is easy to verify $N(1-\alpha)^{N-1} \leq r_\alpha$ for $N \leq \lfloor \frac{1}{\alpha} \rfloor$ observing that the left-hand side of this inequality is monotone increasing from $N = 1$ to $N = \lfloor \frac{1}{\alpha} \rfloor$ and the inequality trivially holds for the latter value of N. Figure 5.6 compares the general with the improved upper bound on the ratio between $\max_{1 \leq i \leq N} \mathsf{w}(p_i)$ and $\frac{\mathsf{w}(p)}{N}$ for $N = 8$.

Let $\hat{\alpha}$ be the real root of the equation $(1-\alpha)^2 + (1-\alpha)^3 - 1 = 0$. It can be shown that $\hat{\alpha} \approx 0.245122$ is the largest possible value for α in Lemma 5.6 and Theorem 5.7. For $\alpha > \hat{\alpha}$ there are indeed leaf-branches and bisection

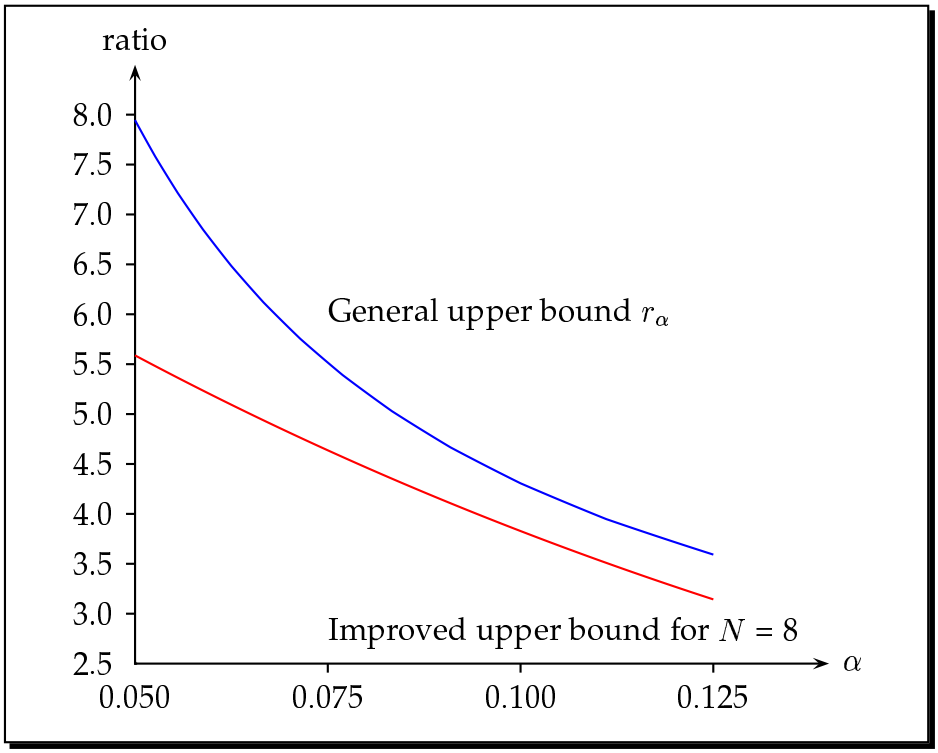

Figure 5.6: Comparison of general and improved upper bound

trees with a maximum weight leaf that is heavier than the upper bound provided by Lemma 5.6 and Theorem 5.7. If we choose $\alpha = 1/4$ and $N = 4$, for example, there is a leaf-branch whose maximum leaf weight is $(3/7)\mathsf{w}(p) > (3/4)^3\mathsf{w}(p)$. Furthermore, it is possible to construct bisection trees with $\max_{1\le i\le N} \mathsf{w}(p_i) = \mathsf{w}(p)(1-\alpha)/(2-\alpha)$ for $N = 4$, $\alpha \le (3-\sqrt{5})/2$. Figure 5.7 illustrates these exceptional cases for $N = 4$.

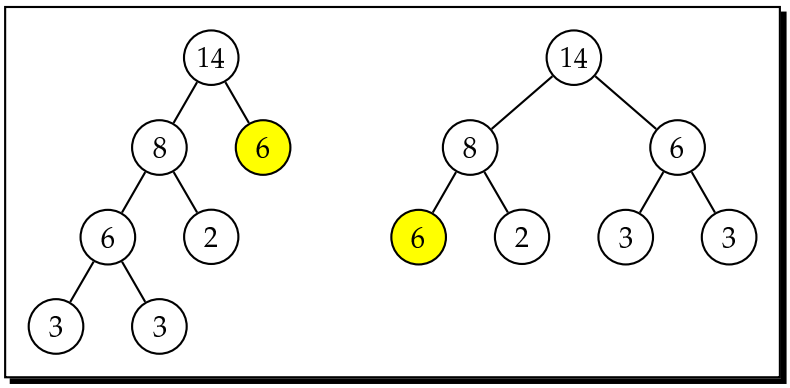

Figure 5.7: Worst-case leaf-branch and bisection tree for $N = 4$, $\alpha = \frac{1}{4}$

5.1.3 Generating more Subproblems

It is suggestive to improve the balancing quality of Algorithm HF or any other bisection-based load balancing algorithm for our model by generating more than N subproblems that are subsequently assigned to the N processors. We show in this subsection that this is indeed possible and discuss possible drawbacks of this approach.

If we assume that a bisection-based load balancing algorithm generates $N' > N$ independent subproblems, the assignment of those subproblems to the N processors is an instance of the MULTIPROCESSOR SCHEDULING problem [BEP+96, Gra66, Gra69]. Since the decision variant of this problem is $\mathcal{NP}$-complete in the strong sense if N is part of the problem instance, it is unlikely that an efficient (i.e., polynomial time) algorithm to compute an optimum solution exists. Therefore, we have to use approximation algorithms to find solutions that are as close to the optimum as possible.

It has been shown [HS87] that MULTIPROCESSOR SCHEDULING admits a polynomial approximation scheme[1]. However, it seems difficult to use this or related [HS86, Fri84] results for our purposes because we are interested in the deviation of the maximum load of the approximate solution from the ideal average load.

Therefore we use GRAHAM's list scheduling algorithm [Gra66, Gra69] to assign the N' subproblems generated by Algorithm HF. List scheduling proceeds as follows: Given a list of jobs with fixed weights, the job from the head of the list is deleted and assigned to the currently least loaded machine until the list is empty. We refer to this extension of Algorithm HF as Algorithm HFL, and denote by $w_{\max}^{(N,N')}$ the maximum load that Algorithm HFL generates on N processors using $N' - 1$ bisections.

Theorem 5.8 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ that has α-bisectors. Given a problem $p \in \mathcal{P}$ and positive integers N, N', it holds for Algorithm HFL that*

$$w_{\max}^{(N,N')} \leq \frac{\mathsf{w}(p)}{N}\left(1 + \frac{N}{N'} r_\alpha\right).$$

[1] A polynomial approximation scheme (PAS) is a family of algorithms $\{A_\varepsilon\}$ such that A_ε produces a $(1+\varepsilon)$- approximate solution. The running time of A_ε is polynomial in the length of the input for fixed ε. This means that the running time of a PAS may be, for example, exponential in $1/\varepsilon$.

Proof: For $N' \leq N$ the claim follows immediately from Theorem 5.2. Assume now that $N' > N$ and let $p_1, p_2, \ldots, p_{N'}$ denote the subproblems that resulted from calling HF(p,N'). Consider any processor k that has received load $w_{\max}^{(N,N')}$ and let p_l be the last subproblem assigned to it by the list scheduling algorithm. Since k was the least loaded processor when p_l was assigned, its load was at most $\mathsf{w}(p)/N$. Thus, we have

$$w_{\max}^{(N,N')} \leq \frac{\mathsf{w}(p)}{N} + \mathsf{w}(p_l) \leq \frac{\mathsf{w}(p)}{N} + \max_{1 \leq i \leq N'} \mathsf{w}(p_i).$$

As Theorem 5.2 implies $\max_{1 \leq i \leq N'} \mathsf{w}(p_i) \leq \mathsf{w}(p)/N' \cdot r_\alpha$ the proof is complete. □

If we choose $N' > 2N$ the worst-case upper bound of the above theorem is better than the one provided by Theorem 5.2 for any $\alpha \leq 1/2$. The upper bound of Theorem 5.8 can be improved slightly if $\max_{1 \leq i \leq N'} \mathsf{w}(p_i) > 2\mathsf{w}(p)/N$ by first sorting the list of subproblems according to their weight in non-increasing order.

There are two possible drawbacks of this approach. First, the load balancing overhead increases and therefore the gain in balancing quality has to be greater than the additional overhead in order to reduce the overall runtime of the application. Second, in some applications the bisection of a problem entails communication during the execution of the generated subproblems. A greater number of subproblems therefore most likely increases the network load. Again, this may outweigh the improved load balance.

5.2 Application of Algorithm HF to Distributed Finite Element Simulations

In this section, we present the application of Algorithm HF for load balancing in the field of numerical simulations with the finite element (FE) method [Bra97, Bur87, Sch84]. The FE method is used in statics analysis, for example, to calculate the response of objects under certain loading and boundary conditions.

In [Hüt96, HS94], an adaptive FE method based on the principle of recursive substructuring has been developed. It is an iterative procedure where in several runs of computation the result is improved automatically

until a predefined accuracy is reached. The costs for achieving this accuracy are much lower than with a non-adaptive procedure.

5.2.1 Recursive Substructuring

Starting an analysis with the FE method, an object is described by defining its shape and its structural properties. Then, the boundary and loading conditions have to be imposed on the object. A system of partial differential equations describes the relation between external loads and internal forces.

As an example from structural engineering, we consider a short cantilever under plane stress conditions, a problem from the domain of plane elasticity. The quadratic panel is uniformly loaded on its upper side. The left side of the cantilever is fixed as shown in Figure 5.8 (cf. [Hüt96, HS94]).

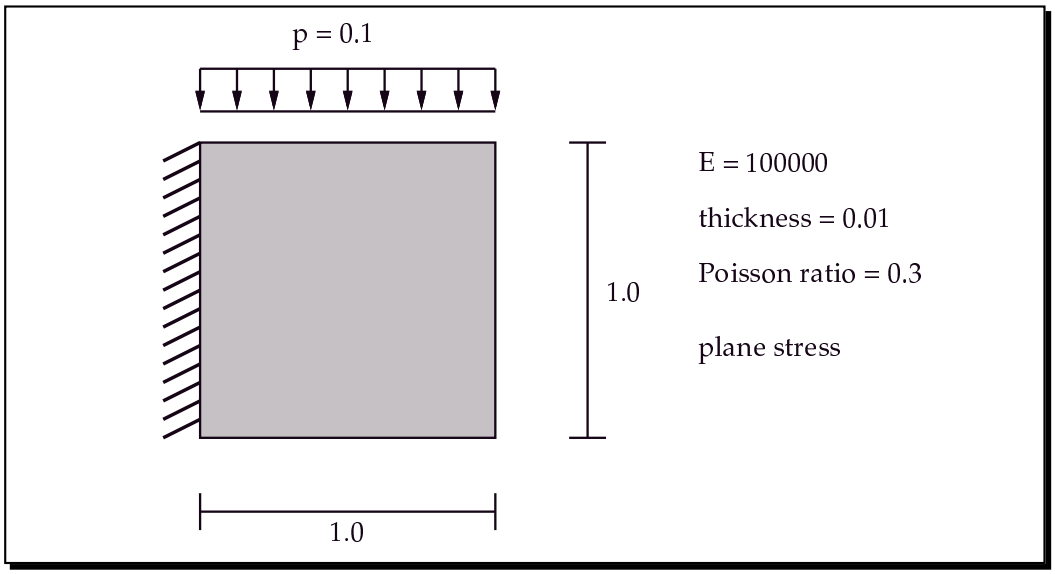

Figure 5.8: Static system of a short cantilever

The physical properties for the material of the cantilever are given by the Young's modulus E and the Poisson ratio ν. The differential equations (5.6) and (5.7) describe the response of the object under the external loads (see [Sch84]):

$$\frac{E}{1-\nu^2} \cdot \frac{\partial^2 u}{\partial x^2} + \frac{E}{2(1-\nu)} \cdot \frac{\partial^2 v}{\partial x \partial y} + \frac{E}{2(1+\nu)} \cdot \frac{\partial^2 u}{\partial y^2} = -f \tag{5.6}$$

$$\frac{E}{1-\nu^2}\cdot\frac{\partial^2 v}{\partial y^2}+\frac{E}{2(1-\nu)}\cdot\frac{\partial^2 u}{\partial x\partial y}+\frac{E}{2(1+\nu)}\cdot\frac{\partial^2 v}{\partial x^2}=-g, \tag{5.7}$$

where u and v are the unknown displacements and f and g the external forces in x- and y-direction, respectively.

We substructure the physical domain of the cantilever recursively (Figure 5.9, left). With the method of [Hüt96, HS94], a tree data structure is built reflecting the hierarchy of the substructured domain (see Figure 5.9, right). In each node, points on the separator line represent unknown values of displacement, and points on the border carry variable boundary conditions imposed by the parent node. Each tree node contains a system of linear equations whose *stiffness matrix* S determines the unknown displacement values dependent on the external forces:

$$S\begin{pmatrix}\hat{u}\\ \hat{v}\end{pmatrix}=\begin{pmatrix}\hat{f}\\ \hat{g}\end{pmatrix}.$$

In the leaves, the system of linear equations is constructed by a standard FE discretization. Roughly speaking, the equations are obtained by an approximation of the functions u, v, f, and g by linear combinations ($\hat{u}$, $\hat{v}$, $\hat{f}$, and $\hat{g}$, respectively) of partially bilinear basis functions with limited support within the discretizing mesh, and some additional algebraic and analytical transformations. The system of linear equations of an internal tree node is assembled out of the equations of its children, as described in [Hüt96].

Now, the task is to solve all those systems of linear equations. We use an iterative solver which traverses the tree several times, promoting displacements in top-down direction and reaction forces in bottom-up direction. In each node, the amount of work to be done stays the same during the iterations. But since the adaptive structure of the tree is not known *a priori*, it is essential to have a good load balancing strategy before the parallel execution of the solving phase.

5.2.2 Application of Algorithm HF

We assign a load value $\ell(v)$ to each tree node v, given by

$$\ell(v)=C_b\,n_b(v)+C_s\,n_s(v)$$

with $n_b(v)$ points on the border without boundary conditions (grey points in Figure 5.9), and $n_s(v)$ points on the separator line of node v (black points

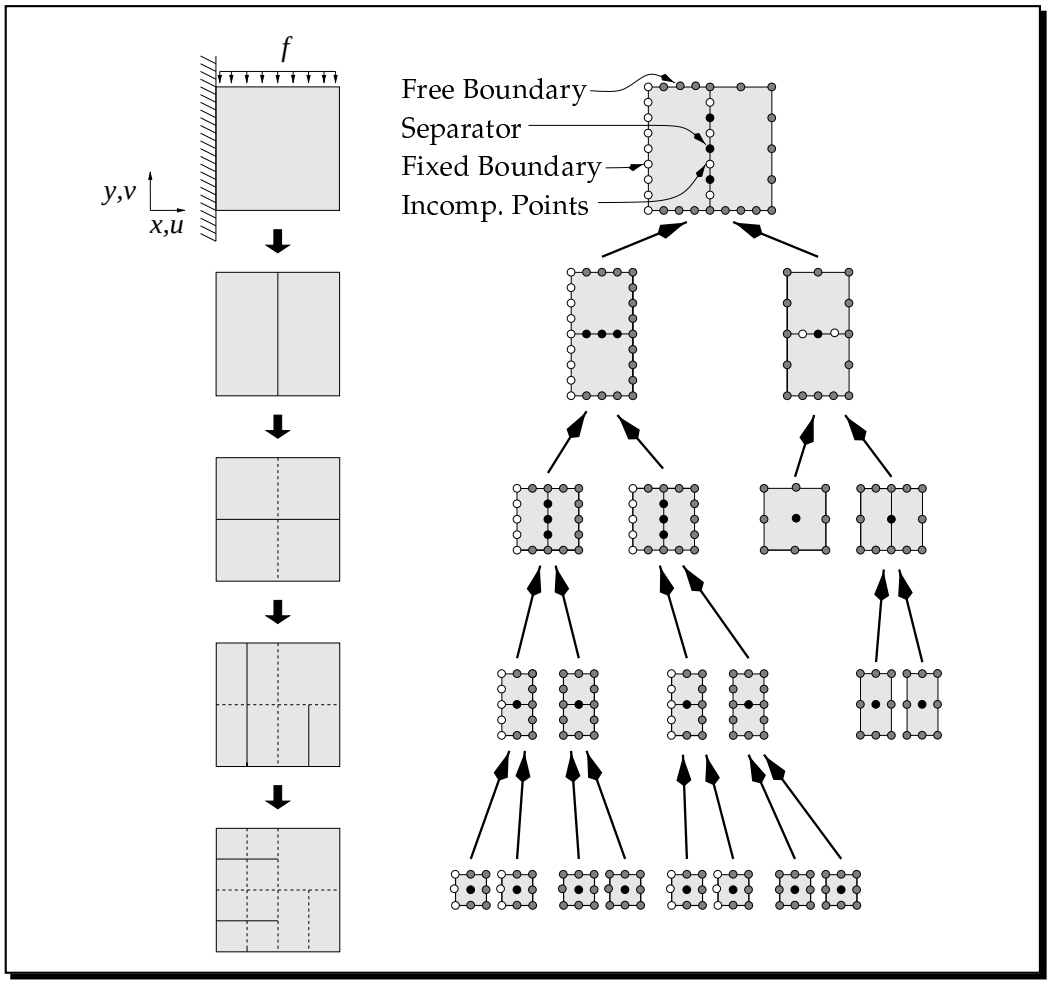

Figure 5.9: A coarse discretizing mesh and the resulting binary tree data structure for the short cantilever

belonging to the borders of both children in Figure 5.9). The load value $\ell(v)$ models the computing time of node v, where the constants C_s and C_b are independent of node v and $C_s \approx 6\,C_b$. Points with fixed boundary values as well as incompatible points on the separator (white points) do not contribute to the load value $\ell(v)$.

We can interpret the FE tree as an approximate (potential) bisection tree by accumulating the load of all nodes in the subtree rooted in node v to get the weight value $\mathsf{w}(v)$:

$$\mathsf{w}(v) = \begin{cases} \ell(v) & \text{if } v \text{ is a leaf} \\ \ell(v) + \mathsf{w}(c_1) + \mathsf{w}(c_2) & \text{if } v \text{ is internal or the root,} \end{cases}$$

where c_1 and c_2 are the children of v.

The weight values are collected during the tree construction phase by simply counting and accumulating the number of points on the separator of each tree node.

If we want to apply Algorithm HF to this tree of weight values, we must specify which bisection steps the algorithm can perform. Our first approach is to define a bisection step as the removal of the root node v of a subtree. This yields two subtrees rooted at the children c_1 and c_2 of v, and v is ignored for the remainder of the load balancing phase.

Such bisection steps do not exactly match Definition 5.1, because the weight of node v exceeds the weight sum $\mathsf{w}(c_1) + \mathsf{w}(c_2)$ of the children by $\ell(v)$. However, $\ell(v)$ (work load of the one-dimensional separator) is negligible compared to $\mathsf{w}(v)$ (work load of the two-dimensional domain) in our application if the FE tree is large enough. Hence, the results of Theorem 5.2 and Corollary 5.3 are well approximated.

Algorithm HF chops N subtrees off the FE tree, each of which can be traversed in parallel by the iterative solver. These N subtrees contain the main part of the solving work and may be distributed over the available N processors. The upper $N - 1$ tree nodes cannot exploit the whole number of processors, anyway. Therefore, such a distribution does not sacrifice parallel potential in the upper tree levels.

5.2.3 Runtime Examples

EBNER and PFAFFINGER [EP98] described a parallel implementation of the recursive substructuring technique that uses the dataflow language FASAN as coordination and automatic parallelization platform. In our implementation, however, a hand-coded parallel version based on PVM (see [BDG+94]) is used in order to keep the communication overhead as low as possible. The number of solver iterations (tree traversals) was fixed to 100. The experiments were run on a cluster of workstations of type HP 9000/720.

Figure 5.11 shows the runtime results of the numerical simulation of the short cantilever under uniform load described above. In this experiment we have chosen a rather small FE tree with 1,279 element nodes and maximum depth 11 (see the left discretizing mesh in Figure 5.10, representing the leaves of the FE tree). Since the adaptivity was limited to only two additional levels in the FE tree, the node weights resulted in $\alpha = 0.18571$.

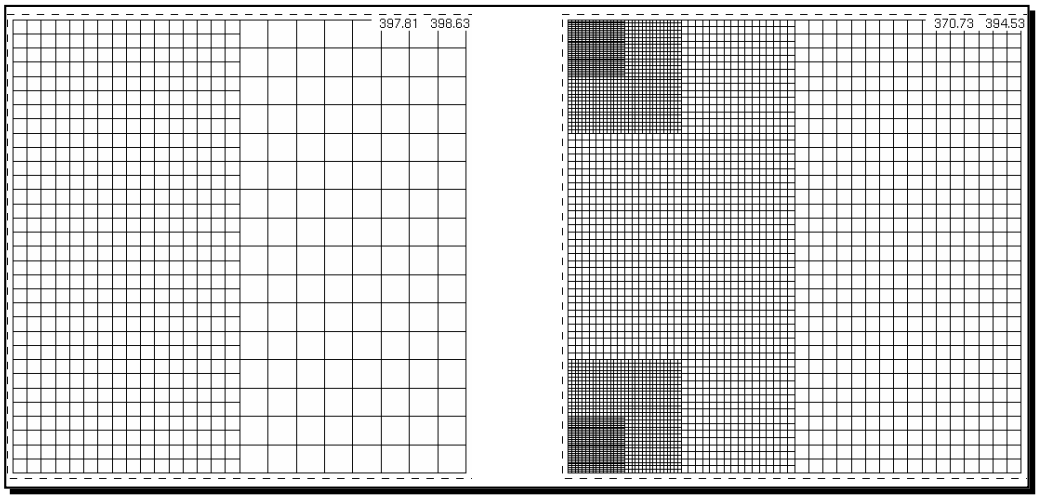

Figure 5.10: The discretizing meshes for the domain of the short cantilever

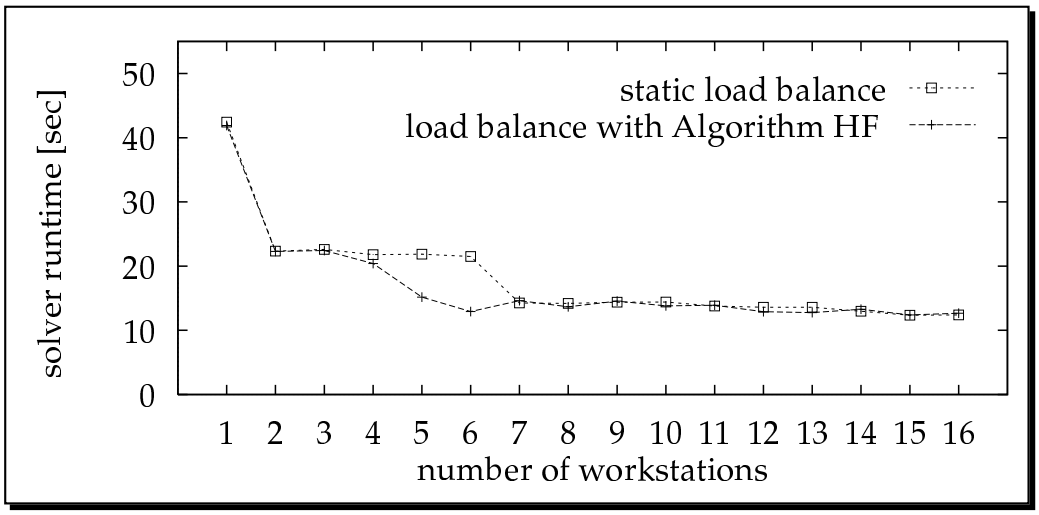

Figure 5.11: Runtime results for 1,279 tree nodes

For 2 workstations, the partitioning generated by Algorithm HF is identical to the static partitioning (just chopping off the root node). Further speedup from 4 to 6 processors with Algorithm HF occurs earlier than with static partitioning (from 6 to 7 processors). In this comparatively small problem, it is mainly the critical path (cf. Section 4.1) of the FE tree that determines the lower bound on the tree traversal time and inhibits further acceleration with more than 6 processors.

The effect of Algorithm HF is more significant in larger simulations, where adaptivity for high numerical accuracy is distinct and where it is

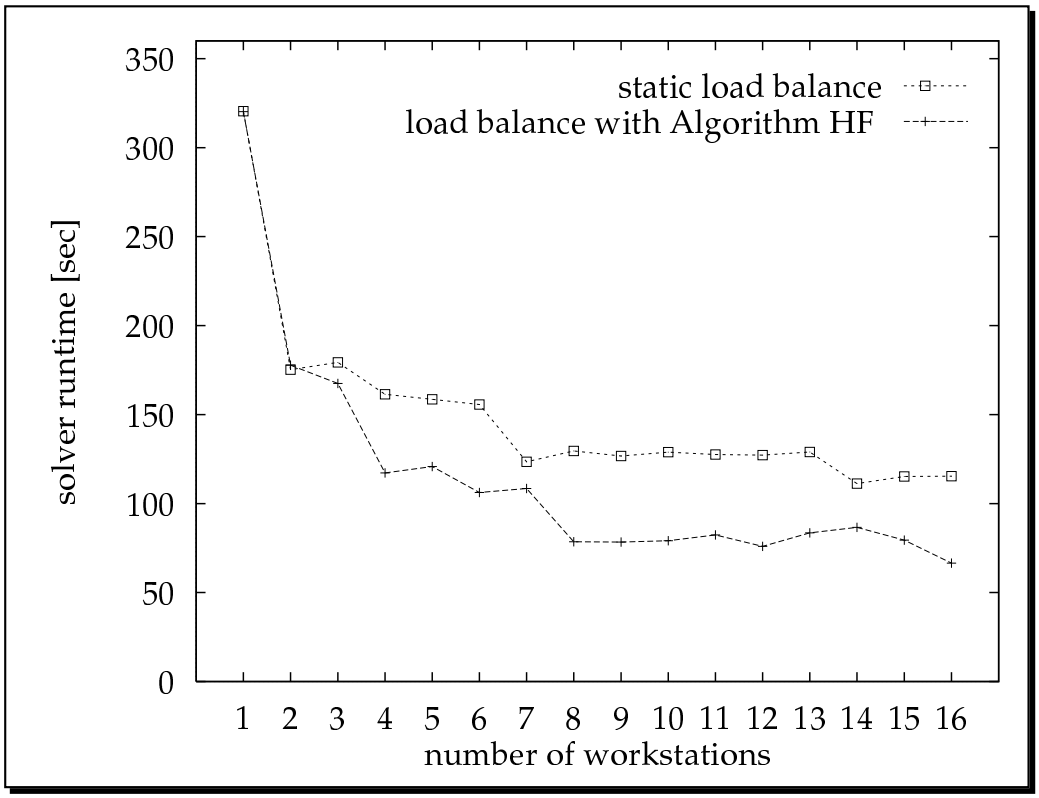

Figure 5.12: Runtime results for 11,263 tree nodes

essential to split the biggest subtrees. The runtime results of a computation with a deeper FE tree of maximal depth 17 are shown in Figure 5.12. It contains 11,263 nodes (see Figure 5.10, right side), and the value of the bisection parameter $\alpha = 0.10615$ is rather bad. Here, the runtime improvement with Algorithm HF is more significant, since the load value $\ell(p)$ of the root p is relatively small compared to the weight sum $w(p)$ of the whole FE tree. We observe that the distributed iterative solver is up to 70% faster with application of Algorithm HF in comparison to the static partitioning if at least four processors are used. Even on 16 processors, static partitioning does not split the subtree which is responsible for the overall execution time.

Nevertheless, we clearly have to recognize once more the strong influence of the critical path of the FE tree on the parallel execution time. Employing Algorithm HF, we almost reach the minimum execution time using only 8 processors. This implies that larger numbers of processors can only be used efficiently if the the FE tree is of adequate size and structure.

5.2.4 Further Improvements

For arbitrary adaptive FE simulations, we cannot give a limit for the bisection parameter α using the bisection method described above. If α gets too small, there are two possibilities to prevent an unsatisfactory partitioning:

- During the tree construction phase, we can choose between horizontal and vertical bisection of the subdomain of each node, whichever leads to the larger local value of α.
- Moreover, we might set N as a small multiple of the number of available processors, so there is still a chance to compensate a small α value by assigning multiple partitions to one processor. We expect that the increased load balancing overhead and the additional network load will be tolerable because of the coarse granularity of our parallel application (see also Section 5.1.3).

To avoid the small-α-problem completely, another strategy using Algorithm HF allows the removal of a single edge of a tree as a bisection step. This strategy partitions the *entire* given FE tree into N subtrees of approximately equal size. Section 5.3 shows that FE trees satisfying the conditions

$$\begin{aligned} \ell(v) &\leq \ell(c_1) + \ell(c_2) \\ \ell(v) &\geq \ell(c_i) \qquad (i = 1, 2) \end{aligned}$$

have good bisectors. This application of Algorithm HF also takes into account that the main memory resources of the processors become the limiting factor if very high accuracy of the simulation is required. In this case, finding a partitioning of the entire tree (not only a set of equal-sized subtrees ignoring their ancestors in the tree) is necessary.

5.3 Weighted Trees with Good Bisectors

Let $\mathcal{T}$ be the set of all rooted binary trees with node weights $\ell(v)$ satisfying:

(1) $\ell(v) \leq \ell(c_1) + \ell(c_2)$ for nodes v with two children c_1 and c_2
(2) $\ell(v) \geq \ell(c)$ if c is a child of v

The weight of a tree $T = (V, E)$ in $\mathcal{T}$ is defined as $\mathsf{w}(T) = \sum_{v \in V} \ell(v)$.

This class $\mathcal{T}$ of binary trees models the load of applications in hierarchical finite element simulations, as discussed in Section 5.2. Recall that in these applications the domain of the computation is repeatedly subdivided into smaller subdomains. The structure of the domains and subdomains yields a binary tree in which every node has either two children or is a leaf. The resource demands (CPU and main memory) of the nodes in this FE tree are such that the resource demand at a node is at most as large as the sum of the resource demands of its two children. In order to parallelize the computation, it is necessary to distribute the FE tree among a number of processors in a balanced way.

Note that Conditions (1) and (2) ensure that the two subtrees obtained by removing a single edge from a tree in $\mathcal{T}$ are also members of $\mathcal{T}$.

The following theorem shows that trees from the class $\mathcal{T}$ can be $\frac{1}{4}$-bisected by removal of a single edge unless the weight of the tree is concentrated in the root.

Theorem 5.9 *Let $T = (V, E)$ be a tree in $\mathcal{T}$, and let r be its root. If $\ell(r) \leq \frac{3}{4}\mathsf{w}(T)$, then there is an edge $e \in E$ such that the removal of e partitions T into subtrees T_1 and T_2 with $\mathsf{w}(T_1), \mathsf{w}(T_2) \in [\frac{1}{4}\mathsf{w}(T), \frac{3}{4}\mathsf{w}(T)]$.*

Proof: We give a simple method to find the required edge. Pick an arbitrary node v of T as a start node. While $T \setminus \{v\}$ contains a subtree T' with $\mathsf{w}(T') > \frac{3}{4}\mathsf{w}(T)$, replace v by the node adjacent to v which is contained in T'. This process always terminates after less than $|V|$ iterations at a node v such that all subtrees T' in $T \setminus \{v\}$ satisfy $\mathsf{w}(T') \leq \frac{3}{4}\mathsf{w}(T)$. We claim that at least one of these subtrees also satisfies $\mathsf{w}(T') \geq \frac{1}{4}\mathsf{w}(T)$, and thus the edge connecting v and T' can be picked as the required separator edge. In order to prove the claim we distinguish several cases regarding the position of v in T (see Figure 5.13). For every case the assumption that all subtrees of $T \setminus \{v\}$ have weight $< \frac{1}{4}\mathsf{w}(T)$ will lead to a contradiction.

Case 1: v has degree 3. Let u be the parent of v. Let T_1 be the subtree of $T \setminus \{v\}$ that contains u, and let T_2 and T_3 be the other two subtrees. Assume that all three subtrees have weight $< \frac{1}{4}\mathsf{w}(T)$. Consequently, v must have weight $> \frac{1}{4}\mathsf{w}(T)$ because $\mathsf{w}(T) = \mathsf{w}(T_1) + \mathsf{w}(T_2) + \mathsf{w}(T_3) + \ell(v)$. But then u must also have weight $> \frac{1}{4}\mathsf{w}(T)$ because it is the parent of v, and this implies $\mathsf{w}(T_1) > \frac{1}{4}\mathsf{w}(T)$. A contradiction.

Case 2: v has degree 2 and is the root of T. Let T_1 and T_2 be the two subtrees of $T \setminus \{v\}$, and let u_1 and u_2 be the corresponding children of v. If both subtrees have weight $< \frac{1}{4}\mathsf{w}(T)$, it follows also that $\ell(u_1) < \frac{1}{4}\mathsf{w}(T)$ and

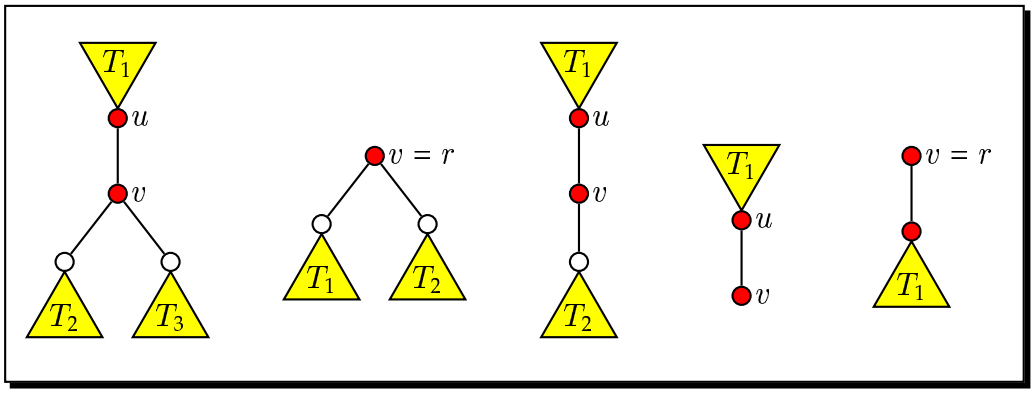

Figure 5.13: The 5 cases for v in proof of Theorem 5.9

$\ell(u_2) < \frac{1}{4}\mathsf{w}(T)$, which implies $\ell(v) \leq \ell(u_1) + \ell(u_2) < \frac{1}{2}\mathsf{w}(T)$. On the other hand, $\mathsf{w}(T) = \mathsf{w}(T_1) + \mathsf{w}(T_2) + \ell(v)$ implies $\ell(v) > \frac{1}{2}\mathsf{w}(T)$. A contradiction.

Case 3: v has degree 2 and is not the root of T. Let u be the parent of v, and let c be the child of v. Let T_1 be the subtree in $T \setminus \{v\}$ that contains u, and let T_2 be the other subtree. If both subtrees have weight $< \frac{1}{4}\mathsf{w}(T)$, $\mathsf{w}(T) = \mathsf{w}(T_1) + \mathsf{w}(T_2) + \ell(v)$ implies $\ell(v) > \frac{1}{2}\mathsf{w}(T)$. But then $\ell(u) \geq \ell(v) > \frac{1}{2}\mathsf{w}(T)$. A contradiction.

Case 4: v has degree 1 and is a leaf of T. Let T_1 be the tree $T \setminus \{v\}$, and let u be the parent of v. Assume that $\mathsf{w}(T_1) < \frac{1}{4}\mathsf{w}(T)$. Then $\mathsf{w}(T) = \mathsf{w}(T_1) + \ell(v)$ implies $\ell(v) > \frac{3}{4}\mathsf{w}(T)$. But then $\mathsf{w}(T_1) \geq \ell(u) \geq \ell(v) > \frac{3}{4}\mathsf{w}(T)$. A contradiction.

Case 5: v has degree 1 and is the root of T. Let T_1 be the tree $T \setminus \{v\}$. As $\ell(v) = \ell(r) \leq \frac{3}{4}\mathsf{w}(T)$, $\mathsf{w}(T) = \ell(v) + \mathsf{w}(T_1)$ implies $\mathsf{w}(T_1) \geq \frac{1}{4}\mathsf{w}(T)$. □

According to Theorem 5.2 a problem p from a class of problems that has $\frac{1}{4}$-bisectors can always be subdivided into N subproblems $p_1, \ldots, p_N$ such that $\max_{1 \leq i \leq N} \mathsf{w}(p_i) \leq \frac{\mathsf{w}(p)}{N} \cdot \frac{9}{4}$. The following corollary gives a condition on trees in $\mathcal{T}$ that ensures that they can be subdivided into N subproblems using $\frac{1}{4}$-bisectors.

Corollary 5.10 *Let $T = (V, E)$ be a tree in $\mathcal{T}$, and let r be its root. Let N be a positive integer. If $\mathsf{w}(T) \geq \frac{4}{3}(N-1)\ell(r)$, Algorithm HF partitions T into N subtrees by cutting exactly $N-1$ edges such that the maximum weight of the resulting subtrees is at most $\frac{9}{4} \cdot \frac{\mathsf{w}(T)}{N}$.*

Proof: After k bisection steps according to Theorem 5.9 there are $k+1$ subtrees. There is at least one subtree T' with weight at least $\frac{\mathsf{w}(T)}{k+1}$. Let r' be

the root of T'. If $k+1 < N$, we have $\mathsf{w}(T') \geq \frac{\mathsf{w}(T)}{k+1} \geq \frac{\mathsf{w}(T)}{N-1} \geq \frac{4}{3}\ell(r) \geq \frac{4}{3}\ell(r')$, and another bisection step is possible.

The upper bound on the maximum weight of any subtree follows directly from Theorem 5.2 for $\alpha = 1/4$ (see also Table 5.1). □

Note that an optimal min-max k-partition of a weighted tree (i.e., a partition with minimum weight of the heaviest component after removing k edges) can be computed in linear time [BP95, Fre91]. These algorithms are preferable to our approach using Algorithm HF in the case of trees that are to be subdivided by removing a minimum number of edges. Since the heaviest subtree in the optimal solution does obviously not have a greater weight than the maximum generated by Algorithm HF, the bound from Corollary 5.10 still applies and provides a non-trivial worst-case performance guarantee for these optimal algorithms as well. We are not aware of any efficient parallel algorithm for solving this tree partitioning problem optimally. Therefore, the parallel algorithms for load balancing with good bisectors presented in the next chapter also provide a good and fast parallel approximation for classes of trees with good bisectors.

5.4 Classification of Algorithm HF

We apply the classification scheme for load distribution strategies that was presented in Chapter 3 to Algorithm HF (see Table 5.2). Since it is straightforward to classify Algorithm HF, we only give a few remarks regarding some of the classification criteria.

Our system model and the theoretical analysis of Algorithm HF is purely combinatorial. Assuming that the subproblems can be solved independently, we clearly have non-interacting entities that are transferred non-preemptively through the whole system. Algorithm HF can be used for any target topology and uses complete information in a centralized fashion (we will show how to overcome this "shortcoming" in the next chapter). All targets participate in the load distribution cooperatively. The algorithm is dynamic since all decisions are made during runtime and the weights of the subproblems are used as load values. Load distribution is initiated by the parallel application at the location of the initial problem.

Table 5.2: Classification of Algorithm HF

System Model	
Model Flavor	combinatorial
Target Topology	arbitrary
Entity Topology	non-interacting entities
Transfer Model	
Transfer Space	systemwide
Transfer Policy	non-preemptive
Information Exchange	
Information Space	central
Information Scope	complete
Coordination	
Decision Structure	centralized
Decision Mode	cooperative
Participation	global
Algorithm	
Decision Process	dynamic
Initiation	central
Adaptivity	fixed
Cost Sensitivity	none
Stability Control	not required

Chapter 6

Parallel Load Balancing for Problems with Good Bisectors

In this chapter we extend the results for our load balancing model obtained in the previous chapter to parallel algorithms in order to reduce the balancing overhead as much as possible. We present efficient parallel load balancing algorithms for problems with good bisectors that maintain identical or comparable worst-case bounds on the balancing quality. The results apply to a large class of parallel machines since the algorithms use only a few basic primitives that can be performed efficiently in parallel. To provide additional insight into the *average-case* behavior and the relative performance of the proposed algorithms, we carried out extensive simulation experiments regarding the load imbalance incurred by the proposed load balancing algorithms.

We assume a parallel system that consists of N processors, numbered from 1 to N. The number of a processor is referred to as its *id*. The i-th processor is denoted P_i . Initially, a problem p resides on P_1 and P_1 is called *busy*, while the other processors are idle and called *free*. A free processor becomes busy when it receives a subproblem from a busy processor. The goal of a parallel load balancing algorithm in this model is to split p into N subproblems $p_1, p_2, \ldots, p_N$ such that subproblem p_i can subsequently be processed by P_i , $1 \leq i \leq N$. However, it is allowed to split p into fewer than N subproblems. In this case, some processors remain idle.

Furthermore, we assume that the bisection of a problem into two subproblems requires one unit of time and that the transmission of a subproblem to a free processor requires one unit of time. Our results

can easily be adapted to different assumptions about the time for bisections and for interprocessor communication. Finally, we assume that standard operations like computing the maximum weight of all subproblems generated so far or sorting a subset of these subproblems according to their weights can be done in time $O(\log N)$. This assumption is satisfied by the idealized PRAM model [JáJ92], which can be simulated on many realistic architectures with at most logarithmic slowdown.

The remainder of this chapter is organized as follows. Section 6.1 first presents and analyzes a parallel implementation of Algorithm HF and discusses certain drawbacks regarding the communication overhead of this parallelization (Section 6.1.1). Consequently, Algorithm BA, a highly parallel load balancing algorithm for problems with good bisectors, is described and analyzed (Section 6.1.2). Then it is shown that Algorithm BA-HF, which is a combination of Algorithm BA and Algorithm HF, can be used to obtain both a very fast parallel runtime and a performance guarantee for the balancing quality that is arbitrarily close to the one for Algorithm HF (Section 6.1.3). The management of free processors for the three parallel algorithms is discussed in Section 6.1.4. To compare the proposed parallel load balancing approaches, we give simulation results in Section 6.2 for the achieved load distribution under certain stochastic assumptions.

6.1 Parallel Load Balancing

Although Algorithm HF achieves good load balance even in the worst case, its drawback is that it is a sequential algorithm that bisects only one problem at a time. Hence, the time for load balancing grows (at least) linearly with the number of processors. If the number N of processors is large, it is clearly advantageous to use a parallel algorithm for the decomposition of a given problem into the desired number of subproblems. We assume that each bisection step is performed sequentially on one processor. Therefore, the time spent in load balancing can be reduced substantially if several bisection steps are executed concurrently on different processors.

Recall that the problem that is to be partitioned resides on a single processor initially and that the other $N - 1$ processors are free. For now we assume that a processor that bisects a problem can quickly acquire the number of a free processor and, after bisecting the problem into two

subproblems, send one of the two subproblems to that free processor. In Section 6.1.4 we will discuss ways to implement such a management of free processors in a parallel system without increasing the asymptotic running time.

6.1.1 Parallelizing Algorithm HF

Algorithm HF repeatedly bisects the remaining subproblem with maximum weight. For a sequential implementation, one would use a priority queue to hold the remaining subproblems and, in each step, remove the maximum weight subproblem from the priority queue, bisect it into two subproblems, and insert the two resulting subproblems into the queue. In this section, we describe a possible parallelization of Algorithm HF. We refer to this parallel version of Algorithm HF as Algorithm PHF. Algorithm PHF is shown in Figure 6.1. Steps that require a form of global communication (communication involving more than two processors at a time) are shaded in the figure; on most parallel machines these steps will not take constant time, but time $\Theta(\log N)$. The pseudocode in Figure 6.1 is considerably more detailed than the code given for the other algorithms presented in this chapter. This is because an adequate description of Algorithm PHF requires showing the code for a particular processor P_i, whereas sketching the bisection process from a global perspective will be sufficient for the parallel algorithms presented in Sections 6.1.2 and 6.1.3.

In order to parallelize Algorithm HF, we must, on the one hand, perform several bisections simultaneously, but, on the other hand, ensure that no subproblem is bisected unless it would also have been bisected by the sequential Algorithm HF. Fortunately, the following two simple observations are helpful:

- Subproblems with weight greater than $\frac{\mathsf{w}(p)}{N} \cdot r_\alpha$ are certainly bisected by Algorithm HF. Consequently, such subproblems can be bisected in parallel.
- Subproblems with weight at most $\frac{\mathsf{w}(p)}{N}$ are certainly not bisected any further. These subproblems can be assigned to a processor immediately.

Hence, Algorithm PHF can carry out a first phase of load balancing as follows. Before the first bisection is made, the values $\mathsf{w}(p)$, N, and α

```
     Algorithm PHF(p, N)
     // code for P_i
     begin
     if i = 1 then q := p;
     else wait until a problem q is received;
     fi;
     while w(q) > (w(p)/N) · r_α do
           begin
           bisect q into q_1 and q_2;
(a)        send q_2 to a free processor;
           q := q_1;
           end;
(b)  barrier;
     // start of phase two
(c)  f := number of free processors;
     repeat
(d)        m := maximum weight of remaining subproblems;
(e)        h := number of proc. with subproblem ≥ m(1 − α);
           if h ≤ f then
                 if w(q) ≥ m(1 − α) then
                       bisect q into q_1 and q_2;
(f)                    send q_2 to a free processor;
                       q := q_1;
                 fi;
           else
                  determine the f heaviest subproblems;
                 if q is among them then
                       bisect q into q_1 and q_2;
(g)                    send q_2 to a free processor;
                       q := q_1;
                 fi;
           fi;
           f := f − min{h, f};
(h)        if f > 0 then barrier; fi;
     until f = 0;
     end.
```

Figure 6.1: Algorithm PHF (Parallel HF)

are broadcast to every processor. Unlike the sequential Algorithm HF, Algorithm PHF therefore requires advance knowledge of the bisection parameter α of the given class of problems. Then, P_1 bisects the original problem p into p_1 and p_2, and then sends p_2 to P_2.

Whenever a processor gets a subproblem q in the following (either a free processor gets it from a busy processor or a busy processor gets it from a bisection performed on the same processor), it checks whether $\mathsf{w}(q)$ is greater than $\frac{\mathsf{w}(p)}{N} \cdot r_\alpha$ and, if so, bisects it into two subproblems and sends one of the two subproblems to a free processor. (The question of determining which free processor the problem is sent to in this step (a) without increasing the asymptotic running time will be treated in Section 6.1.4.) This phase ends when all subproblems have weight at most $\frac{\mathsf{w}(p)}{N} \cdot r_\alpha$. (Detecting this in a distributed system is not trivial, but standard techniques for distributed termination detection can be employed. Once the termination of the first phase is detected, the free processors can be informed about this with a broadcast message.) A barrier synchronization is performed in step (b) to ensure that all processors finish the first phase of the algorithm at the same time. (Free processors that have not received a subproblem during the first phase go to step (b) of the algorithm directly as soon as they are informed about the termination of phase one.)

What is the running time for this first phase? Consider the bisection tree representing the bisections performed in the first phase. Let D denote the maximum depth of a leaf in this bisection tree. Under our assumptions, the running time for the first phase is clearly bounded by $O(D)$. Now observe that a node at depth d in the bisection tree has weight at most $\mathsf{w}(p)(1-\alpha)^d$. Therefore, D can be at most $\log_{\frac{1}{1-\alpha}} N$. The running time for the first phase can thus be bounded as follows.

Lemma 6.1 *The running time for Algorithm PHF during phase one is bounded from above by* $O(\log_{\frac{1}{1-\alpha}} N)$.

Note that the running time for phase one is, for constant α, larger than $\log N$ only by a constant factor. For comparison, recall that $\log N$ time is already required for broadcasting a value from one processor to the other $N-1$ processors in most models of parallel machines.

If one was only interested in obtaining a parallel algorithm with the same performance guarantee as Algorithm HF, it would be sufficient to stop the load balancing after this first phase and leave the remaining free processors idle. However, as can be seen from the simulation

results presented in Section 6.2 as well as the average-case analysis provided by [BSS99], the maximum load achieved by Algorithm HF is much smaller than the worst-case bound for many problem instances, especially when α is small. Therefore, the balancing quality of this approach would often be worse than what can be achieved if all available processors are used. Thus it is desirable to aim at a parallel solution that produces the same partitioning as the sequential Algorithm HF.

In order to achieve this, Algorithm PHF continues as follows. After the barrier synchronization in step (b), the number f of free processors is calculated in step (c) and made known to all processors. Let f_0 denote this initial value of f. At the same time, the free processors can be numbered from 1 to f, and the id of the i-th free processor can be stored at P_i. (This way, any processor can later on determine the id of the i-th free processor by questioning P_i. No global communication is required.)

Then, the second phase of the algorithm consists of iterations of the following steps:

1. Determine the maximum weight m among the subproblems generated so far (m is broadcast to all processors).

2. Determine the number h of processors that have a subproblem of weight at least $m(1-\alpha)$, and number them from 1 to h (h is broadcast to all processors).

3a. If $h \leq f$, all h processors that have subproblems of weight at least $m(1-\alpha)$ bisect their subproblem; the i-th such processor sends one of its two resulting subproblems to the i-th free processor that has not received a subproblem in a previous iteration.

3b. If $h > f$, the f heaviest subproblems are determined and numbered from 1 to f (employing either selection or sorting as a subroutine); for $1 \leq i \leq f$, the processor with the i-th among the f heaviest subproblems bisects its subproblem and sends one of the two resulting subproblems to the i-th free processor that has not received a subproblem in a previous iteration.

In each iteration, the value of f represents the number of remaining free processors. Every processor can update its copy of f locally by subtracting $\min\{h, f\}$. The load balancing terminates when there are no free processors left, i.e., when $f = 0$. Observe that the $\min\{h, f\}$ subproblems chosen for bisection in each iteration would also have been bisected by

the sequential Algorithm HF, because none of the bisections in the current iteration can generate a new subproblem with weight greater than $m(1-\alpha)$. Hence, Algorithm PHF produces the same partitioning of p into N subproblems as Algorithm HF.

Note that global communication is required in every iteration of phase two. The values of m and h can be determined and broadcast to all processors in steps (d) and (e) by simple prefix computations (see [JáJ92]) in time $O(\log N)$. The barrier at the end of every iteration takes time at most $O(\log N)$ as well. In all iterations except the last one, no further global communication is required: a processor that bisects a problem in that iteration can determine the id of a free processor by a single request to one of the processors $P_1, \ldots, P_{f_0}$. Only in the last iteration it can be necessary to determine the f heaviest subproblems and number them from 1 to f. This can be done by employing a parallel sorting or selection algorithm (see [JáJ92]). The time requirement is also $O(\log N)$, but with a larger constant than for the simple prefix computations.

Taking all this into account, the reader may easily verify that the running time for one iteration of phase two is $O(\log N)$ under our assumptions. In addition, the maximum weight among all subproblems is reduced at least by a factor of $(1-\alpha)$ in each iteration (unless all free processors receive a subproblem in the iteration, in which case the algorithm terminates). As it is clear that the maximum weight can never become smaller than $\frac{\mathsf{w}(p)}{N}$, the algorithm will terminate no later than after I iterations if I satisfies the following inequality:

$$\frac{\mathsf{w}(p)}{N} \cdot r_\alpha \cdot (1-\alpha)^I \leq \frac{\mathsf{w}(p)}{N}.$$

This can be rewritten as

$$\left\lfloor \frac{1}{\alpha} \right\rfloor \cdot (1-\alpha)^{I+\left\lfloor \frac{1}{\alpha} \right\rfloor - 2} \leq 1.$$

This inequality is implied by the following inequality

$$(1-\alpha)^{I+\left\lfloor \frac{1}{\alpha} \right\rfloor - 2} \leq \alpha. \tag{6.1}$$

From $(1-\alpha)^{\frac{1}{\alpha}} \leq 1/e$ and $(1/e)^{\ln \frac{1}{\alpha}} = \alpha$ we conclude $(1-\alpha)^{\frac{1}{\alpha} \ln \frac{1}{\alpha}} \leq \alpha$. Thus, (6.1) is satisfied for $I + \left\lfloor \frac{1}{\alpha} \right\rfloor - 2 \geq \frac{1}{\alpha} \ln \frac{1}{\alpha}$, which surely holds if $I \geq \frac{1}{\alpha} \ln \frac{1}{\alpha}$. Hence, we obtain the following lemma.

Lemma 6.2 *The number of iterations performed by Algorithm PHF in phase two is bounded from above by* $\frac{1}{\alpha} \ln \frac{1}{\alpha}$, *and each iteration takes time* $O(\log N)$. *Altogether, phase two has running time* $O(\frac{1}{\alpha} \ln \frac{1}{\alpha} \log N)$.

Lemmata 6.1 and 6.2 can be summarized by the following theorem.

Theorem 6.3 *Given a class* $\mathcal{P}$ *of problems with* α*-bisectors for a fixed constant* α, *there is a parallel implementation of Algorithm HF, called Algorithm PHF, that subdivides a problem from* $\mathcal{P}$ *into* N *subproblems in time* $O(\log N)$. *The resulting subproblems are the same as for the sequential Algorithm HF.*

While the described parallel implementation of Algorithm HF has optimal running time for constant α under our assumptions, there are also drawbacks of this approach. First, we have completely ignored the management of free processors in phase one so far. Although we will show in Section 6.1.4 that this can be implemented without increasing the asymptotic running time, it must be expected that substantial communication overhead will occur if many processors want to acquire the id of a free processor simultaneously. Further, the second phase of Algorithm PHF requires global communication in each iteration; effectively, the described implementation simulates a specialized parallel priority queue (see [BTZ98, BDMR96, San98] for further information on parallel priority queues) that allows selection of the $\min\{h, f\}$ heaviest remaining subproblems. While this overhead may be small on parallel machines with high-bandwidth and low-latency interconnection networks, it is likely to limit the speed-up achievable with this algorithm in practice on less powerful platforms, like networks of workstations.

6.1.2 Algorithm BA

To overcome the difficulties encountered with Algorithm PHF, we now propose an alternative algorithm for the load balancing problem that is inherently parallel. This algorithm tries to execute as many of the necessary bisection steps as possible concurrently while maintaining a worst-case bound on the resulting maximum load comparable to the bound for Algorithm HF.

The maximum load generated by this algorithm in the worst case is larger than the worst-case bound for Algorithm HF only by a small constant factor. In addition, the management of free processors can be realized

efficiently for this algorithm, and no global communication is required at all. Furthermore, we will show in the next Section that it is possible to integrate Algorithm BA and Algorithm HF into an efficient parallel load balancing algorithm that combines the advantages of both approaches.

Algorithm BA (see Figure 6.2) receives a problem p from a class of problems that has α-bisectors and a number N of processors as input. Its output is a set containing the N subproblems generated from p. If $N = 1$, the output is the singleton set $\{p\}$. If $N > 1$, the problem p is bisected into subproblems p_1 and p_2, and the N processors are partitioned between the two subproblems according to their relative weight. Subproblem p_i receives $N_i \geq 1$ processors, $i = 1, 2$, and $N_1 + N_2 = N$. Then, Algorithm BA is invoked recursively with input (p_i, N_i), $i = 1, 2$. Note that these recursive calls can be executed in parallel on different processors. The output is the union of the two sets of subproblems generated by the recursive calls. We observe that $1 \leq N_1 \leq N_2$ holds for each bisection step during the run of Algorithm BA if $\mathsf{w}(p_1) \leq \mathsf{w}(p_2)$. Furthermore, note that Algorithm BA does not require knowledge of the bisection parameter α of the given class of problems.

```
Algorithm BA(p, N)
begin
    if N > 1 then
        bisect p into p_1 and p_2;
        // assume w.l.o.g. w(p_1) ≤ w(p_2)
        α̂ := w(p_1)/w(p);
        if α̂N − ⌊α̂N⌋ ≤ α̂ then
            N_1 := ⌊α̂N⌋;
        else
            N_1 := ⌈α̂N⌉;
        fi;
        N_2 := N − N_1;
        return BA(p_1, N_1) ∪ BA(p_2, N_2);
    else
        return {p};
    fi;
end.
```

Figure 6.2: Algorithm BA (Best Approximation of ideal weight)

In each bisection step Algorithm BA chooses N_1 and N_2 such that the

maximum of $\mathsf{w}(p_i)/N_i$, $i = 1, 2$, is minimized. More precisely, the following minimization problem is solved:

$$\min_{\substack{N_1+N_2=N \\ N_1,N_2\geq 1}} \max\left\{\frac{\mathsf{w}(p_1)}{N_1}, \frac{\mathsf{w}(p_2)}{N_2}\right\}.$$

We assume w.l.o.g. that $\mathsf{w}(p_1) \leq \mathsf{w}(p_2)$ and therefore $\hat{\alpha} = \mathsf{w}(p_1)/\mathsf{w}(p)$ is the actual bisection parameter of that bisection. Clearly, the above minimization problem reduces to the decision whether $\hat{\alpha}N$ should be rounded up or down to obtain N_1. This can be done by checking if

$$\frac{\hat{\alpha}}{\lfloor \hat{\alpha}N \rfloor} \leq \frac{1-\hat{\alpha}}{N - \lceil \hat{\alpha}N \rceil}, \tag{6.2}$$

since $\mathsf{w}(p_1) = \hat{\alpha}\mathsf{w}(p)$, $\mathsf{w}(p_2) = (1-\hat{\alpha})\mathsf{w}(p)$. If $\hat{\alpha}N$ is an integer, (6.2) holds with equality. Otherwise let $\lceil \hat{\alpha}N \rceil = \hat{\alpha}N + u$, $\lfloor \hat{\alpha}N \rfloor = \hat{\alpha}N - d$, with $0 < d, u < 1$, $u + d = 1$. It is easy to see that (6.2) holds iff $d \leq \hat{\alpha}$.

Next, we will prove a series of lemmata in order to obtain a performance guarantee for Algorithm BA. The first lemma gives an upper bound on the rounding error made by Algorithm BA in a single bisection step.

Lemma 6.4 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ that has α-bisectors. Given a problem $p \in \mathcal{P}$ and an integer $N \geq 2$, it holds for each bisection step made by Algorithm BA:*

$$\max\left\{\frac{\mathsf{w}(p_1)}{N_1}, \frac{\mathsf{w}(p_2)}{N_2}\right\} \leq \frac{\mathsf{w}(p)}{N} \cdot \frac{N}{N-1},$$

where $N_i \geq 1$ is the number of processors assigned to subproblem p_i, $i = 1, 2$, by Algorithm BA.

Proof: We assume w.l.o.g. that $\mathsf{w}(p_1) \leq \mathsf{w}(p_2)$ and let d, u be defined as above. We have to distinguish two cases:

Case 1: $d \leq \hat{\alpha}$. Then $N_1 = \lfloor \hat{\alpha}N \rfloor$, and

$$\begin{aligned}
\max\left\{\frac{\mathsf{w}(p_1)}{N_1}, \frac{\mathsf{w}(p_2)}{N_2}\right\} &= \frac{\mathsf{w}(p_1)}{N_1} \\
&= \frac{\mathsf{w}(p)}{N} \cdot \frac{\hat{\alpha}N}{\hat{\alpha}N - d} \\
&\leq \frac{\mathsf{w}(p)}{N} \cdot \frac{\hat{\alpha}N}{\hat{\alpha}N - \hat{\alpha}}
\end{aligned}$$

$$= \frac{\mathsf{w}(p)}{N} \cdot \frac{N}{N-1}.$$

Case 2: $d > \hat{\alpha}$. Then $N_1 = \lceil \hat{\alpha} N \rceil$, $u < (1-\hat{\alpha})$, and

$$\begin{aligned}
\max\left\{\frac{\mathsf{w}(p_1)}{N_1}, \frac{\mathsf{w}(p_2)}{N_2}\right\} &= \frac{\mathsf{w}(p_2)}{N_2} \\
&= \frac{\mathsf{w}(p)}{N} \cdot \frac{(1-\hat{\alpha})N}{N-(\hat{\alpha}N+u)} \\
&< \frac{\mathsf{w}(p)}{N} \cdot \frac{(1-\hat{\alpha})N}{N-\hat{\alpha}N-(1-\hat{\alpha})} \\
&= \frac{\mathsf{w}(p)}{N} \cdot \frac{N}{N-1}.
\end{aligned}$$

□

Now we analyze the ratio between the maximum weight subproblem produced by Algorithm BA and the ideal weight for small numbers of processors.

Lemma 6.5 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ that has α-bisectors. Given a problem $p \in \mathcal{P}$ and a positive integer $N \leq 1/\alpha$, Algorithm BA uses $N-1$ bisections to partition p into N subproblems $p_1, \ldots, p_N$ such that*

$$\max_{1 \leq i \leq N} \mathsf{w}(p_i) \leq \mathsf{w}(p)(1-\alpha)^{\lfloor \frac{N}{2} \rfloor}.$$

Proof: It is easily seen that Algorithm BA uses $N-1$ bisections to partition p into N subproblems for any positive integer N. Now we show that the above inequality regarding the maximum weight among these subproblems holds.

Consider a path $Q = (q_0, q_1, \ldots, q_k)$, $0 \leq k \leq N-1$, from the root $p =: q_0$ of the associated bisection tree T_p^N to some leaf $p_i =: q_k$, $i \in \{1, \ldots, N\}$. Let N_j denote the number of processors assigned by Algorithm BA to problem q_j, $0 \leq j \leq k$. Using Lemma 6.4 we conclude:

$$\mathsf{w}(p_i) \leq \frac{\mathsf{w}(p)}{N} \cdot \prod_{j=0}^{k-1} \frac{N_j}{N_j - 1}.$$

For fixed k, $\prod_{j=0}^{k-1} \frac{N_j}{N_j-1}$ is maximized if each factor is as large as possible. Since $N_{k-1} \geq 2$ and $N_j \geq N_{j+1}+1$ it follows that $\frac{N_j}{N_j-1} \leq \frac{2+(k-1)-j}{1+(k-1)-j}$. Therefore,

$$\mathsf{w}(p_i) \leq \frac{\mathsf{w}(p)}{N}(k+1).$$

Since there are k bisection steps along Q we have:

$$\mathsf{w}(p_i) \leq \mathsf{w}(p)(1-\alpha)^k.$$

Thus,

$$\max_{1\leq i\leq N} \mathsf{w}(p_i) \leq \frac{\mathsf{w}(p)}{N} \cdot \max_{0\leq k\leq N-1} \min\{k+1, N(1-\alpha)^k\}.$$

It remains to show that $\lfloor \frac{N}{2} \rfloor \leq N(1-\alpha)^{\lfloor \frac{N}{2} \rfloor}$. The case $N = 1$ is trivial. If $N \geq 2$, it holds that

$$N(1-\alpha)^{\lfloor \frac{N}{2} \rfloor} \geq N\left(1-\frac{1}{N}\right)^{\frac{N}{2}} \geq \frac{N}{2},$$

as $1/N \geq \alpha$. □

The following lemma provides an upper bound on the increase of the average weight per processor of a subproblem that has been assigned a certain number of processors by Algorithm BA.

Lemma 6.6 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ that has α-bisectors. Let further be $p \in \mathcal{P}$, $\rho \geq \alpha$, and $N \in \mathbb{N}$. Then, for any subproblem $\hat{p}$ generated by Algorithm BA on input (p, N) that has been assigned $\hat{N} \geq \rho/\alpha + 1$ processors it holds that:*

$$\frac{\mathsf{w}(\hat{p})}{\hat{N}} \leq \frac{\mathsf{w}(p)}{N} \cdot e^{(1-\alpha)/\rho}.$$

Proof: Similarly to the proof of Lemma 6.5 we consider the path $Q = (q_0, q_1, \ldots, q_l)$ from the root $p =: q_0$ of the bisection tree T_p^N to $\hat{p} =: q_l$. With Lemma 6.4 we have:

$$\begin{aligned} \frac{\mathsf{w}(\hat{p})}{\hat{N}} &\leq \frac{\mathsf{w}(p)}{N} \cdot \prod_{j=0}^{l-1} \frac{N_j}{N_j - 1} \\ &= \frac{\mathsf{w}(p)}{N} \cdot \prod_{j=0}^{l-1} \left(1 + \frac{1}{N_j - 1}\right), \end{aligned} \tag{6.3}$$

where N_j is the number of processors assigned to subproblem q_j, $0 \leq j \leq l$. To bound the product $\prod_{j=0}^{l-1}(1 + 1/(N_j - 1))$ we show that for $0 \leq j \leq l-1$:

$$\frac{1}{N_j - 1} \leq (1-\alpha)\frac{1}{N_{j+1} - 1}. \tag{6.4}$$

In order to derive (6.4) we have to prove $\frac{N_{j+1}-1}{N_j-1} \leq 1-\alpha$ for $0 \leq j \leq l-1$. If $N_{j+1} = N_j - \lfloor \hat{\alpha}_j N_j \rfloor$, where $\hat{\alpha}_j$ is the actual bisection parameter of the $(j+1)$-th bisection on Q, we conclude

$$\begin{aligned}
\frac{N_{j+1}-1}{N_j-1} &= \frac{N_j - \hat{\alpha}_j N_j + d_j - 1}{N_j - 1} \\
&\leq \frac{N_j - \hat{\alpha}_j N_j + \hat{\alpha}_j - 1}{N_j - 1} \\
&= 1 - \hat{\alpha}_j \\
&\leq 1 - \alpha.
\end{aligned}$$

The remaining cases can be proven analogously.

Note that (6.4) implies $\frac{1}{N_j-1} \leq (1-\alpha)^{l-j}\frac{1}{N_l-1}$. Therefore,

$$\begin{aligned}
\prod_{j=0}^{l-1}\left(1+\frac{1}{N_j-1}\right) &\leq \prod_{j=0}^{l-1}\left(1+(1-\alpha)^{l-j}\frac{1}{N_l-1}\right) \\
&\leq \prod_{j=0}^{l-1} e^{(N_l-1)^{-1}(1-\alpha)^{l-j}} \\
&= e^{(N_l-1)^{-1}\sum_{j=0}^{l-1}(1-\alpha)^{l-j}} \\
&\leq e^{(N_l-1)^{-1}\sum_{j=1}^{\infty}(1-\alpha)^{j}} \\
&= e^{(N_l-1)^{-1}(1-\alpha)/\alpha} \\
&\leq e^{\alpha/\rho\cdot(1-\alpha)/\alpha} \\
&= e^{(1-\alpha)/\rho}.
\end{aligned}$$

The claim of the lemma now follows from (6.3). □

We are now in a position to bound the ratio between the maximum weight subproblem produced by Algorithm BA and the ideal weight by an appropriate combination of Lemmata 6.5 and 6.6.

Theorem 6.7 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ that has α-bisectors. Given a problem $p \in \mathcal{P}$ and a positive integer N, Algorithm BA partitions p into N subproblems $p_1, \ldots, p_N$ such that*

$$\max_{1\leq i\leq N} \mathsf{w}(p_i) \leq \frac{\mathsf{w}(p)}{N} \cdot e \cdot \left\lfloor \frac{1}{\alpha} \right\rfloor \cdot (1-\alpha)^{\lfloor \frac{1}{2\alpha} \rfloor - 1}.$$

Proof: If $N \leq 1/\alpha$, the theorem follows from Lemma 6.5. Therefore, we assume $N > 1/\alpha$ for the remainder of this proof. Again, we consider a path

$Q = (q_0, q_1, \ldots, q_k)$ from the root $p =: q_0$ of the bisection tree T_p^N to some leaf $p_i =: q_k$, $i \in \{1, \ldots, N\}$. Let q_{l+1} be the first node on this path (i.e., the node with minimum index) such that $N_{l+1} \leq \frac{1}{\alpha}$, where N_j is the number of processors assigned to subproblem q_j, $0 \leq j \leq k$. Using Lemma 6.5 we conclude:

$$\mathsf{w}(p_i) \leq \frac{\mathsf{w}(q_{l+1})}{N_{l+1}} \cdot \left\lfloor \frac{1}{\alpha} \right\rfloor \cdot (1-\alpha)^{\lfloor \frac{1}{2\alpha} \rfloor}. \tag{6.5}$$

Since $N_l > 1/\alpha$ we can apply Lemma 6.6 for q_l with $\rho = 1 - \alpha$. This yields:

$$\frac{\mathsf{w}(q_l)}{N_l} \leq \frac{\mathsf{w}(p)}{N} \cdot e. \tag{6.6}$$

It remains to consider the bisection step at subproblem q_l. By Lemma 6.4 and from $N_l > 1/\alpha$ we derive:

$$\frac{\mathsf{w}(q_{l+1})}{N_{l+1}} \leq \frac{\mathsf{w}(q_l)}{N_l} \cdot \frac{N_l}{N_l - 1} \leq \frac{\mathsf{w}(q_l)}{N_l} \cdot \frac{1}{1-\alpha}. \tag{6.7}$$

Combining (6.5), (6.7), and (6.6) completes the proof because the bounds are valid for any root-to-leaf path. □

Now we analyze the running time of Algorithm BA. The management of free processors for Algorithm BA is very simple and does not introduce any communication overhead (see Section 6.1.4). Therefore, the running time of Algorithm BA in our model is obviously bounded by the maximum depth of a node in the bisection tree representing the run of Algorithm BA, because in every step of the algorithm the internal nodes at one level of the bisection tree are bisected in parallel.

Consider a problem p that is to be split among N_p processors and that is bisected into p_1 and p_2 by Algorithm BA. Inequality (6.4) from the proof of Lemma 6.6 shows that each of p_1 and p_2 receives at most $N_p(1-\alpha) + \alpha$ processors. As $N_p(1-\alpha) + \alpha \leq N_p(1 - \alpha/2)$ for $N_p \geq 2$, the number of processors is at least reduced by a factor of $(1 - \alpha/2)$ in each bisection step, and thus the depth of a leaf in the bisection tree can be at most $\log_{\frac{1}{1-\alpha/2}} N$. Thus, we have the following theorem:

Theorem 6.8 *Given a class $\mathcal{P}$ of problems that has α-bisectors for a fixed constant α, Algorithm BA subdivides a problem from $\mathcal{P}$ into N subproblems in time $O(\log N)$.*

6.1.3 Combining BA and HF: Algorithm BA-HF

The worst-case upper-bound on the ratio between the maximum weight subproblem produced by Algorithm BA and the ideal weight $w(p)/N$, which is stated in Theorem 6.7, is not as good as the one for Algorithm HF. To obtain a highly parallel algorithm for our load balancing problem with a performance guarantee very close to the one for Algorithm HF we integrate Algorithm BA and Algorithm HF by dividing the bisection process into two phases. While the number of processors available for a particular subproblem is large enough, Algorithm BA-HF (see Figure 6.3) acts like

```
Algorithm BA-HF(p, N)
begin
    if N ≥ σ/α + 1 then
        bisect p into p1 and p2;
        // assume w.l.o.g. w(p1) ≤ w(p2)
        α̂ := w(p1)/N;
        if α̂N − ⌊α̂N⌋ ≤ α̂ then
            N1 := ⌊α̂N⌋;
        else
            N1 := ⌈α̂N⌉;
        fi;
        N2 := N − N1;
        return BA-HF(p1, N1) ∪ BA-HF(p2, N2);
    else
        return (P)HF(p,N);
    fi;
end.
```

Figure 6.3: Algorithm BA-HF

Algorithm BA. However, if the number of processors assigned to a subproblem is below a certain threshold, Algorithm HF is used to partition this subproblem further. To define this threshold precisely, we assume that Algorithm BA-HF has knowledge about the bisection parameter α of a given problem class. Algorithm HF is invoked by Algorithm BA-HF if $N < \sigma/\alpha + 1$, where $\sigma \in \mathbb{R}^+$ is a parameter predefined by the application to reach the desired performance guarantee. (If α is not known, Algorithm BA-HF can still be used after setting the threshold for N directly. In this case, however, it is not possible to ensure a performance guarantee better than that for Algorithm BA.) Note that the two phases

of Algorithm BA-HF are not completely separate; while some processors are still assigning processors to subproblems according to Algorithm BA, others may already have switched to Algorithm HF.

Depending on the value of σ/α, it may be advantageous to choose either the sequential Algorithm HF or Algorithm PHF for the implementation of the second phase of Algorithm BA-HF: If σ/α is very small, the simple sequential implementation of Algorithm HF may be perfectly sufficient; if σ/α is somewhat larger, it may be beneficial to employ Algorithm PHF. The discussion of this trade-off depends on the particular parallel architecture used.

The next theorem gives a bound on the quality of the load balancing achieved by Algorithm BA-HF.

Theorem 6.9 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ that has α-bisectors. Given a problem $p \in \mathcal{P}$, a positive integer N, $\alpha \in (0, 1/2]$, and $\sigma \geq \alpha$, Algorithm BA-HF partitions p into N subproblems $p_1, \ldots, p_N$ such that*

$$\max_{1 \leq i \leq N} \mathsf{w}(p_i) \leq \frac{\mathsf{w}(p)}{N} \cdot e^{(1-\alpha)/\sigma} \cdot \left(1 + \frac{\alpha}{\sigma}\right) \cdot r_\alpha \,.$$

Proof: The proof is analogous to the proof of Theorem 6.7. □

According to this theorem we can bring the worst-case bound of Algorithm BA-HF on the ratio between $\max_{1 \leq i \leq N} \mathsf{w}(p_i)$ and $\frac{\mathsf{w}(p)}{N}$ arbitrarily close to the corresponding bound for Algorithm HF. For any $\epsilon > 0$, if we let $\sigma \geq 1/\ln(1+\epsilon)$, the performance guarantee of Algorithm BA-HF is increased at most by a factor of $1 + \epsilon$ in comparison to Algorithm HF.

Regarding the running time, it is clear that the first phase of Algorithm BA-HF is at most as long as the running time for Algorithm BA and can thus be bounded by $O(\log N)$ for fixed α. If σ and, therefore, σ/α are considered constants as well, the second phase of Algorithm BA-HF requires only constant additional work per processor, no matter whether the sequential Algorithm HF or Algorithm PHF is used. In this case, the overall running time for Algorithm BA-HF is $O(\log N)$. If σ is allowed to be arbitrarily large, it is necessary to use Algorithm PHF in the second phase of Algorithm BA-HF in order to achieve running time $O(\log N)$, because if the sequential Algorithm HF were used, the running time would be $O(\log N + \sigma/\alpha)$. Using Algorithm PHF in the second phase of Algorithm BA-HF, we get the following theorem.

Theorem 6.10 *Given a class $\mathcal{P}$ of problems that has α-bisectors for a fixed constant α, Algorithm BA-HF subdivides a problem from $\mathcal{P}$ into N subproblems in time $O(\log N)$.*

6.1.4 Managing the Free Processors

In the previous sections we have assumed that a processor that wants to send a newly generated subproblem to a free processor can quickly acquire the id of a free processor. Now we investigate how difficult the realization of this access to free processors is in the context of the different load balancing algorithms.

For Algorithm PHF, the problem of managing the free processors is the most challenging. In the first phase, it can be the case that a large number of processors bisect problems in parallel simultaneously and need to get access to a free processor in order to send a newly generated subproblem to it. Each sender must get the id of a different free processor. Basically, this task can be viewed as mapping a dynamically growing tree onto the processors of the parallel architecture [Lei92, pp. 410–430]. Depending on the machine model, various solutions employing distributed data structures for managing the free processor may be applicable: (randomized) work stealing [BL94], dynamic embeddings [Lei92, LNRS92, AL91, Heu96], etc. In the following, we outline a solution that employs a modified version of Algorithm BA as a subroutine.

Let Algorithm $\overline{\text{BA}}$ be an algorithm that is identical to Algorithm BA except that it does not bisect any subproblems with weight at most $\frac{w(p)}{N} \cdot r_\alpha$. The first part of phase one of Algorithm PHF consists of an execution of Algorithm $\overline{\text{BA}}$ on inputs p and N. Theorem 6.8 implies that this execution takes time $O(\log N)$. At the end of this execution, only subproblems of weight greater than $\frac{w(p)}{N} \cdot r_\alpha$ have been bisected. Furthermore, no remaining subproblem can be heavier than $\frac{w(p)}{N} \cdot e \cdot \lfloor \frac{1}{\alpha} \rfloor \cdot (1-\alpha)^{\lfloor \frac{1}{2\alpha} \rfloor - 1}$. This follows from Theorem 6.7, because the only subproblems of weight greater than $\frac{w(p)}{N} \cdot r_\alpha$ that are not bisected by Algorithm $\overline{\text{BA}}$ are those that have been assigned to a single processor, and these subproblems would not have been bisected by Algorithm BA either.

The second (and last) part of phase one of Algorithm PHF is very similar to phase two and consists of a constant number of iterations (for fixed α) in each of which all remaining subproblems with weight

greater than $\frac{w(p)}{N} \cdot r_\alpha$ are bisected. Each such iteration can be implemented in time $O(\log N)$ by numbering the free processors, numbering all remaining subproblems with weight greater than $\frac{w(p)}{N} \cdot r_\alpha$, bisecting all these heavy subproblems, and sending one of the two new subproblems generated from the i-th heavy subproblem to the i-th free processor. As each iteration reduces the maximum weight among the remaining subproblems at least by a factor of $(1-\alpha)$, we can conclude from

$$\frac{w(p)}{N} \cdot r_\alpha \geq \frac{w(p)}{N} \cdot \left\lfloor \frac{1}{\alpha} \right\rfloor \cdot e \cdot (1-\alpha)^{\lfloor \frac{1}{2\alpha} \rfloor - 1} \cdot (1-\alpha)^{\lfloor \frac{1}{2\alpha} \rfloor + \log_{\frac{1}{1-\alpha}} e}$$

that $\lfloor \frac{1}{2\alpha} \rfloor + \lceil \log_{\frac{1}{1-\alpha}} e \rceil$ iterations suffice to reach a situation where all remaining subproblems have weight at most $\frac{w(p)}{N} \cdot r_\alpha$.

After the end of the first phase of Algorithm PHF, the remaining free processors are determined and assigned numbers in time $O(\log N)$. As explained in Section 6.1.1, this knowledge can be exploited during the second phase of Algorithm PHF as follows: a busy processor can locally determine the number of the free processor to which a newly generated subproblem must be sent, and a single request to another processor suffices to acquire the id of that free processor.

Although the solution outlined above shows that the management of free processors during phases one and two of Algorithm PHF can be implemented without making the asymptotic running time larger than $O(\log N)$, it is clear that substantial communication overhead is caused. Therefore, Algorithm PHF will perform best in practice if the network is very powerful or if the number of processors is not too large.

For Algorithm BA, the management of the free processors can be done in a very simple and efficient way. With each subproblem q, we simply store the range $[i, j]$ of processors available for subproblems resulting from q. Initially, the original problem p has the full range $[1, N]$ of processors available. When p is divided into p_1 and p_2 such that p_1 gets N_1 processors and p_2 gets N_2 processors, p_1 stays on P_1 and gets the range $[1, N_1]$, while p_2 is sent to P_{N_1+1} and gets the range $[N_1 + 1, N]$. Similarly, a problem q with range $[i, j]$ is always bisected at P_i, and the resulting subproblems q_1 and q_2 with corresponding numbers of processors N_1 and $N_2 = j + 1 - i - N_1$ are associated with ranges $[i, i + N_1 - 1]$ and $[i + N_1, j]$, and problem q_2 is sent to processor $i + N_1$. In this way, each processor can locally determine to which free processor it should send a newly generated subproblem, and no overhead is incurred for the management of free processors at all. This is one of the main advantages of Algorithm BA.

For Algorithm BA-HF, the simple management of free processors that is applicable for Algorithm BA can be used while the number of processors for a subproblem is large. For subproblems that are to be partitioned among a small number of processors ($N < \sigma/\alpha + 1$), the management of free processors is trivial if the sequential Algorithm HF is used. If Algorithm PHF is used in the second phase of Algorithm BA-HF, however, the more complicated management of free processors described above can be deployed. As this method with expensive communication is used only for small numbers of processors, the communication overhead should be small enough to achieve good running times in practice.

6.2 Simulation Results

To gain further insight about the balancing quality achieved by the proposed algorithms, we carried out a series of simulation experiments. The goal of this simulation study was to obtain information about the average-case behavior of Algorithms BA and BA-HF in comparison to Algorithm HF. Since Algorithm PHF produces the same partitioning as Algorithm HF, no separate experiments were conducted for Algorithm PHF. The following stochastic model for an average-case scenario that may arise from practical applications seems reasonable: Assume that the actual bisection parameter $\hat{\alpha}$ is drawn uniformly at random from the interval $[\alpha, \beta]$, $0 < \alpha \leq \beta \leq 1/2$, and that all $N - 1$ bisection steps are independent and identically distributed. We will write $\hat{\alpha} \sim U[\alpha, \beta]$ if $\hat{\alpha}$ has uniform distribution on $[\alpha, \beta]$. Such an assumption is valid, for example, if the problems are represented by lists of elements taken from an ordered set, and if a list is bisected by choosing a random pivot element and partitioning the list into those elements that are smaller than the pivot and those that are larger. Some of the experimental results for Algorithm HF (see below) were confirmed by thorough mathematical analysis in [BSS99].

We repeated each simulation experiment 1000 times to obtain data that is statistically meaningful. In each experiment the *ratio* between the maximum load generated by our algorithms and the uniform load distribution was recorded. The main focus of interest was on the sample mean of the observed ratios for all three algorithms, but also the maximum and minimum ratio, as well as the sample variance was computed. Algorithm BA-HF is identical to Algorithm HF if $N < \sigma/\alpha + 1$. Consequently, the entry for Algorithm BA-HF is omitted in our plots and tables if the initial number of processors is smaller than the threshold.

Simulations of this stochastic model for several choices of the interval $[\alpha, \beta]$ and $N = 2^k$, $k \in \{5, 6, \ldots, 20\}$, were performed. We chose the number of processors as consecutive powers of 2 to explore the asymptotic behavior of our load balancing algorithms. The reader should therefore bear in mind that all but one of the plots are to a logarithmic scale. Choosing N as a power of 2 is possible since the results shown depend smoothly on N under the above stochastic assumptions if the interval $[\alpha, \beta]$ is not too small. Figure 6.5 shows the full range from $N = 32$ to $N = 64$ processors for $\hat{\alpha} \sim U[0.1, 0.5]$, $\sigma = 1.0$, to provide a detailed view of the results shown in Figure 6.4. The influence of the threshold parameter σ on the performance of Algorithm BA-HF in our stochastic model was also studied.

Table 6.1: Sample variance of some experiments ($\sigma = 1.0$)

$\log N$	$\hat{\alpha} \sim U[0.1, 0.5]$			$\hat{\alpha} \sim U[0.01, 0.25]$		
	BA	BA-HF	HF	BA	BA-HF	HF
5	0.093	0.045	0.011	0.482	0.482	0.062
10	0.048	0.032	0.000	0.321	0.194	0.003
15	0.030	0.022	0.000	0.194	0.186	0.000
20	0.020	0.015	0.000	0.163	0.147	0.000

It is remarkable that the sample variance was very small (see Table 6.1) in almost all cases. Only if an interval $[\alpha, 2\alpha]$ with very small α was chosen, a significantly higher sample variance was observed. This is at least partially due to the much higher ratios that result from permanent "bad" bisections (see Figure 6.10). Even more astonishingly the outcome of each experiment was fairly close to the sample mean of all 1000 experiments in every simulation. Especially for Algorithm HF the observed ratios were sharply concentrated around the sample mean for larger values of N (see Table 6.2). Motivated by these observations, we analyze the average-case behavior of Algorithm HF under the above stochastic assumptions in the next chapter.

In all experiments Algorithm HF performed best and Algorithm BA-HF outperformed Algorithm BA. For all but one choice of $[\alpha, \beta]$ the observed ratios differ at most by a factor 3 for fixed N. The performance gap increases with the number of available processors. This is due to the fact that the performance of Algorithm BA (and therefore also of Algorithm BA-HF during its first phase) suffers from accumulated rounding errors when the number of processors grows. However, it follows from Lemma 6.5 that this process converges as N approaches

infinity. We have shown in the previous sections that all three algorithms have worst-case upper bounds on the ratio that depend solely on α. Therefore, for fixed α, the ratio generated on any input by our load balancing algorithms is bounded from above by a constant. In each of our simulations the average and even the maximum ratio were substantially lower than the corresponding worst-case bound (see Table 6.2). These bounds were calculated according to Lemma 6.5, Theorems 6.7, 6.9, 5.2, and 5.7. This indicates that the performance of the proposed load balancing algorithms for practical applications will most likely be significantly better than the worst-case guarantees.

Table 6.2: Comparison of the worst-case upper bounds and the observed minimum, average, and maximum ratios for $\hat{\alpha} \sim U[0.01, 0.5]$, $\sigma = 1.0$

BA				
$\log N$	ub	min	avg	max
5	27.25	1.70	2.73	4.98
10	166.12	3.14	4.01	5.80
15	166.12	4.31	5.04	6.47
20	166.12	5.27	6.03	8.16
BA-HF				
$\log N$	ub	min	avg	max
10	101.51	1.99	2.27	3.40
15	101.51	2.40	2.92	4.72
20	101.51	3.16	3.88	5.89
HF				
$\log N$	ub	min	avg	max
5	23.43	1.55	1.94	2.63
10	37.35	1.87	1.96	2.08
15	37.35	1.94	1.96	1.98
20	37.35	1.96	1.96	1.97

Figures 6.4–6.8 reveal that the average ratio produced by Algorithm HF is almost constant for the whole range of $N = 32$ to $N = 2^{20} = 1048576$ processors and depends only on the particular choice of the interval $[\alpha, \beta]$. This is due to the length of the interval from which the actual bisection parameter is drawn which is at least 0.15 in each simulation setup. Therefore, bisections with small $\hat{\alpha}$ are compensated by bisections that generate subproblems with roughly equal weight. We observed that the distribution

of the weights of the subproblems generated by Algorithm HF is highly regular in these cases: If the subproblems were sorted by non-increasing weight after a sufficiently large numbers of bisections the difference of two consecutive weights in this order was almost constant. This phenomenon allows for a steady reduction of the maximum weight by the selection rule employed by Algorithm HF in consecutive bisection steps. Only when the range for the actual bisection parameter is very small (cf. Figures 6.9–6.11) the observed ratios change with varying numbers of processors. This results from the existence of many subproblems whose weights are close to the maximum weight at certain stages during the run of Algorithm HF.

It is intuitively clear that no bisection-based load balancing algorithm is able to perform well (in comparison with the ideal uniform load distribution) when each bisection step has the least possible $\hat{\alpha}$. Figure 6.11 shows a comparison of Algorithm BA and Algorithm HF for $\hat{\alpha} \equiv 0.1$. This computation was done only once since there are no random choices in this case. It turns out that the resulting ratios are separated by the worst-case upper bound $r_{0.1} < 4.31$ of Algorithm HF. Therefore, Algorithm BA is inferior to Algorithm HF from a worst-case point of view. It is also worth noticing that the ratios generated by Algorithm BA are way below the upper bound provided by Theorem 6.7 for the numbers of processors under consideration. This might indicate that this upper bound is not tight.

Finally we studied the influence of the threshold parameter σ on the average-case performance of Algorithm BA-HF. Figure 6.12 shows that the improvement of the average ratio is approximately 10% when σ increases from 1.0 to 2.0 and another 5% when $\sigma = 3.0$. Therefore, we can expect a sufficient balancing quality from Algorithm BA-HF using relatively small values of σ. This ensures that the inherent parallelism of Algorithm BA can be almost fully exploited during the first phase of Algorithm BA-HF and Algorithm HF (or PHF) is only used on a small number of processors during the second phase.

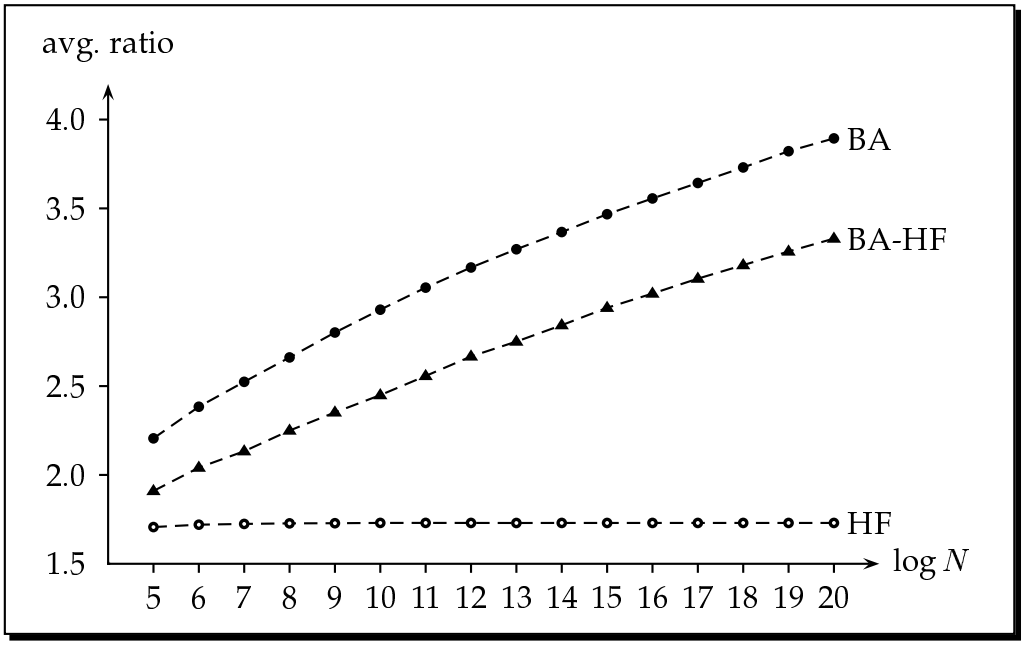

Figure 6.4: Comparison of the average ratio for $\hat{\alpha} \sim U[0.1, 0.5]$, $\sigma = 1.0$

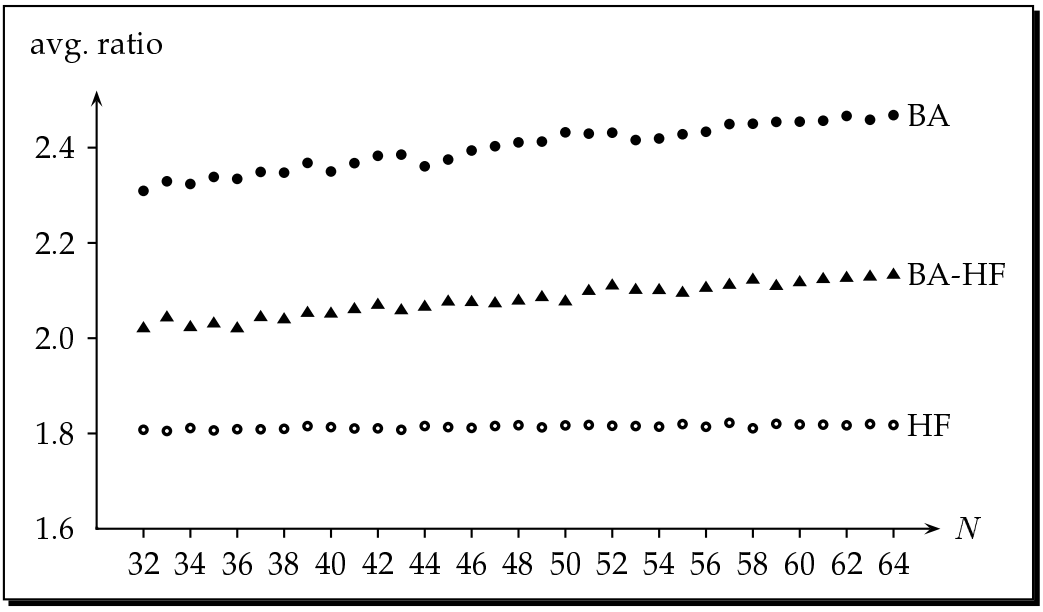

Figure 6.5: Influence of the number of processors
($\hat{\alpha} \sim U[0.1, 0.5]$, $\sigma = 1.0$)

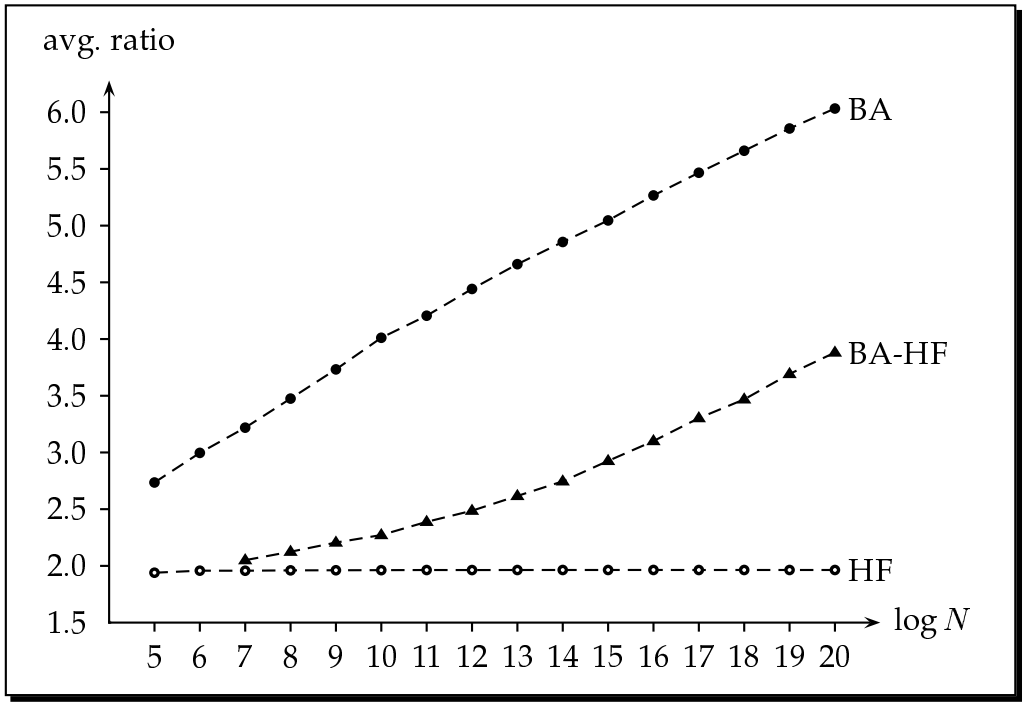

Figure 6.6: Comparison of the average ratio for $\hat{\alpha} \sim U[0.01, 0.5]$, $\sigma = 1.0$

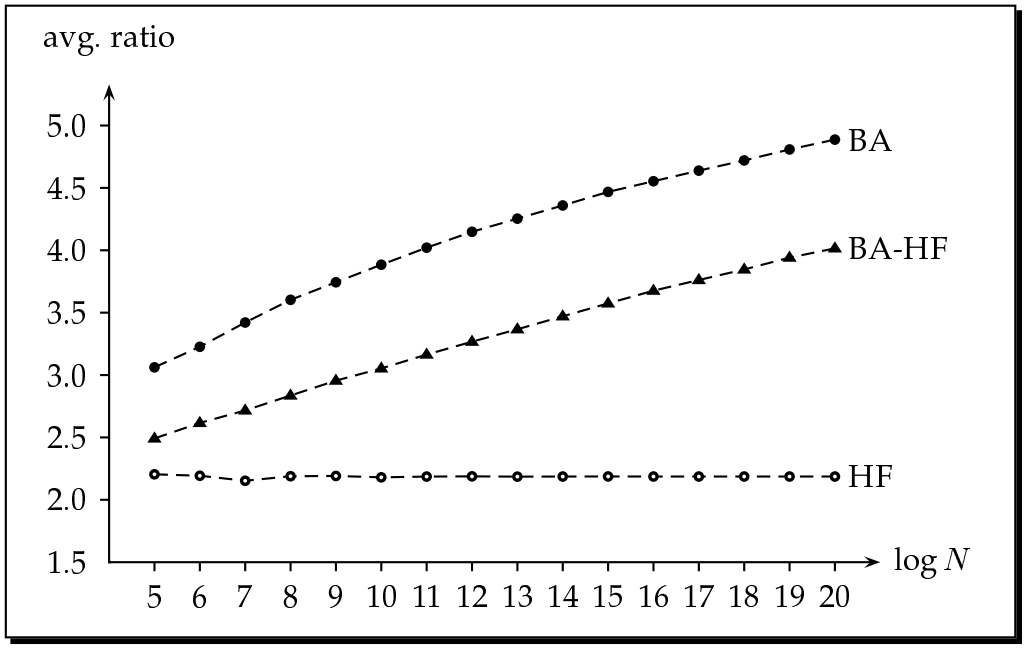

Figure 6.7: Comparison of the average ratio for $\hat{\alpha} \sim U[0.1, 0.25]$, $\sigma = 1.0$

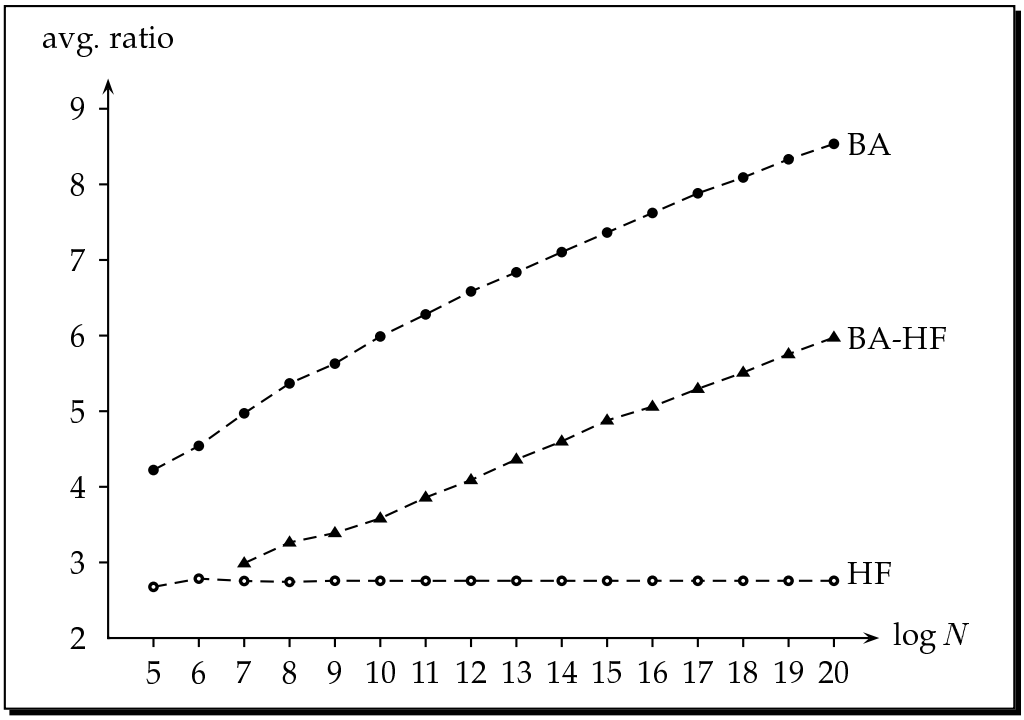

Figure 6.8: Comparison of the average ratio for $\hat{\alpha} \sim U[0.01, 0.25]$, $\sigma = 1.0$

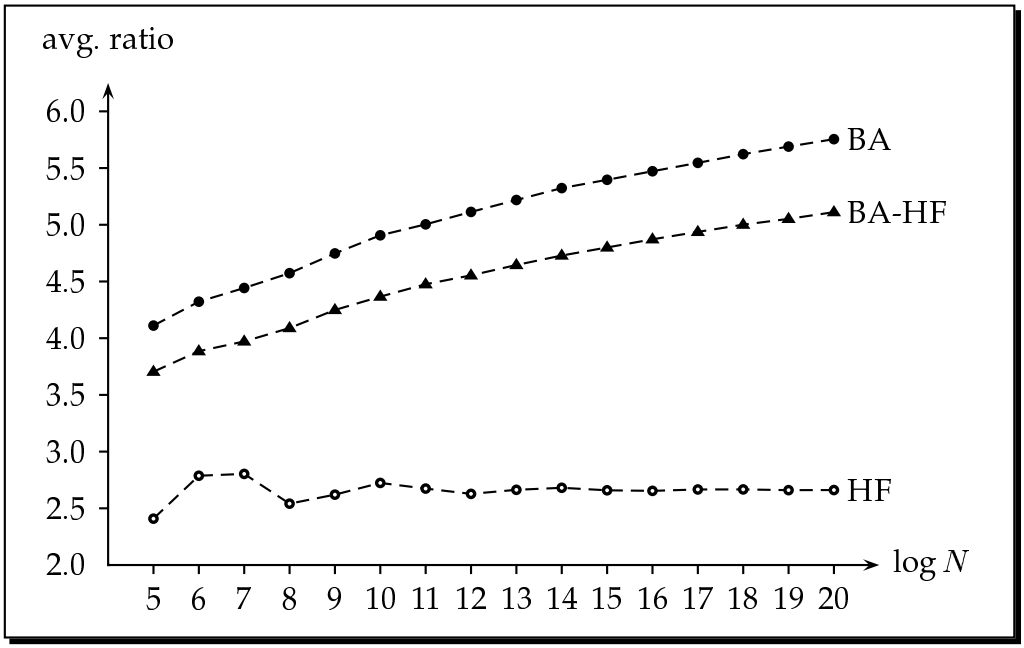

Figure 6.9: Comparison of the average ratio for $\hat{\alpha} \sim U[0.1, 0.15]$, $\sigma = 1.0$

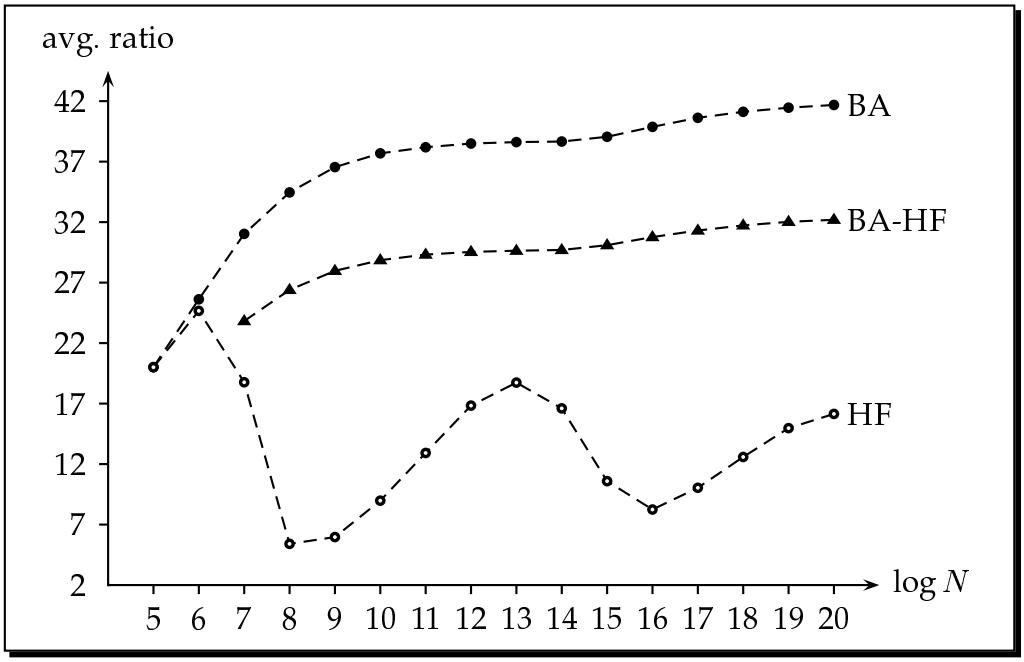

Figure 6.10: Comparison of the average ratio for $\hat{\alpha} \sim U[0.01, 0.02]$, $\sigma = 1.0$

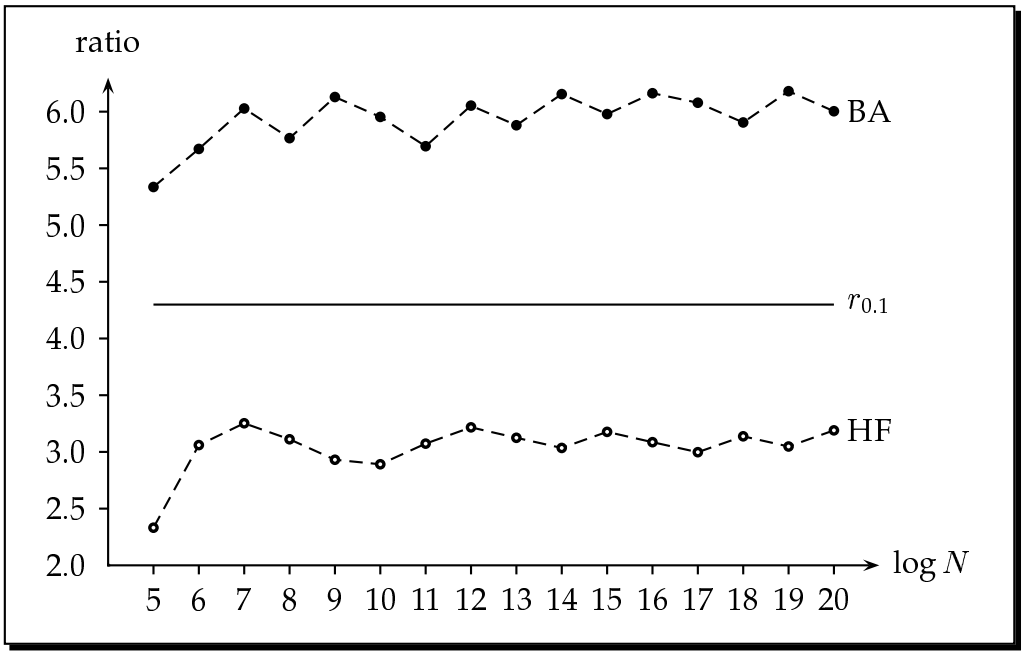

Figure 6.11: Comparison of the ratio generated by Algorithm BA and Algorithm HF for $\hat{\alpha} \equiv 0.1$

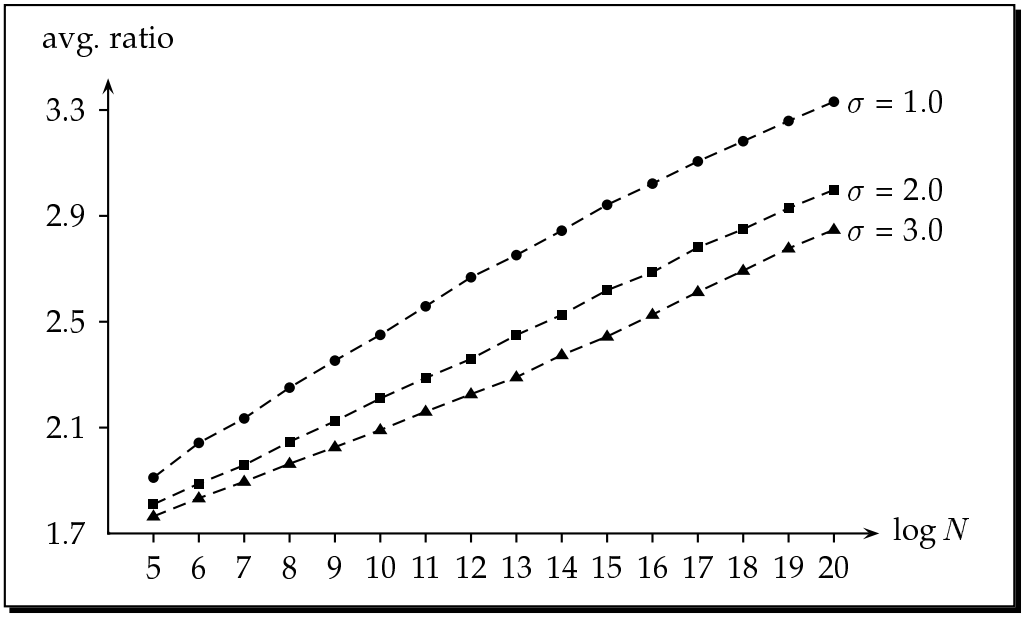

Figure 6.12: Influence of the threshold parameter σ on the average ratio of Algorithm BA-HF for $\hat{\alpha} \sim U[0.1, 0.5]$, $\sigma \in \{1.0, 2.0, 3.0\}$

Chapter 7

Average-Case Analysis of Load Balancing using Bisectors

The upper bounds on the maximum load produced by Algorithm HF given in Section 5.1 ensure an acceptable balancing quality even in the worst case for an impressive range of the actual bisection parameter. However, if the left boundary of this range gets too close to zero, the worst-case bounds increase dramatically. But in our lower bound construction in Section 5.1, a long sequence of "bad" bisections was necessary to come close to the worst case. Therefore, one might expect that Algorithm HF (or some other bisection-based load balancing algorithm for our model) performs much better *on average* than the worst-case upper bounds indicate. We show in this chapter that this is indeed the case assuming a straight-forward and rather pessimistic random distribution for the individual bisection steps.

Recall the stochastic model that we used to obtain the simulation results for our load balancing algorithms (see Section 6.2): The actual bisection parameter is drawn uniformly at random from the interval $[\alpha, \beta]$, $0 \leq \alpha \leq \beta \leq 1/2$, and all $N - 1$ bisection steps are independent and identically distributed. In this chapter, we study this average-case scenario for $\alpha = 0, \beta = 1/2$ and $N \to \infty$. We will show that in this case the maximum load generated by Algorithm HF is sharply concentrated around two times the ideal load (uniform partition).

The remainder of this chapter is structured as follows. In Section 7.1 we state some necessary definitions and give a brief outline of our analysis. Section 7.2 presents a formula for the expected number of problems

in the bisection tree whose weights exceed a given value although the bisection process runs forever. Furthermore, we show in Section 7.3 that the outcome of this stochastic process is sharply concentrated around its expectation using AZUMA's tail bound for martingales. Then, the results concerning the average-case behavior of Algorithm HF are derived. Finally, in Section 7.4 we describe how these results can be used to reduce the computational complexity of our sequential and parallel load balancing algorithms if the above stochastic assumptions hold.

7.1 Definitions and Outline of the Analysis

For our analysis we assume that Algorithm HF receives a problem p and executes infinitely many iterations of the while-loop (cf. Figure 5.1 on page 77). We consider the *infinite bisection tree (IBT)* T_p^∞ generated by this process. This tree grows larger with each iteration of Algorithm HF. There is a one-to-one mapping from the subproblems produced by Algorithm HF to the nodes in the IBT. At every point in time the subproblems in P correspond to the leaves in the part of T_p^∞ which has been generated so far. Therefore, N now counts the number of subproblems generated so far and is no longer a fixed input parameter of Algorithm HF. One can also imagine that the IBT exists a priori and Algorithm HF visits all nodes one by one. When we say that Algorithm HF visits or expands a node in the IBT, we adopt this view on the model.

Now we define our probability space formally: Let $\Omega := [0,1]^{\mathbb{N}}$ with the uniform distribution. As usual in the continuous case, the set of events is restricted to the σ-field $\mathcal{F} := \overline{\mathcal{B}}^{\mathbb{N}}$ where

$$\overline{\mathcal{B}} := \{\, I \cap [0,1] \mid I \in \mathcal{B} \,\},$$

and $\mathcal{B}$ denotes the Borel sets over $\mathbb{R}$.

An element $(\hat{\alpha}_1, \hat{\alpha}_2, \dots) \in \Omega$ has the following interpretation: $\hat{\alpha}_i$ corresponds to the relative weight of the, say, left successor of the node which is split by the i-th bisection. We do not consider the actual bisection parameter (which is equal to $\hat{\alpha}_i$ or $(1 - \hat{\alpha}_i)$), since studying $U[0,1]$ instead of $U[0, 0.5]$ simplifies some calculations, and it is in most cases irrelevant for our arguments which successor node is heavier. Consequently, we call $\hat{\alpha}_i$ the *generalized actual bisection parameter of bisection* i. The following definition states this concept in terms of classes of problems:

Definition 7.1 *A class $\mathcal{P}$ of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ has* uniform bisectors *if every problem $p \in \mathcal{P}$ can efficiently be divided into two problems $p_1, p_2 \in \mathcal{P}$ with $\mathsf{w}(p_1) + \mathsf{w}(p_2) = \mathsf{w}(p)$ and $\mathsf{w}(p_1)/\mathsf{w}(p) \sim U[0,1]$.*

Note that we could use $(0,1)^{\mathbb{N}}$ instead of $[0,1]^{\mathbb{N}}$ as the sample space throughout our analysis without any modifications in the proofs. However, since $\hat{\alpha} = 0$ might result from a randomized bisection subroutine, we decided to use the more general formulation.

We call a node v in an IBT *d-heavy*, iff $\mathsf{w}(v) \geq d$ and accordingly we use the notion *d-light*. If the value of d is obvious from the context, we just say *heavy* and *light* for short.

Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ that has uniform bisectors, and let C denote the complete binary tree of infinite height. Given a problem $p \in \mathcal{P}$ and a number $d \in \mathbb{R}^+$, we define for each node $v \in C$ the random variables

$$H_v^d := \begin{cases} 1 & \text{if } \mathsf{w}(v) \geq d, \\ 0 & \text{otherwise.} \end{cases}$$

Since C is a superset of the nodes in any IBT T_p^∞, $H^d := \sum_{v \in C} H_v^d$ counts the number of d-heavy nodes in T_p^∞. If a node v is not contained in T_p^∞, $H_v^d = 0$. The random variables H^d directly correspond to the performance of Algorithm HF, since H^d equals the number of iterations after which the set P of subproblems generated by Algorithm HF contains only d-light nodes. This is due to the fact that Algorithm HF visits all d-heavy nodes before it expands any d-light node.

For brevity, we denote by W the weight of the initial problem p. Finally, for $v \in T_p^\infty$, $v \neq p$, we define X_v to be the weight of v relative to its parent u in the tree, i.e., $X_v := \mathsf{w}(v)/\mathsf{w}(u)$. Clearly, $X_v \sim U[0,1]$.

Our analysis proceeds as follows: First we will show that the expected number of d-heavy nodes $\mathbb{E}[H^d]$ is comparatively small. Then we will prove that H^d is sharply concentrated around $\mathbb{E}[H^d]$, and thus H^d is small with high probability. Finally, the results for Algorithm HF follow easily.

7.2 Expected Number of Heavy Nodes

Consider a node v_l on level l of T_p^∞ (level 0 contains only the root of the tree, level i contains all nodes at BFS-distance i) and denote its ancestors

on levels $0, 1, \ldots, l-1$ by $v_0, v_1, \ldots, v_{l-1}$. Using the notation introduced above, we have

$$\mathsf{w}(v_l) = W \cdot \prod_{i=1}^{l} X_{v_i} . \tag{7.1}$$

The following lemma enables us to analyze the distribution of products of $U[0,1]$-distributed random variables exactly.

Lemma 7.2 *Let $X_1, X_2, \ldots, X_n$ be independent random variables with exponential distribution and $\mathbb{E}[X_i] = 1/\lambda$, $1 \leq i \leq n$. Then for $X := \sum_{i=1}^{n} X_i$*

$$\Pr[X \leq t] = \begin{cases} 1 - \sum_{i=0}^{n-1} \frac{(\lambda t)^i}{i!} \cdot e^{-\lambda t} & \text{if } t \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Proof: See [Fel71, p. 11]. □

The distribution shown in Lemma 7.2 is a special case of the *gamma distribution* and is also known as *Erlang distribution*. It can be rewritten in a form which is more convenient for our purposes:

$$\Pr[X \leq t] = 1 - \sum_{i=0}^{n-1} \frac{(\lambda t)^i}{i!} \cdot e^{-\lambda t} = e^{-\lambda t} \left(e^{\lambda t} - \sum_{i=0}^{n-1} \frac{(\lambda t)^i}{i!} \right) = e^{-\lambda t} \cdot \sum_{i=n}^{\infty} \frac{(\lambda t)^i}{i!} .$$

It is easy to show (see [Fel71, p. 25]) that a random variable $X_i \sim U[0,1]$ can be transformed to a random variable $Y_i := -\ln X_i$ with exponential distribution and $\mathbb{E}[Y_i] = 1$. If we want to analyze $X := \prod_{i=1}^{n} X_i$, we can equivalently analyze $Y := -\ln X$. Combining this fact with Lemma 7.2 yields the following lemma.

Lemma 7.3 *Let $X := \prod_{i=1}^{n} X_i$ be the product of independent random variables with $X_i \sim U[0,1]$. If we define Y as $Y := -\ln X$, we obtain for $t \in [0,1]$*

$$\Pr[X \geq t] = \Pr[Y \leq -\ln t] = t \cdot \sum_{i=n}^{\infty} \frac{(-\ln t)^i}{i!} .$$

Proof: See [Fel71, p. 25]. □

In the following discussion we will often be concerned with the probability p_l^d that an arbitrary node on level l in the IBT is d-heavy.

Lemma 7.4 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : P \to \mathbb{R}^+$ that has uniform bisectors. Furthermore, let $p \in \mathcal{P}$, $d \in \mathbb{R}^+$ such that $\mathsf{w}(p) \geq d$, and let v_l be an arbitrary node on level l of T_p^∞. Denote the ancestors of v_l on levels $0, 1, \ldots, l-1$ by $v_0, v_1, \ldots, v_{l-1}$. Then for*

$$p_l^d := \Pr[\mathsf{w}(v_l) \geq d]$$

it holds that:

$$p_l^d = \frac{d}{W} \cdot \sum_{i=l}^{\infty} \frac{(\ln(W/d))^i}{i!}, \tag{7.2}$$

$$p_l^d \leq \frac{d}{W} \cdot \left(\frac{\ln(W/d) \cdot e}{l}\right)^l \quad \text{for } l \geq \max\{\ln(W/d), 1\}. \tag{7.3}$$

Proof: Using Equation (7.1) and applying Lemma 7.3 proves Equation (7.2):

$$\begin{aligned} p_l^d &= \Pr[\mathsf{w}(v_l) \geq d] \\ &= \Pr\left[W \cdot \prod_{i=1}^{l} X_{v_i} \geq d\right] \\ &= \Pr\left[\sum_{i=1}^{l} Y_{v_i} \leq -\ln\left(\frac{d}{W}\right)\right] \\ &= \frac{d}{W} \cdot \sum_{i=l}^{\infty} \frac{(\ln(W/d))^i}{i!}. \end{aligned}$$

Next we show that Inequality (7.3) holds:

$$\begin{aligned} \sum_{i=l}^{\infty} \frac{(\ln(W/d))^i}{i!} &= (\ln(W/d))^l \cdot \sum_{i=0}^{\infty} \frac{(\ln(W/d))^i}{(i+l)!} \\ &= (\ln(W/d))^l \cdot \frac{1}{l^l} \cdot \sum_{i=0}^{\infty} \frac{(\ln(W/d))^i \cdot l^l}{(i+l)!} \\ &\leq (\ln(W/d))^l \cdot \frac{1}{l^l} \cdot \sum_{i=0}^{\infty} \frac{l^{i+l}}{(i+l)!} \\ &\leq (\ln(W/d))^l \cdot \frac{e^l}{l^l}. \end{aligned}$$

□

Now we are in a position to state the first main result of this chapter, namely the expected value for the number of d-heavy nodes in an IBT.

Theorem 7.5 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : P \to \mathbb{R}^+$ that has uniform bisectors. Given a problem $p \in \mathcal{P}$ and a number $d \in \mathbb{R}^+$ such that $\mathsf{w}(p) \geq d$, it holds that the expected number of d-heavy nodes*

$$\mathbb{E}[H^d] = \frac{2}{d} \cdot \mathsf{w}(p) - 1.$$

Proof: Let C_l denote the set of nodes on level l in C. For node $v \in C_l$ we obtain

$$\mathbb{E}[H_v^d] = p_l^d \cdot \Pr[v \in T_p^\infty] = p_l^d,$$

because $\Pr[v \in T_p^\infty] = 1$.

To see this, suppose that $v \notin T_p^\infty$. Clearly, this implies that the level l of v is at least 2, and that there is a leaf $\tilde{v}$ of T_p^∞ on level $\tilde{l}$, $1 \leq \tilde{l} < l$. Since $\tilde{v}$ is never bisected by Algorithm HF, it follows that in each bisection step there exists a node distinct from $\tilde{v}$ that is at least as heavy as $\tilde{v}$ and that is bisected during this step. Consequently, infinitely many nodes in T_p^∞ have weight at least $\mathsf{w}(\tilde{v})$.

Let us first assume that $\mathsf{w}(\tilde{v}) = 0$. This means that the generalized bisection parameter of the bisection which yielded $\tilde{v}$ was 0 or 1. Clearly, the set of sequences in Ω that contain one or more elements in $\{0, 1\}$ is a null set with respect to the uniform distribution on Ω.

Suppose now that $\mathsf{w}(\tilde{v}) > 0$. At the beginning of each bisection step we denote by L the set of leaves of the bisection tree generated so far that have weight at least $\mathsf{w}(\tilde{v})$ and that are distinct from $\tilde{v}$. From $\mathsf{w}(\tilde{v}) > 0$ we conclude that L contains at most $k := \lfloor W/\mathsf{w}(\tilde{v}) \rfloor$ elements at any given moment. Fix an arbitrary $\varepsilon > 0$ and assume that a sequence in Ω contains infinitely many "good" bisections from the interval $[\varepsilon, 1 - \varepsilon]$. After at most k such good bisections the maximum weight in L is reduced by at least $1 - \varepsilon$, since all intermediate "bad" bisections cannot increase the maximum weight in L. But since there are infinitely many good bisections this implies that L will be empty after a finite number of bisection steps. A contradiction. Thus, any sequence in Ω such that Algorithm HF never bisects $\tilde{v}$ contains only finitely many elements from the interval $[\varepsilon, 1 - \varepsilon]$. Again, the set of all such sequences in Ω is a null set.

The result now follows by linearity of expectation using the expression for p_l^d given by Equation (7.2):

$$\mathbb{E}[H^d] = \sum_{l=0}^{\infty} \sum_{v \in T(l)} \mathbb{E}[H_v^d]$$

$$
\begin{aligned}
&= \sum_{l=0}^{\infty} 2^l \cdot p_l^d \\
&= \frac{d}{W} \cdot \sum_{l=0}^{\infty} \sum_{i=l}^{\infty} 2^l \cdot \frac{(\ln(W/d))^i}{i!} \\
&= \frac{d}{W} \cdot \sum_{i=0}^{\infty} \sum_{l=0}^{i} 2^l \cdot \frac{(\ln(W/d))^i}{i!} \\
&= \frac{d}{W} \cdot \sum_{i=0}^{\infty} \frac{(\ln(W/d))^i}{i!} \cdot \sum_{l=0}^{i} 2^l \\
&= \frac{d}{W} \cdot \sum_{i=0}^{\infty} \frac{(\ln(W/d))^i}{i!} (2^{i+1} - 1) \\
&= \frac{d}{W} \cdot \left(2 \cdot \sum_{i=0}^{\infty} \frac{(2\ln(W/d))^i}{i!} - \sum_{i=0}^{\infty} \frac{(\ln(W/d))^i}{i!} \right) \\
&= \frac{d}{W} \cdot \left[2 \left(\frac{W}{d} \right)^2 - \frac{W}{d} \right] \\
&= \frac{2}{d} W - 1.
\end{aligned}
$$

□

7.3 Concentration

Theorem 7.5 gives us an idea how well Algorithm HF performs. If we set $d = 2W/N$, we get $\mathbb{E}[H^d] = N - 1$. This is exactly the number of bisections that are necessary to get N subproblems. It follows that on average, after $N - 1$ iterations of Algorithm HF, every heavy subproblem has been bisected, and all N subproblems generated have weight smaller than $2W/N$. This exceeds the optimal value W/N only by a factor of 2. In the following, we will show that Algorithm HF really behaves the way this intuitive argument suggests. This is due to the fact that H^d is sharply concentrated around its expectation.

The following lemma shows that with high probability all heavy nodes reside rather close to the root of the IBT.

Lemma 7.6 *Fix $c \in \mathbb{R}^+$ and $p \in \mathcal{P}$. Then, with probability $1 - o(1/N)$, all nodes of T_p^∞ on level $l > 2e \cdot \ln((WN)/c)$ are (c/N)-light.*

Proof: Setting $l := k \cdot \ln((WN)/c)$ for $k > 1$ such that $l \in \mathbb{N}$, and applying Lemma 7.4 with $d = c/N$ we obtain

$$\begin{aligned}\Pr\left[\exists\ (c/N)\text{-heavy node on level } l\right] &\leq 2^l \cdot p_l^{c/N} \\ &\leq 2^{k\ln((WN)/c)} \cdot \frac{c}{WN} \cdot \left(\frac{e}{k}\right)^{k\ln((WN)/c)} \\ &= \left(\frac{WN}{c}\right)^{k\ln 2} \frac{c}{WN} \cdot \left(\frac{WN}{c}\right)^{k-k\ln k} \\ &= \left(\frac{c}{WN}\right)^{1+k\ln k-k-k\ln 2}.\end{aligned}$$

A simple analysis of the exponent shows that for $k > 2e$

$$\Pr\left[\exists\ (c/N)\text{-heavy node on level } l\right] = o(1/N). \qquad \square$$

Note that the error term $o(1/N)$ in the above lemma is chosen rather arbitrarily and could be changed to $o(1/\mathrm{poly}(N))$ without major changes in the proof. This remark also applies to our subsequent results.

Lemma 7.6 immediately yields a rather weak upper bound on the total number of heavy nodes, which we will improve later.

Corollary 7.7 *With probability* $1 - o(1/N)$ *it holds that* $H^{c/N} = O(N \log N)$.

Proof: Lemma 7.6 shows that with high probability only the first, say, $6\ln(WN/c)$ levels contain heavy nodes. In every level there are at most WN/c heavy nodes since the weights of all nodes on the same level must have sum W. $\square$

Since we want to show that with high probability $H^{2W/N} \approx N$, Corollary 7.7 is still far off from our desired result, but we already know now that the number of bisections needed by Algorithm HF to ensure that all subproblems are $(2W/N)$-light is $O(N \log N)$ with probability $1 - o(1/N)$. In order to improve this result, we define a martingale and apply the method of bounded differences to show the sharp concentration of H^d around its expectation.

For the definition of the martingale we denote by $\mathcal{F}_i := \sigma(A_1, A_2, \ldots, A_i)$ the σ-field generated by the random variables $A_i \sim U[0,1]$, $i \geq 1$ (i.e., $\hat{\alpha}_i$ is a realization of A_i). Furthermore, we set $\mathcal{F}_0 := \{\emptyset, \Omega\}$. Clearly, the sequence $(\mathcal{F}_i)_{i\geq 0}$ forms a *filter* (or *filtration*) in $(\Omega, \mathcal{F})$. Since $\mathbb{E}[H^d]$ is finite

$$Z_i^d := \mathbb{E}[H^d \mid \mathcal{F}_i] = \mathbb{E}[H^d \mid A_1, A_2, \ldots, A_i]$$

defines a martingale, which is sometimes called a *Doob-martingale* (see [FG97, Wil91, Fel71]). It holds that $Z_0^d = \mathbb{E}[H^d]$. As usual it is understood that all equations involving conditional expectations hold *almost surely*.

The intuitive interpretation of Z_i^d is as follows: Given a sequence $s := (\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_i, \ldots) \in \Omega$, $Z_i^d(s)$ tells us how many heavy nodes we expect in the corresponding IBT, if the generalized actual bisection parameters of the first i bisections are equal to $s_i := (\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_i)$.

Let T_i denote the part of the IBT visited by Algorithm HF up to and including the i-th iteration (for example, T_0 contains only the root of the tree). When we know s_i, we can simulate Algorithm HF and compute T_i. Therefore, evaluating Z_i^d corresponds to calculating the expected value of H^d, given the tree T_i which is generated by the first i bisections of Algorithm HF. In order to capture this intuition, we will use the notation $\mathbb{E}[H^d \mid T_i] := Z_i^d = \mathbb{E}[H^d \mid \mathcal{F}_i]$ in the sequel.

Since we want to apply the method of bounded differences for this martingale we need an upper bound on $|Z_{i+1}^d - Z_i^d|$. The following lemma shows that these differences are bounded by small constants.

Lemma 7.8 *For all $i \geq 0$ it holds that $|Z_{i+1}^d - Z_i^d| \leq 2$.*

Proof: Let v denote the node that is bisected during iteration $i+1$. If v is light, the claim follows easily: All nodes not yet visited by Algorithm HF must also be light, because v is a heaviest leaf in the expanded part of the tree. Thus we have already seen all heavy nodes and obtain $Z_{i+1}^d = Z_i^d$ in this case.

Now we assume that $w := \mathsf{w}(v) \geq d$. Denote by v_1 and v_2 the two nodes generated by the bisection of v, and let $\mathsf{w}(v_1) = \hat{\alpha}_{i+1} w$ and $\mathsf{w}(v_2) = (1 - \hat{\alpha}_{i+1}) w$. We may assume w.l.o.g. that $\mathsf{w}(v_1) \geq \mathsf{w}(v_2)$.

Let I denote the set of interior nodes of T_i, and let L denote the set of leaves of T_i except v. Then we have:

$$\begin{aligned} T_i &= I \cup L \cup \{v\} \\ T_{i+1} &= I \cup L \cup \{v\} \cup \{v_1\} \cup \{v_2\}\,. \end{aligned}$$

We have shown in the proof of Theorem 7.5 that T_p^∞ is a (weighted) complete binary tree of infinite height with probability 1. Therefore, we may assume in the following that for all $u \in C$ the node u is part of T_p^∞. Let $u \in C$, and denote by $T(u)$ the subtree of T_p^∞ rooted at u. Furthermore,

let $\overline{H}^d(u \mid T_j) := \sum_{x \in T(u)} \mathbb{E}[H_x^d \mid T_j]$ for some prefix T_j of T_p^∞. Suppose now that u is a leaf of T_j. If u is light, $\overline{H}^d(u \mid T_j) = 0$. If u is heavy, we conclude from Theorem 7.5 that $\overline{H}^d(u \mid T_j) = (2/d)\mathsf{w}(u) - 1$.

By linearity of conditional expectation, and by applying the notation introduced above, we have:

$$\begin{aligned} Z_i^d &= \mathbb{E}[H^d \mid T_i] = \sum_{u \in C} \mathbb{E}[H_u^d \mid T_i] = |I| + \sum_{u \in L} \overline{H}^d(u \mid T_i) + \overline{H}^d(v \mid T_i), \\ Z_{i+1}^d &= \mathbb{E}[H^d \mid T_{i+1}] = \sum_{u \in C} \mathbb{E}[H_u^d \mid T_{i+1}] \\ &= |I| + \sum_{u \in L} \overline{H}^d(u \mid T_i) + 1 + \overline{H}^d(v_1 \mid T_{i+1}) + \overline{H}^d(v_2 \mid T_{i+1}), \end{aligned}$$

since $\mathbb{E}[H_v^d \mid T_{i+1}] = 1$, because v is heavy. Subtracting these two equations shows that we may restrict our attention to that part of the tree which is changed by the $(i+1)$-st bisection:

$$Z_i^d - Z_{i+1}^d = \overline{H}^d(v \mid T_i) - \left(1 + \overline{H}^d(v_1 \mid T_{i+1}) + \overline{H}^d(v_2 \mid T_{i+1})\right).$$

We bound the absolute value of this difference by considering three cases regarding the weights of v_1 and v_2.

Case 1: $(1 - \hat{\alpha}_{i+1})w \geq d$. This means that v_1 and v_2 are heavy. It follows that

$$Z_i^d - Z_{i+1}^d = w\frac{2}{d} - 1 - \left(1 + \hat{\alpha}_{i+1} w\frac{2}{d} - 1 + (1 - \hat{\alpha}_{i+1})w\frac{2}{d} - 1\right) = 0.$$

Case 2: $(1 - \hat{\alpha}_{i+1})w < d$ and $\hat{\alpha}_{i+1}w \geq d$. Thus, only v_1 is heavy. This yields

$$Z_i^d - Z_{i+1}^d = w\frac{2}{d} - 1 - \left(1 + \hat{\alpha}_{i+1} w\frac{2}{d} - 1\right) = (1 - \hat{\alpha}_{i+1})w\frac{2}{d} - 1 < 1.$$

As $\hat{\alpha}_{i+1} \leq 1$ we have also

$$Z_i^d - Z_{i+1}^d \geq -1.$$

Case 3: $\hat{\alpha}_{i+1}w < d$. In this case v_1 and v_2 are both light. This implies that $w < 2d$ and therefore

$$Z_i^d - Z_{i+1}^d = w\frac{2}{d} - 1 - 1 < 2.$$

Furthermore, we obtain due to $w \geq d$

$$Z_i^d - Z_{i+1}^d \geq 0,$$

and the proof is complete. □

The next lemma is basic probability theory, but we prefer to state it separately, in order to make the proof of Theorem 7.11 more readable.

Lemma 7.9 *Let A and B be two events over a probability space Ω. If* $\Pr[B] = 1 - o(1/N)$,

$$\Pr[A] \leq \Pr[A \cap B] + o(1/N).$$

Proof:

$$\begin{aligned}\Pr[A] &= \Pr[A|B] \cdot \Pr[B] + \Pr[A|\overline{B}] \cdot \Pr[\overline{B}] \\ &\leq \Pr[A|B] \cdot \Pr[B] + o(1/N) \\ &= \Pr[A \cap B] + o(1/N),\end{aligned}$$

since $\Pr[A|\overline{B}] \leq 1$. □

Before proving the sharp concentration result, let us first state the theorem from the method of bounded differences, which we are going to use (see also [McD89, McD98, MR95]):

Theorem 7.10 (Azuma's inequality, *[Azu67]) Let $X_0, X_1, \ldots$ be a martingale sequence such that for each k*

$$|X_k - X_{k-1}| \leq c_k\,,$$

where c_k may depend on k. Then, for all $t \geq 0$ and any $\lambda > 0$

$$\Pr\left[|X_t - X_0| \geq \lambda\right] \leq 2\exp\left(-\frac{\lambda^2}{2\sum_{k=1}^{t} c_k^2}\right).$$

Using this inequality we prove the following theorem:

Theorem 7.11 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ that has uniform bisectors. Given a problem $p \in \mathcal{P}$, and a fixed $c \in \mathbb{R}^+$, it holds for $k > 0$ with probability $1 - 2e^{-k^2/9} - o(1/N)$ that*

$$|H^d - \mathbb{E}[H^d]| < k\sqrt{\frac{2\mathsf{w}(p)}{c}}\sqrt{N},$$

where $d = c/N$.

Proof: First we show that, if $H^d \le t$, $Z_t^d = H^d$. After t steps Algorithm HF has bisected all H^d heavy nodes. Hence, only light nodes remain and we know the exact value of H^d. Consequently, $H^d = \mathbb{E}[H^d \mid T_t] = Z_t^d$ in this case.

We have shown in Lemma 7.8 that $|Z_i^d - Z_{i+1}^d| \le 2$. Now we apply Azuma's inequality for $\lambda = N^\gamma$ with $\gamma = 0.5 + \varepsilon$, $\varepsilon > 0$. For $t' = \Theta(N \log N)$ we obtain from Corollary 7.7 and Lemma 7.9:

$$\begin{aligned}
\Pr\left[|H^d - \mathbb{E}[H^d]| \ge N^\gamma\right] &\le \Pr\left[|H^d - \mathbb{E}[H^d]| \ge N^\gamma \ \wedge\ H^d \le t'\right] + o(1/N) \\
&= \Pr\left[|Z_{t'}^d - Z_0^d| \ge N^\gamma \ \wedge\ H^d \le t'\right] + o(1/N) \\
&\le \Pr\left[|Z_{t'}^d - Z_0^d| \ge N^\gamma\right] + o(1/N) \\
&\le 2 \cdot \exp\left(-\frac{N^{2\gamma}}{2\sum_{i=1}^{t'} 4}\right) + o(1/N) \\
&= 2 \cdot \exp\left(-\frac{N^{2\gamma}}{8t'}\right) + o(1/N) \\
&= o(1/N).
\end{aligned}$$

Now we know that with high probability $H^d < t'' := 2W/d - 1 + N^\gamma = (2W/c)N - 1 + N^\gamma$. If we apply Azuma's inequality one more time using this estimate for H^d, we get (for N sufficiently large)

$$\begin{aligned}
&\Pr[|H^d - \mathbb{E}[H^d]| \ge k\sqrt{(2W/c)N}] \\
&\quad\le \Pr[|H^d - \mathbb{E}[H^d]| \ge k\sqrt{(2W/c)N} \ \wedge\ H^d \le t''] + o(1/N) \\
&\quad\le \Pr[|Z_{t''}^d - Z_0^d| \ge k\sqrt{(2W/c)N}] + o(1/N) \\
&\quad\le 2 \cdot \exp\left(-\frac{k^2 \cdot (2W/c)N}{8((2W/c)N - 1 + N^\gamma)}\right) + o(1/N) \\
&\quad< 2e^{-k^2/9} + o(1/N).
\end{aligned}$$

□

The results for the random variable $H^{c/N}$ immediately yield the desired results for the performance of Algorithm HF.

Theorem 7.12 *Let $\mathcal{P}$ be a class of problems with weight function $\mathsf{w} : \mathcal{P} \to \mathbb{R}^+$ that has uniform bisectors. Given a problem $p \in \mathcal{P}$, Algorithm HF partitions p into N subproblems $p_1, \ldots, p_N$ such that for $\varepsilon = 9\sqrt{\ln(N)/N}$*

$$\Pr\left[(2-\varepsilon)\frac{\mathsf{w}(p)}{N} \le \max_{1\le i\le N} \mathsf{w}(p_i) < (2+\varepsilon)\frac{\mathsf{w}(p)}{N}\right] = 1 - o(1/N).$$

Proof: Let $m := \max_{1 \le i \le N} \mathsf{w}(p_i)$. We show first that

$$p^+ := \Pr\left[m \ge (2+\varepsilon)\frac{W}{N}\right] = o(1/N).$$

For $\gamma := (2+\varepsilon)\frac{W}{N}$ it holds that

$$p^+ := \Pr[m \ge \gamma] = \Pr[H^\gamma \ge N],$$

because Algorithm HF expands heavy subproblems before light subproblems. By Theorem 7.5 we have

$$\mathbb{E}[H^\gamma] = \frac{2}{\gamma}W - 1 = \frac{2}{2+\varepsilon}N - 1.$$

Thus, we can rewrite p^+ as follows:

$$\begin{aligned} p^+ &= \Pr\left[H^\gamma \ge \frac{2}{2+\varepsilon}N - 1 + \frac{\varepsilon}{2+\varepsilon}N + 1\right] \\ &\le \Pr\left[H^\gamma - \mathbb{E}[H^\gamma] \ge \frac{\varepsilon}{2+\varepsilon}N\right] \\ &\le \Pr\left[|H^\gamma - \mathbb{E}[H^\gamma]| \ge \frac{\varepsilon}{2+\varepsilon}N\right]. \end{aligned}$$

For N sufficiently large we have

$$\frac{\varepsilon}{2+\varepsilon}N \ge \frac{\varepsilon}{2.9}N \ge 3.1\sqrt{\ln(N)\cdot N} \ge 3.1\sqrt{\ln(N)}\sqrt{\frac{2}{2+\varepsilon}}\sqrt{N}.$$

Therefore, using Theorem 7.11 with $k = 3.1\sqrt{\ln(N)}$ we obtain:

$$p^+ \le \Pr\left[|H^\gamma - \mathbb{E}[H^\gamma]| \ge 3.1\sqrt{\ln(N)}\sqrt{2W/(2+\varepsilon)W}\sqrt{N}\right] = o(1/N).$$

The second case is very similar. With $\delta := (2-\varepsilon)\frac{W}{N}$ we have for N sufficiently large

$$p^- := \Pr[m < \delta] = \Pr[H^\delta \le N - 1].$$

Furthermore,

$$\mathbb{E}[H^\delta] = \frac{2}{\delta}W - 1 = \frac{2}{2-\varepsilon}N - 1.$$

Hence,

$$\begin{aligned} p^- &= \Pr\left[H^\delta \le \frac{2}{2-\varepsilon}N - 1 - \frac{\varepsilon}{2-\varepsilon}N\right] \\ &= \Pr\left[H^\delta - \mathbb{E}[H^\delta] \le -\frac{\varepsilon}{2-\varepsilon}N\right] \\ &\le \Pr\left[|H^\delta - \mathbb{E}[H^\delta]| \ge \frac{\varepsilon}{2-\varepsilon}N\right] \\ &= o(1/N). \end{aligned}$$

□

7.4 From Heaviest to Heavy

All the proofs did not depend much on the exact order in which Algorithm HF processes the nodes of the bisection tree. We only used the observation that heavy nodes are processed before light ones. This property can easily be achieved by maintaining a list or a queue of all d-heavy leaves of the bisection tree for a suitably chosen d. Using a queue is favorable since then the nodes of the bisection tree are processed level by level. This insures that after $N-1$, $N \ge 2$, bisections all heavy subproblems on levels $0, 1, \ldots, \lfloor \log N \rfloor - 1$ have already been bisected. This approach requires only constant time per iteration in contrast to logarithmic time, if priority queues are used to retrieve a heaviest node in each bisection step.

Therefore, the results for our stochastic model suggest a faster variant of Algorithm HF: First, the threshold for the "heaviness" of a subproblem is computed as $d = (2+\varepsilon) \cdot \mathsf{w}(p)/N$, $\varepsilon = 9\sqrt{\ln N/N}$, for which we expect $(2/(2+\varepsilon))N - 1$ d-heavy nodes in the bisection tree. Then, p is inserted into the queue of heavy subproblems (if $N \ge 2$) and the head of the queue is repeatedly bisected (inserting the resulting subproblems appropriately into the list of light or the queue of heavy subproblems) until the queue of heavy subproblems is empty or there are N subproblems (or both).

In the first case, we know that the ratio between the maximum weight subproblem generated by Algorithm HF and the ideal weight $\mathsf{w}(p)/N$ is less than $2+\varepsilon$. The number of subproblems generated in this case is close to N with high probability by Theorem 7.11. Therefore, we cannot hope for much improvement if the bisection process were continued to generate exactly N subproblems. We have shown in the proof of Theorem 7.12 that the remaining case (there are N subproblems in the end and the queue of heavy subproblems is not empty) occurs with probability

$o(1/N)$. If the maximum load generated in this case is "too bad", we can afford to restart the bisection process using the original Algorithm HF (or simply repeat the faster variant in case of a randomized bisection subroutine) without affecting the overall running time in order to achieve a possibly better maximum load.

These observations can also be applied to simplify Algorithm PHF (see Section 6.1.1). Assuming classes of problems that have uniform bisectors, the first phase of Algorithm PHF using the above threshold instead of r_α (which is used for classes of problems that have α bisectors) would produce with high probability a well balanced load on the N processors leaving only few processors idle.

Chapter 8

Conclusion

In this chapter we first summarize and discuss in Section 8.1 the results presented in this thesis. Then, in Section 8.2, we expose some open problems arising from our research, and give possible directions for future work.

8.1 Summary of Results

In this thesis we studied on-line scheduling and load distribution, two fundamental and widely used techniques for efficient resource management in parallel systems. To reveal the major differences of these two approaches, we gave a comparison of on-line scheduling and load distribution from our point of view in Chapter 2. A classification scheme for load distribution strategies was proposed in Chapter 3 that is aimed at making the comparison and evaluation of different strategies easier. Furthermore, it may serve as a guideline for future work in this area since many important aspects that a load distribution strategy should take into account are exposed in a clear and understandable way. We also surveyed several load distribution strategies that were selected in order to demonstrate exemplary algorithmic approaches and techniques for the solution of load distribution problems.

We presented and analyzed several on-line scheduling algorithms for parallel job systems in Chapter 4. Three different models regarding the degree of a priori knowledge about the execution times of the jobs that is available to the on-line scheduler were studied. The first model requires

that the jobs have unit execution time. A general lower bound of 2 on the competitive ratio was given that is valid for any network topology and deterministic as well as randomized on-line schedulers.

Then we proposed the LEVEL Algorithm for this scheduling problem that repeatedly schedules a set of available jobs (using a packing algorithm that is suitable for the interconnection structure of a given parallel system) and collects all jobs that arrive during a phase for execution in the next phase. For the complete model (parallel systems that support arbitrary allocation of processors to parallel jobs) and the linear array we showed that using the Next-Fit BIN PACKING approximation algorithm yields a 3-competitive algorithm. This result can be improved at the expense of a slightly increased scheduling overhead if First-Fit is employed instead. We showed that LEVEL(First-Fit) is 2.7-competitive. This is almost optimal for deterministic on-line scheduling algorithms due to a lower bound of 2.691.

Using a simple packing algorithm it was shown that the LEVEL Algorithm achieves optimal competitive ratio 2 for hypercubes, and for the 2-dimensional array the competitive ratio of the LEVEL(PACK_2D) was shown to be in the interval $[13/4, 46/7]$. Finally, we gave a general theorem for arbitrary networks that provides an upper bound on the competitive ratio of the LEVEL(PACK) Algorithm if PACK meets certain requirements.

To investigate the entire bandwidth between unit and arbitrary running times of the jobs, we subsequently assumed that the runtime ratio (i.e., the ratio between the longest and shortest running time of any job) of a job system is bounded by a parameter T_R that is unknown to the on-line scheduler. Again, for an arbitrary network topology, a general lower bound of $(T_R + 1)/2$ on the competitive ratio of any deterministic or randomized on-line scheduler was derived that still holds if the actual runtime ratio of a particular job system is known to the on-line algorithm in advance.

The RRR Algorithm is designed for the complete model and requires that the minimum execution time of any job in a job system is known in the beginning. We showed that this on-line scheduler is $(T_R/2+4)$-competitive and is thus optimal up to an additive constant. Then we removed all a priori knowledge about job running times and showed that it is still possible to devise a nearly optimal on-line algorithm. More precisely, we proved that the RRR_ADAPTIVE Algorithm is $(T_R/2+5.5)$-competitive in this third model. Furthermore, we showed that *waiting* if a big parallel job cannot be scheduled although the efficiency is low and *collecting* small jobs during a

delay phase are essential characteristics of our on-line schedulers in this model. Finally, additional results for other interconnection topologies are given.

It became evident that *runtime restrictions* improve the competitive performance achievable by on-line schedulers. Therefore, if enough a priori knowledge about job running times is available to bound the runtime ratio of a job system by a reasonably small constant, our schedulers can guarantee a satisfactory utilization of the parallel system. But even without any such knowledge the RRR_ADAPTIVE algorithm produces schedules that are almost best possible from a worst-case point of view. All proposed on-line algorithms are computationally simple, and thus the scheduling overhead involved can safely be neglected, provided that the parallel system is able to deliver the necessary information quickly.

In Chapter 5 the existence of α-bisectors for a class of problems was shown to allow good load balancing for a surprisingly large range of values of α. The maximum load produced by Algorithm HF is at most a factor of $\lfloor 1/\alpha \rfloor \cdot (1-\alpha)^{\lfloor 1/\alpha \rfloor - 2}$ larger than the theoretical optimum (uniform distribution). This bound was shown to be tight. It gives a performance guarantee between 5 and 2 when the bisection parameter ranges from 0.084 to 0.5. Furthermore, it was shown that Algorithm HF is optimal from a worst-case point of view.

Load balancing for distributed hierarchical finite element simulations was discussed, and two strategies for applying Algorithm HF were presented. The first strategy tries to make the best use of the available parallelism, but requires that the nodes of the FE tree representing the load of the computation have good separators. The second strategy tries to partition the entire FE tree into subtrees with approximately equal load. For this purpose, it was shown that a certain class of weighted trees, which include FE trees, has 1/4-bisectors. Here, the trees are bisected by removing a single edge. Partitioning the trees by removing a minimum number of edges ensures that only a minimum number of communication channels of the application must be realized by network connections. Our results provide performance guarantees for balancing the load of applications with good bisectors in general and of distributed hierarchical finite element simulations in particular. For the latter application, we showed that the maximum resulting load is at most a factor of 9/4 larger than in a perfectly uniform distribution.

We implemented the proposed load balancing methods and integrated them into the existing finite element simulation software ARESO.

We obtained considerable improvements already for small problems, as compared to the static (compile-time) processor allocation currently in use. Since ARESO is primarily a solver of hierarchical systems of equations, it is not limited to static simulations. Other physical problems described by elliptic partial differential equations are tractable as well. Currently, a component for CFD (computational fluid dynamics) simulations taken from [Fun97] is added.

It seems possible that a variety of other hierarchical numerical distributed algorithms could be accelerated with Algorithm HF. Domain decomposition in the process of chip layout with the placement tool GORDIAN [RR93, Reg97] may result in an unbalanced binary tree. The subsequent layout process could be improved by load distribution using Algorithm HF. Another application is the multi-dimensional adaptive numerical quadrature `aqho` [Bon93, Bon95]. It is based on an adaptively growing binary tree. Algorithm HF may be applied in much the same way as in the ARESO application, because each traversal visits all tree nodes and adds a new (and potentially incomplete) layer of leaves.

Based on the results on the sequential Algorithm HF, we derived three promising parallel algorithms for load balancing of problems with good bisectors in Chapter 6. While the sequential Algorithm HF has running time $O(N)$ for distributing a problem onto N processors, all three parallel algorithms require only running time $O(\log N)$ on N processors under reasonably general assumptions about the parallel machine architecture.

Algorithm PHF is a parallelization of Algorithm HF that produces the same load balancing as the sequential algorithm. Its advantage is that it inherits the good performance of Algorithm HF in the worst case and in the average case. Its drawback is that the management of free processors is costly and that global communication is required during its execution. If global communication is too expensive on a particular machine, the practical use of Algorithm PHF may be limited.

Algorithm BA is inherently parallel and partitions a problem into N subproblems without requiring any global communication. In fact, the only communication carried out during the run of Algorithm BA is the transmission of subproblems to free processors after bisections. Furthermore, the management of free processors is trivial for Algorithm BA. Regarding the worst-case performance guarantee of Algorithm BA, we have proven the upper bound $e \cdot \lfloor \frac{1}{\alpha} \rfloor \cdot (1-\alpha)^{\lfloor \frac{1}{2\alpha} \rfloor - 1}$, which is not as good as the upper bound for Algorithm PHF, but still constant for fixed α.

Algorithm BA-HF is a combination of Algorithm BA and Algorithm HF or PHF. It uses Algorithm BA in the beginning and switches to Algorithm HF or PHF once the number of processors for a subproblem is below a certain threshold. By adjusting this threshold using a parameter σ, the worst-case performance guarantee of Algorithm BA-HF, which is bounded by $e^{(1-\alpha)/\sigma} \cdot (1 + \alpha/\sigma) \cdot r_\alpha$, can be brought arbitrarily close to that of Algorithm PHF at the expense of increasing the influence of the drawbacks of Algorithm PHF.

Furthermore, we conducted extensive simulation experiments to determine the relative quality of the load distribution achieved by the individual algorithms in the average case. The actual bisection parameters were drawn uniformly and independently at random from the interval $[\alpha, \beta]$, $0 \leq \alpha < \beta \leq 1/2$. The experiments showed that the performance in the average case was substantially better than the worst-case upper bounds for all three algorithms. It was confirmed that the balancing quality was best for Algorithm HF and worst for Algorithm BA in all experiments. In order to choose one of the proposed load balancing algorithms in practice, one must take into account the characteristics of the parallel machine architecture as well as the relative importance of fast running time of the load balancing algorithm and of the quality of the achieved load balance. Our worst-case bounds and extensive simulation results provide helpful guidance for this decision.

Motivated by the quite surprising simulation results, we analyzed the average-case balancing quality of Algorithm HF under the assumption that the actual bisection parameter of each bisection step is drawn uniformly and independently at random from the interval $[0, 1/2]$. In Chapter 7 we showed that under these assumptions the maximum load generated by Algorithm HF is in the interval $[(2 - \varepsilon)\frac{w(p)}{N}, (2 + \varepsilon)\frac{w(p)}{N}]$ with high probability. Moreover, ε is close to zero already for moderate values of N. Thus, assuming a natural and rather pessimistic distribution for the average case, the partitioning computed by Algorithm HF exceeds the optimum solution only roughly by a factor of 2 already for realistic numbers of processors. From our analysis we also derived faster and simpler variants of Algorithm HF and Algorithm PHF for this stochastic model. These results demonstrate that a satisfactory balancing quality can be achieved by efficient algorithms even if no lower bound for the bisection parameter of a given class of problems is known, provided that "good" and "bad" bisections are equally likely.

8.2 Open Problems and Future Work

Not surprisingly, many open problems arise from our research. In particular, it remains to determine the exact competitive ratio of several on-line scheduling algorithms that were proposed in Chapter 4. As an example, the competitive ratio of the LEVEL(First-Fit) Algorithm is in the interval $[1 + h_\infty, 2.7]$, i.e., a there is a gap of less than 0.009 between the lower and the upper bound. We conjecture that the exact competitive ratio of this algorithm is either $1 + h_\infty$ or 2.7. In case that LEVEL(First-Fit) turns out to be sub-optimal, it would be a challenging task to devise an optimal on-line scheduler for this model. Although open questions of this kind may appear to be solely of academic interest, their solution often leads to new algorithmic techniques, improved tools of analysis, and a better understanding of the structure of the problem.

For on-line scheduling parallel job systems with restricted runtime ratio it seems also worthwhile to find suitable modifications of the RRR algorithms for other interconnection topologies since it appears that the LEVEL Algorithm does not yield a close to optimal competitive ratio.

Another possible scheduling model is to assume that the running time of a job is known to the online scheduler (or can be estimated with sufficient precision) when a job becomes available, but the precedence constraints are unknown. This appears to be a good starting point for the development of less restrictive scheduling models that admit on-line schedulers with good competitive performance in the presence of dependencies. The addition of communication delays (cf. [VLL90, BEP+96, Ver98]) to our scheduling models would be an important extension since the time for transmission of data from a job to those successors that are scheduled on different processors is non-negligible for many parallel applications.

All our bisection-based load balancing algorithms did not specify which bisection should be performed if there were several possibilities with different actual bisection parameter. But this opportunity will often be given in practical applications, and it is near at hand to choose the best possible bisection in each step[1]. Furthermore, it would be an advantage in this model to explore a greater part of the bisection tree performing more than $N-1$ bisection in order to obtain N subproblems. The analysis of such

[1] However, this local greedy strategy is not optimal. This is seen easily, for example, in the case $N = 3$.

extensions might yield load balancing algorithms with better performance than Algorithm HF.

Our simulation results indicate that the performance of Algorithm BA is inferior to the performance of Algorithm HF both in the worst case and in the average case. Hence, it seems promising to confirm these results by mathematical analysis. Firstly, a worst-case lower bound on the maximum load generated by Algorithm BA might be derived that is larger than r_α, and secondly an average-case analysis similar to Algorithm HF might be done. Such an analysis could also influence the design and tuning of parallel load balancing algorithms such as Algorithm BA-HF.

A natural and important generalization of our average-case analysis of load balancing using bisectors would be an analysis for an arbitrary uniform distribution $U[\alpha, \beta]$ for the actual bisection parameter. So far, we have not been able to transfer our approach for the distribution $U[0, 1/2]$ to the general case. Furthermore, it would be interesting to allow for a certain degree of dependency between bisection steps that take place on a root-to-leaf path in the bisection tree.

Prerequisites

There is an abundance of sources for the prerequisites from mathematics and computer science underlying the topics discussed in this thesis. In the following, we give only a few examples.

The necessary mathematical background for the analysis of algorithms may be found in the book of GRAHAM, KNUTH, and PATASHNIK [GKP94]. Design of algorithms and their analysis is covered by the classical exposition of AHO, HOPCROFT, and ULLMAN [AHU76]. CORMEN, LEISERSON, and RIVEST [CLR90] also provide a readable and thorough introduction into this field. Randomized Algorithms and means to analyze them is the subject of the textbook by MOTWANI and RAGHAVAN [MR95]. Parallel algorithms and machine models are treated by JÁJÁ [JáJ92] within the PRAM framework, whereas LEIGHTON [Lei92] covers network-based architectures such as arrays and hypercubes. A thorough introduction to graph theory is provided by the books of BOLLOBÁS [Bol98] and WEST [Wes96], and fundamental graph algorithms are described by GIBBONS [Gib85]. The book of GAREY and JOHNSON provides an introduction to the theory of $\mathcal{NP}$-completeness, and a collection of surveys regarding approximation algorithms for $\mathcal{NP}$-hard problems is presented by HOCHBAUM [Hoc96]. Finally, probability theory is covered by the classical books of FELLER [Fel68, Fel71] that also contain numerous examples. A modern, measure theoretic approach can be found in [FG97].

Bibliography

[AAG+95] Baruch Awerbuch, Yossi Azar, Edward F. Grove, Ming-Yang Kao, P. Krishnan, and Jeffrey Scott Vitter. Load Balancing in the L_p Norm. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science FOCS'95*, pages 383–391, Los Alamitos, 1995. IEEE Computer Society Press.

[ABKU94] Y. Azar, A.Z. Broder, A.R. Karlin, and E. Upfal. Balanced Allocations. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing STOC'94*, pages 593–602, New York, 1994. ACM Press.

[ABS98] Micah Adler, Petra Berenbrink, and Klaus Schröder. Analyzing an Infinite Parallel Job Allocation Process. In G. Bilardi, G.F. Italiano, A. Piertracaprina, and G. Pucci, editors, *Proceedings of the 6th Annual European Symposium on Algorithms ESA'98*, volume 1461 of *LNCS*, pages 417–428, Berlin, 1998. Springer-Verlag.

[ACMR95] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. Parallel Randomized Load Balancing. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing STOC'95*, pages 238–248, New York, 1995. ACM Press.

[AG94] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company, Redwood City, CA, second revised edition, 1994.

[AHU76] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, MA, 1976.

[AL91] Bill Aiello and Tom Leighton. Coding Theory, Hypercube Embeddings, and Fault Tolerance. In *Proceedings of the 3rd An-*

nual ACM Symposium on Parallel Algorithms and Architectures SPAA'91, pages 125–136, New York, 1991. ACM Press.

[Asp98] James Aspnes. Competitive Analysis of Distributed Algorithms. In Amos Fiat and Gerhard J. Woeginger, editors, *Online Algorithms : The State of the Art*, volume 1442 of *LNCS*, pages 118–146. Springer Verlag, Berlin, 1998.

[ATM95] The ATM Forum, Upper Saddle River, NJ. *ATM User-Network Interface (UNI) Specification Version 3.1.*, 1995.

[Aza98] Yossi Azar. On-line Load Balancing. In Amos Fiat and Gerhard J. Woeginger, editors, *Online Algorithms : The State of the Art*, volume 1442 of *LNCS*, pages 178–195. Springer Verlag, Berlin, 1998.

[Azu67] Kazuoki Azuma. Weighted Sums of Certain Dependent Random Variables. *Tôhoku Mathematical Journal*, 19(3):357–367, 1967.

[BDG+94] A. Beguelin, J. Dongarra, A. Geist, W. Jiang, R. Manchek, and V. Sunderam. *PVM : Parallel Virtual Machine : A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, MA, 1994.

[BDMR96] A. Bäumker, W. Dittrich, F. Meyer auf der Heide, and I. Rieping. Realistic Parallel Algorithms: Priority Queue Operations and Selection for the BSP* Model. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Proceedings of the Second International EURO-PAR Conference on Parallel Processing EURO-PAR'96, Volume II*, volume 1124 of *LNCS*, pages 369–376, Berlin, 1996. Springer-Verlag.

[BDW86] J. Błazewicz, M. Drabowski, and J. Węglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, C-35(5):389–393, 1986.

[BE97] Stefan Bischof and Thomas Erlebach. Classification and Survey of Strategies. In Thomas Schnekenburger and Georg Stellner, editors, *Dynamic Load Distribution for Parallel Applications*, Teubner-Texte zur Informatik. Teubner Verlag, Stuttgart, 1997.

[BE98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, 1998.

[BEE98a] Stefan Bischof, Ralf Ebner, and Thomas Erlebach. Load Balancing for Problems with Good Bisectors, and Applications in Finite Element Simulations. In A. Bode, A. Ganz, C. Gold, S. Petri, N. Reimer, B. Schiemann, and T. Schnekenburger, editors, *"Anwendungsbezogene Lastverteilung" ALV'98*, Institut für Informatik, Technische Universität München, 1998. Sonderforschungsbereich 342 "Werkzeuge und Methoden zur Nutzung paralleler Rechnerarchitekturen", Graduiertenkolleg "Kooperation und Ressourcenmanagement in verteilten Systemen". Technical Report (SFB-Bericht 342/01/98 A).

[BEE98b] Stefan Bischof, Ralf Ebner, and Thomas Erlebach. Load Balancing for Problems with Good Bisectors, and Applications in Finite Element Simulations. In David Pritchard and Jeff Reeve, editors, *Proceedings of the Fourth International EURO-PAR Conference on Parallel Processing EURO-PAR'98*, volume 1470 of *LNCS*, pages 383–389, Berlin, 1998. Springer-Verlag.

[BEE98c] Stefan Bischof, Ralf Ebner, and Thomas Erlebach. Load Balancing for Problems with Good Bisectors, and Applications in Finite Element Simulations: Worst-case Analysis and Practical Results. SFB-Bericht 342/05/98 A, SFB 342, Institut für Informatik, Technische Universität München, 1998.

[BEE99] Stefan Bischof, Ralf Ebner, and Thomas Erlebach. Parallel Load Balancing for Problems with Good Bisectors. In *Proceedings of the 2nd Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing IPPS/SPDP'99*, pages 531–538, Los Alamitos, 1999. IEEE Computer Society Press.

[BEP+96] J. Błażewicz, K.H. Ecker, E. Pesch, G. Schmidt, and J. Węglarz. *Scheduling Computer and Manufacturing Processes*. Springer-Verlag, Berlin, 1996.

[BFM98] Petra Berenbrink, Tom Friedetzky, and Ernst W. Mayr. Parallel Continuous Randomized Load Balancing. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'98*, pages 192–201, New York, 1998. ACM Press.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science FOCS'94*, pages 356–368, Los Alamitos, 1994. IEEE Computer Society Press.

[BM98a] Stefan Bischof and Ernst W. Mayr. On-Line Scheduling of Parallel Jobs with Runtime Restrictions. In Kyung-Yong Chwa and Oscar H. Ibarra, editors, *Proceedings of the 9th International Symposium on Algorithms and Computation ISAAC '98*, volume 1533 of *LNCS*, pages 119–128, Berlin, 1998. Springer-Verlag.

[BM98b] Stefan Bischof and Ernst W. Mayr. On-Line Scheduling of Parallel Jobs with Runtime Restrictions. SFB-Bericht 342/04/98 A, SFB 342, Institut für Informatik, Technische Universität München, April 1998.

[BMS97] Petra Berenbrink, Friedhelm Meyer auf der Heide, and Klaus Schröder. Allocating Weighted Jobs in Parallel. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'97*, pages 302–310, New York, 1997. ACM Press.

[Böh96] Max Böhm. *Verteilte Lösung harter Probleme: Schneller Lastausgleich*. PhD thesis, Mathematisch-Naturwissenschaftliche Fakultät, Universität zu Köln, 1996.

[Bol98] Béla Bollobás. *Modern Graph Theory*, volume 184 of *Graduate Text in Mathematics*. Springer-Verlag, Berlin, 1998.

[Bon93] T. Bonk. A New Algorithm for Multi-Dimensional Adaptive Numerical Quadrature. In W. Hackbusch, editor, *Adaptive Methods – Algorithms, Theory and Applications: Proceedings of the 9th GAMM Seminar*, pages 54–68. Vieweg Verlag, Braunschweig, 1993.

[Bon95] T. Bonk. *Ein rekursiver Algorithmus zur adaptiven numerischen Quadratur mehrdimensionaler Funktionen*. PhD thesis, Institut für Informatik, Technische Universität München, 1995.

[BP95] Ronald I. Becker and Yehoshua Perl. The shifting algorithm technique for the partitioning of trees. *Discrete Appl. Math.*, 62:15–34, 1995.

[BP97] Martin Backschat and Alexander Pfaffinger. Dynasty: Economic-Based Dynamic Load Distribution in Large Workstation Networks. In Thomas Schnekenburger and Georg Stellner, editors, *Dynamic Load Distribution for Parallel Applications*, Teubner-Texte zur Informatik. Teubner Verlag, Stuttgart, 1997.

[Bra97] Dietrich Braess. *Finite Elemente*. Springer, Berlin, 1997. 2. überarbeitete Auflage.

[BS96] M. Böhm and E. Speckenmeyer. Precomputation based load balancing. In *Proceedings of the 4th Workshop on Parallel Systems and Algorithms PASA'96*, pages 173–190, Singapore, 1996. World Scientific Publishing Co.

[BSS99] Stefan Bischof, Thomas Schickinger, and Angelika Steger. Load Balancing Using Bisectors – A Tight Average-Case Analysis. In Jaroslav Nešetřil, editor, *Proceedings of the 7th Annual European Symposium on Algorithms ESA'99*, volume 1643 of *LNCS*, pages 172–183, Berlin, 1999. Springer-Verlag.

[BTZ98] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. A Parallel Priority Queue with Constant Time Operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, 1998. Special Issue on Parallel and Distributed Data Structures.

[Bur87] D.S. Burnett. *Finite Element Analysis*. Addison-Wesley Publishing Company, 1987.

[CGJ82] F.R.K. Chung, M.R. Garey, and D.S. Johnson. On packing two-dimensional bins. *SIAM J. Alg. Disc. Meth.*, 3(1):66–76, March 1982.

[CGJ96] E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson. Approximation Algorithms for Bin Packing: A Survey. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 2, pages 46–93. PWS Publishing Company, Boston, 1996.

[CK88] Thomas L. Casavant and Jon G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990. The MIT Electrical Engineering and Computer Science Series.

[CM96] Soumen Chakrabarti and S. Muthukrishnan. Resource scheduling for parallel database and scientific applications. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'96*, pages 329–335, New York, 1996. ACM Press.

[CS97] A. Czumaj and V. Stemann. Randomized Allocation Processes. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science FOCS'97*, pages 194–203, Los Alamitos, 1997. IEEE Computer Society Press.

[Cyb89] George Cybenko. Dynamic Load Balancing for Distributed Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[DDLM95] Thomas Decker, Ralf Diekmann, Reinhard Lüling, and Burkhard Monien. Towards Developing Universal Dynamic Mapping Algorithms. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing SPDP'95*, pages 456–459, Los Alamitos, 1995. IEEE Computer Society Press.

[DFM98] Ralf Diekmann, Andreas Frommer, and Burkhard Monien. Nearest Neighbor Load Balancing on Graphs. In G. Bilardi, G.F. Italiano, A. Piertracaprina, and G. Pucci, editors, *Proceedings of the 6th Annual European Symposium on Algorithms ESA'98*, volume 1461 of *LNCS*, pages 429–440, Berlin, 1998. Springer-Verlag.

[DL89] Jianzhong Du and Joseph Y.-T. Leung. Complexity of Scheduling Parallel Task Systems. *SIAM J. Disc. Math.*, 2:473–487, 1989.

[DMP97] Ralf Diekmann, Burkhard Monien, and Robert Preis. Load Balancing Strategies for Distributed Memory Machines. Technical Report tr-rsfb-97-050, Fachbereich Mathematik und Informatik, Universität-Gesamthochschule Paderborn, 1997.

[DMS98] Jack J. Dongarra, Hans W. Meuer, and Erich Strohmaier. The TOP500 Supercomputer Sites. http://www.top500.org, 1998.

[EP98] R. Ebner and A. Pfaffinger. Higher Level Programming and Efficient Automatic Parallelization: A Functional Data Flow Approach with FASAN. In E.H. D'Hollander, G.R. Joubert, F.J. Peters, and U. Trottenberg, editors, *Parallel Computing: Fundamentals, Applications and New Directions (Proceedings of the ParCo97 Parallel Computing Conference)*, volume 12 of *Advances in Parallel Computing*. Elsevier Science Publishers, Amsterdam, 1998.

[Eps98] Leah Epstein. Lower Bounds for On-line Scheduling with Precedence Constraints on Identical Machines. In Klaus Jansen and José Rolim, editors, *Proceedings of the First International Workshop on Approximation Algorithms for Combinatorial Optimization APPROX'98*, volume 1444 of *LNCS*, pages 89–98, Berlin, 1998. Springer Verlag.

[Fel68] William Feller. *An Introduction to Probability Theory and its Applications. Volume I.* Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, Chichester, third revised edition, 1968.

[Fel71] William Feller. *An Introduction to Probability Theory and its Applications. Volume II.* Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, Chichester, second edition, 1971.

[FG97] Bert Fristedt and Lawrence Gray. *A Modern Approach to Probability Theory*. Birkhäuser, Boston, 1997.

[FKST93] Anja Feldmann, Ming-Yang Kao, Jiří Sgall, and Shang-Hua Teng. Optimal Online Scheduling of Parallel Jobs with Dependencies. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing STOC'93*, pages 642–651, New York, 1993. ACM Press.

[FKST98] Anja Feldmann, Ming-Yang Kao, Jiří Sgall, and Shang-Hua Teng. Optimal On-Line Scheduling of Parallel Jobs with Dependencies. *Journal of Combinatorial Optimization*, 1(4):393–411, 1998.

[FR95] Dror G. Feitelson and Larry Rudolph. Parallel Job Scheduling: Issues and Approaches. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*

(IPPS'95 Workshop), volume 949 of *LNCS*, pages 1–18, Berlin, 1995. Springer-Verlag.

[FR96] Dror G. Feitelson and Larry Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing (IPPS'96 Workshop)*, volume 1162 of *LNCS*, pages 1–26, Berlin, 1996. Springer-Verlag.

[Fre91] Greg N. Frederickson. Optimal Algorithms for Tree Partitioning. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms SODA'91*, pages 168–177, New York, 1991. ACM Press.

[Fri84] Donald K. Friesen. Tighter bounds for the multifit processor scheduling algorithm. *SIAM J. Comput.*, 13(1):170–181, February 1984.

[FRS+97] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and Practice in Parallel Job Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing (IPPS'97 Workshop)*, volume 1291 of *LNCS*, pages 1–34, Berlin, 1997. Springer-Verlag.

[FST94] Anja Feldmann, Jiří Sgall, and Shang-Hua Teng. Dynamic scheduling on parallel machines. *Theoretical Computer Science, Special Issue on Dynamic and On-line Algorithms*, 130(1):49–72, 1994.

[Fun97] Kilian Funk. Anwendung der algebraischen Mehrgittermethode auf konvektionsdominierte Strömungen. Master's thesis, Technische Universität München, 1997.

[FW98] Amos Fiat and Gerhard J. Woeginger, editors. *Online Algorithms : The State of the Art*, volume 1442 of *LNCS*. Springer Verlag, Berlin, 1998.

[FWM94] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works!* Morgan Kaufmann Publishers, San Francisco, CA, 1994.

[GGJY76] M.R. Garey, R.L. Graham, D.S. Johnson, and A.C.-C. Yao. Resource Constrained Scheduling as Generalized Bin Packing. *J. Comb. Theory Series A*, 21:257–298, 1976.

[GI97] Minos N. Garofalakis and Yannis E. Ioannidis. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *Proceedings of the 23rd International Conference on Very Large Data Bases VLDB'97*, pages 296–305, San Francisco, CA, 1997. Morgan Kaufmann Publishers.

[Gib85] Alan Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, Cambridge, 1985.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness*. W. H. Freeman and Company, New York, 1979.

[GJTY83] M.R. Garey, D.S. Johnson, R.E. Tarjan, and M. Yannakakis. Scheduling Opposing Forests. *SIAM J. Alg. Disc. Meth.*, 4(1):72–93, March 1983.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Publishing Company, Reading, MA, second edition, 1994.

[GLM+95] B. Ghosh, F.T. Leighton, B.M. Maggs, S. Muthukrishnan, C.G. Plaxton, R. Rajaraman, A.W. Richa, R.E. Tarjan, and D. Zuckerman. Tight Analyses of Two Local Load Balancing Algorithms. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing STOC'95*, pages 548–558, New York, 1995. ACM Press.

[GM94] Bhaskar Ghosh and S. Muthukrishnan. Dynamic Load Balancing in Parallel and Distributed Networks by Random Matchings. In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'94*, pages 226–235, New York, 1994. ACM Press.

[GM96] Bhaskar Ghosh and S. Muthukrishnan. Dynamic Load Balancing by Random Matchings. *Journal of Computer and System Sciences*, 53(3):357–370, 1996.

[GMS96] Bhaskar Ghosh, S. Muthukrishnan, and Martin H. Schultz. First and second order diffusive methods for rapid, coarse, distributed load balancing. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'96*, pages 72–81, New York, 1996. ACM Press.

[Gra66] R.L. Graham. Bounds for Certain Multiprocessing Anomalies. *The Bell System Technical Journal*, pages 1563–1581, 1966.

[Gra69] R.L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.*, 17(2):416–429, March 1969.

[GW95] Gábor Galambos and Gerhard G. Woeginger. On-Line Bin Packing – A Restricted Survey. *ZOR – Mathematical Methods of Operations Research*, 42:25–45, 1995.

[Heu96] Volker Heun. *Efficient Embeddings of Treelike Graphs into Hypercubes*. Berichte aus der Informatik. Shaker Verlag, Aachen, 1996.

[Hoc96] Dorit S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, 1996.

[HS86] D.S. Hochbaum and D.B. Shmoys. A unified approach to approximation algorithms for bottleneck problems. *J. ACM*, 33(3):533–550, 1986.

[HS87] D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *J. ACM*, 34(1):144–162, 1987.

[HS94] Reiner Hüttl and Michael Schneider. Parallel Adaptive Numerical Simulation. SFB-Bericht 342/01/94 A, SFB 342, Institut für Informatik, Technische Universität München, 1994.

[Hüt96] Reiner Hüttl. *Ein iteratives Lösungsverfahren bei der Finite-Element-Methode unter Verwendung von rekursiver Substrukturierung und hierarchischen Basen*. PhD thesis, Institut für Informatik, Technische Universität München, 1996.

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, Reading, MA, 1992.

[Joh74] David S. Johnson. Fast Algorithms for Bin Packing. *J. Comput. Syst. Sci.*, 8:272–314, 1974.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Pubishing Company, Redwood City, CA, 1994.

[KGV94] Vipin Kumar, Ananth Y. Grama, and Nageshwara Rao Vempaty. Scalable Load Balancing Techniques for Parallel Computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.

[KV87] Vipin Kumar and Nageshwara Rao Vempaty. Parallel depth-first search, Part II: Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[KZ93] Richard M. Karp and Yanjun Zhang. Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation. *J. ACM*, 40(3):765–789, 1993.

[LC90] Keqin Li and Kam-Hoi Cheng. On three-dimensional packing. *SIAM J. Comput.*, 19:847–867, 1990.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[Leo96] Stefano Leonardi. *On-line Resource Management with Application to Routing and Scheduling*. PhD thesis, Universitá di Roma "La Sapienza", 1996.

[LK87] Frank C.H. Lin and Robert M. Keller. The Gradient Model Load Balancing Model. *IEEE Transactions on Software Engineering*, SE-13(1):32–38, January 1987.

[LMR91] R. Lüling, B. Monien, and F. Ramme. Load Balancing in Large Networks: A Comparative Study. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing SPDP'91*, pages 686–689, Los Alamitos, 1991. IEEE Computer Society Press.

[LMS95] S. Leonardi and A. Marchetti-Spaccamela. On-line Resource Management with Applications to Routing and Scheduling. In Zoltán Fülöp and Ferenc Gécseg, editors, *Proceedings of the 22nd International Colloquium on Automata, Languages and*

Programming ICALP'95, volume 944 of *LNCS*, pages 303–314, Berlin, 1995. Springer-Verlag.

[LNRS92] F.T. Leighton, Mark J. Newman, Abhiram G. Ranade, and Eric J. Schwabe. Dynamic tree embeddings in butterflies and hypercubes. *SIAM J. Comput.*, 21(4):639–654, August 1992.

[Lud93] Thomas Ludwig. *Automatische Lastverwaltung für Parallelrechner*, volume 94 of *Reihe Informatik*. BI-Wissenschaftsverlag, Mannheim, 1993.

[McD89] C. McDiarmid. On the method of bounded differences. In J. Siemons, editor, *Surveys in Combinatorics*, volume 141 of *London Mathematical Society Lecture Note Series*, pages 148–188. Cambridge University Press, Cambridge, 1989.

[McD98] Collin McDiarmid. Concentration. In M. Habib, C. McDiarmid, J. Ramirez-Alfonin, and B. Reed, editors, *Probabilistic Methods for Algorithmic Discrete Mathematics*, volume 16 of *Algorithms and Combinatorics*, pages 195–248. Springer-Verlag, Berlin, 1998.

[MCP+98] Paul Messina, David Culler, Wayne Pfeiffer, William Martin, J. Tinsley Oden, and Gary Smith. The High-Performance Computing Continuum: Architecture. *Communications of the ACM*, 41(11):37–44, 1998.

[Mit96a] Michael Mitzenmacher. Load Balancing and Density Dependent Jump Markov Processes. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science FOCS'96*, pages 213–222, Los Alamitos, 1996. IEEE Computer Society Press.

[Mit96b] Michael David Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, Department of Computer Science, University of California at Berkeley, 1996.

[MM68] A. Meir and L. Moser. On packing of squares and cubes. *J. Comb. Theory*, 5:126–134, 1968.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Press Syndicate of the University of Cambridge, Cambridge, 1995.

[MSS95] Friedhelm Meyer auf der Heide, Christian Scheideler, and Volker Stemann. Exploiting storage redundancy to speed up randomized shared memory simulations. In Ernst W. Mayr and Claude Puech, editors, *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science STACS'95*, volume 900 of *LNCS*, pages 267–278, Berlin, 1995. Springer-Verlag.

[NXG85] Lionel M. Ni, Chong-Wei Xu, and Thomas B. Gendreau. A Distributed Drafting Algorithm for Load Balancing. *IEEE Transactions on Software Engineering*, SE-11(10):1153–1161, 1985.

[Rad96] Ralph Radermacher. *Eine Ausführungsumgebung mit integrierter Lastverteilung für verteilte und parallele Systeme*. PhD thesis, Fakultät für Informatik der Technischen Universität München, 1996.

[Rah96] Erhard Rahm. Dynamic Load Balancing in Parallel Database Systems. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Proceedings of the Second International EURO-PAR Conference on Parallel Processing EURO-PAR'96, Volume I*, volume 1123 of *LNCS*, pages 37–52, Berlin, 1996. Springer-Verlag.

[Reg97] H. Regler. *Anwenden von Algebraischen Mehrgittermethoden auf das Plazierproblem im Chipentwurf und auf die numerische Simulation von Strömungen*. PhD thesis, Technische Universität München, 1997.

[Röd97] Christian Röder. Classifiaction of Load Models. In Thomas Schnekenburger and Georg Stellner, editors, *Dynamic Load Distribution for Parallel Applications*, Teubner-Texte zur Informatik. Teubner Verlag, Stuttgart, 1997.

[RR93] H. Regler and U. Rüde. Layout optimization with Algebraic Multigrid Methods (AMG). In *Proceedings of the Sixth Copper Mountain Conference on Multigrid Methods*, pages 497–512. NASA, 1993.

[RR96] Reinhard Riedl and Lutz Richter. Classification of Load Distribution Algorithms. In *Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing PDP'96*, pages 404–413, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[RS98] Martin Raab and Angelika Steger. Balls into Bins - A Simple and Tight Analysis. In *Proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science RANDOM'98*, volume 1518 of *LNCS*, pages 159–170, Berlin, 1998. Springer-Verlag.

[Sal47] H.E. Salzer. The Approximation of Numbers as Sums of Reciprocals. *American Mathematical Monthly*, 54:135–142, 1947.

[San98] Peter Sanders. Randomized Priority Queues for Fast Parallel Access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, 1998. Special Issue on Parallel and Distributed Data Structures.

[Sch84] H.R. Schwarz. *Methode der finiten Elemente*, volume 47 of *Leitfäden der angewandten Mathematik (Teubner Studienbücher : Mathematik)*. Teubner Verlag, Stuttgart, second edition, 1984.

[Sch97a] Thomas Schnekenburger. Exemplary Load Distribution Concepts: A Classification. In Thomas Schnekenburger and Georg Stellner, editors, *Dynamic Load Distribution for Parallel Applications*, Teubner-Texte zur Informatik. Teubner Verlag, Stuttgart, 1997.

[Sch97b] Thomas Schnekenburger. General Classification of Load Distribution. In Thomas Schnekenburger and Georg Stellner, editors, *Dynamic Load Distribution for Parallel Applications*, Teubner-Texte zur Informatik. Teubner Verlag, Stuttgart, 1997.

[Sga94] Jiří Sgall. *On-Line Scheduling on Parallel Machines*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1994.

[Sga98] Jiří Sgall. On-line Scheduling. In Amos Fiat and Gerhard J. Woeginger, editors, *Online Algorithms : The State of the Art*, volume 1442 of *LNCS*, pages 196–231. Springer Verlag, Berlin, 1998.

[SHK95] Behrooz A. Shirazi, Ali R. Hurson, and Krishna M. Kavi, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[SKS92] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load Distributing for Locally Distributed Systems. *Computer*, 25(12):33–44, 1992.

[Son94] J. Song. A partially asynchronous and iterative algorithm for distributed load balancing. *Parallel Computing*, 20:853–868, 1994.

[SS84] John A. Stankovic and Inderjit S. Sidhu. An Adaptive Bidding Algorithm For Processes, Clusters and Distributed Groups. In *Proceedings of the Fourth International Conference on Distributed Computing Systems*, pages 49–59. IEEE Computer Society Press, 1984.

[SS97] Thomas Schnekenburger and Georg Stellner, editors. *Dynamic Load Distribution for Parallel Applications*. Teubner-Texte zur Informatik. Teubner Verlag, Stuttgart, 1997.

[SSA+94] Craig B. Stunkel, Dennis G. Shea, Bülent Abali, Mark Atkins, Carl A. Bender, Don G. Grice, Peter H. Hochschild, Douglas J. Joseph, Ben J. Nathanson, Richard A. Swetz, Robert F. Stucke, Michael Tsao, and Philip R. Varker. The SP2 Communication Subsystem. Research Report RC 19914, IBM Research Division, T.J. Watson Research, 1994.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202–208, 1985.

[Ste96] Volker Stemann. Parallel Balanced Allocations. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'96*, pages 261–269, New York, 1996. ACM Press.

[Ste97] Georg Stellner. Migration Mechanisms. In Thomas Schnekenburger and Georg Stellner, editors, *Dynamic Load Distribution for Parallel Applications*, Teubner-Texte zur Informatik. Teubner Verlag, Stuttgart, 1997.

[SWW95] David B. Shmoys, Joel Wein, and David P. Williamson. Scheduling Parallel Machines On-Line. *SIAM J. Comput.*, 24(6):1313–1331, 1995.

[Ver98] Jacques Verriet. *Scheduling with communication for multiprocessor computation*. PhD thesis, Faculteit Wiskunde & Informatica, Universiteit Utrecht, 1998.

[VLL90] B. Veltman, B.J. Lageweg, and J.K. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16:173–182, 1990.

[WC92] Qingzhou Wang and Kam Hoi Cheng. A Heuristic of Scheduling Parallel Tasks and its Analysis. *SIAM J. Comput.*, 21(2):281–294, April 1992.

[Wes96] Douglas B. West. *Introduction to Graph Theory*. Prentice-Hall, Upper Saddle River, NJ, 1996.

[Wil91] David Williams. *Probability with Martingales*. Cambridge University Press, Cambridge, 1991.

[WLR93] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, 1993.

[XL94a] Cheng-Zhong Xu and Francis C.M. Lau. Iterative Dynamic Load Balancing in Multicomputers. *J. Opl. Res. Soc.*, 45(7):786–796, 1994.

[XL94b] Cheng-Zhong Xu and Francis C.M. Lau. Optimal Parameters for Load Balancing with the Diffusion Method in Mesh Networks. *Parallel Processing Letters*, 4(1–2):139–148, 1994.

[XL97] Chengzhong Xu and Francis C.M. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, 1997.

Index

D

E

F

H

I

J

L

M

N

O

P

R

S

T

U

V

W