Design and Evaluation of Job Scheduling Strategies for Grid Computing

Doctorial Thesis
Ramin Yahyapour

Genehmigte Dissertation zur
Erlangung des akademischen Grades eines Doktors an der
Fakultät für Elektrotechnik und Informationstechnik
Universität Dortmund

Computer Engineering Institute
Lehrstuhl für Datenverarbeitungssysteme

Acknowledgements

It is my pleasure to thank all the people who supported me to make this thesis possible. In particular, i would like to express my thanks to my advisor, Prof. Dr. Uwe Schwiegelshohn, who supervised my research work and encouraged me every time. I also want to thank Prof. Dr. Burkhard Monien for his helpful support.

Further, i wish to thank all members of the Computer Engineering Institute at the University of Dortmund for providing a pleasant and always helpful working atmosphere. I am especially grateful to my research colleagues, Volker Hamscher, Carsten Ernemann, and Achim Streit, with whom i had the pleasure to work with. I am also deeply indebted to all undergraduate students who contributed to my research work.

Finally, i have to thank my parents who supported my in every possible way with their love, encouragement and patience. I would further like to thank Petra for her help and patience.

1. Gutachter:	Prof. Dr.-Ing. Uwe Schwiegelshohn
2. Gutachter:	Prof. Dr. Burkhard Monien
Tag der Einreichung:	16. Juli 2002
Tag der Prüfung:	13. November 2002
Stand vom:	20. November 2002

Abstract

Grid computing is intended to offer an easy and seamless access to remote resources. The importance of grid computing can be seen by the attention it gained recently in research and industry support. The scheduling task of allocating these resources automatically to user jobs is an essential part of a grid environment. In this work we discuss the evaluation and design of different scheduling strategies. To this end, we present a concept for the design process of such a scheduling system. The evaluation of scheduling algorithms for single parallel machine is done by theoretical analysis and by simulation experiments. The theoretical approach by competitive analysis lead to bounds for the worst-case scenarios. As we are especially interested in the scheduling performance in a real system installation, simulations have been applied for further evaluation. In addition to the theoretical analysis, we show that the presented preemptive scheduling algorithm is also efficient in terms of makespan and average response time in a real system scenario if compared to other scheduling algorithms. In some of the examined scenarios the algorithm could outperform other common algorithms such as backfilling. Based on these results, scheduling algorithms for the grid environment have been developed. On one hand, these methods base on modifications of the examined conventional scheduling strategies for single parallel machines. On the other hand, a scheduling strategy with a market economic approach is presented. These methods are also analyzed and compared by simulations for a real workload environment. The results show that the economic approach produces similar or even better results than the conventional strategies for common criteria as the average weighted response time. Additionally, the economic approach delivers several additional advantages, such as support for variable utility functions for users and owner of resources. As a proof of concept a possible architecture of a scheduling environment is presented, which has been used for the evaluation of the presented algorithms. The work ends with a brief conclusion on the discussed scheduling strategies and gives an outlook on future work.

Contents

Chapter 1

Introduction

The technological progress over the last decades is especially apparent in the vast improvements in computer technology. The computing power of a single computer increased dramatically from 1980 until 2000. Although Moore's law has repeatedly been considered to loose validity as physical limits are reached, it still holds very well for several aspects in computer technology.

Nevertheless, computing power has always been a limited resource although computers became significantly faster. On one hand, this is caused by new and complex applications which just became feasible by new technology. On the other hand, many existing applications grow in demand for computing power. For instance, simulations get more detailed and complex; databases grow to new dimensions; the Grand Challenge problems, e.g. the human genome project, or the sophisticated computer aided development of new drugs are becoming computational tractable. These are all examples leading to the conclusion that there was and still is an ongoing need for more computing power.

Besides making processors - and consequently single processor computer system - faster, the combination of processors to parallel computers is a suitable way to achieve better performance. This makes parallel computers - leaving out special purpose computers - the fastest machines according to common metrics such as the installed GFlops by the LinPack benchmark in the TOP 500 list [45]. For now quite some time the building of clusters is an interesting subject for high-performance computing. Linking single and autonomous computers like workstations via a communication network to a cluster is a cost-efficient alternative for some but not all applications. The rise of the internet in the past decade is an additional development that dramatically changed the view of our information society. The ability to be connected and linked to the internet is omnipresent. The trend of connecting everything via a network has now even reached commodity devices. The simple connection of our computing resources to the net is yet common standard and practice.

Combining the need for computing power with the connection of these resources via the network leads naturally to concepts that have gained a lot of attention by research in the past decade. The term *metacomputing* was established in 1987 by Smarr and Catlett [68]. This approach has similarities to cluster computing as computers are connected via a network. Here, the combined resources are high performance computers. Additionally, the resources are usually geographically distributed. The user of a so called metacomputer need not to be aware where his computing task is actually executed. Metacomputing comes well along the lines of the former slogan by Sun Microsystem: "*The network is the computer*". *Cluster computing* in comparison denotes a computing where single workstations are interconnected and usually belongs to the same owner or administrative instance. Metacomputing is considered to connect geographically remote computing resource from different owners, who usually do not even know each other or the metacomputing users.

Recently, the term metacomputing has been substituted by *computational grid* or just the *grid* [35,53]. This term still describes a similar subject. While metacomputing was limited to compute resources, grid computing takes a broader approach on the resources that are connected. Here, devices for visualization as well as data storage can be included. Metacomputing - besides several limited projects - never got common practice, grid computing became a new buzz word and gained a lot of attention. An example is the Global Grid Forum (GGF) [39] that has been launched to coordinate the different initiatives in Europe (e.g. EGrid [14]), U.S. (e.g. Globus [31]) and Asia.

The term grid has been coined during the Globus project [35] and alludes to the electrical power grid which provides all users with electrical power just on demand without deeper insight how and where the power has actually been generated. The analogy is to provide computational power on demand to all users without prior knowledge where the underlying resources come from. Note, that as we focus on computational resources in this work, we will use the terms metacomputing, computational grid or metasystem mutually exchangeable, denominating the same subject.

Overview

The research presented in this work has been focused on resource management and the scheduling of parallel jobs in a distributed environment. However, the results for this subject are not limited to grid computing as the same concepts can be applied to other application fields. This ranges from logistics, telecommunications to mobile devices. In this typical scenario independent users generate workload by submitting job requests for resources. It is the task of the management system to decide when and where a job is finally executed and resources are allocated to the job. This is done under several constraints in terms of the job requirements, the user's objective and the resource owner's policy to grant access to the resources. The application of this task in grid computing allows the practical study and research on core topics which can be easily extended to other applications.

Therefore, the work presented in this document is dedicated to several issues in grid computing. The main topic is the scheduling task, which is an important part for a resource management system. The scheduling process is responsible for the above mentioned allocation of resources to a user request. This includes the task of finding suitable resources for such a request, deciding which resources to chose from the actual available and when to start the particular application. As there are usually many requests submitted while there are only a limited amount of resources, this leads to resource conflicts which must be settled by the scheduling algorithm. This is done under the premise to generate the most efficient schedules. The design and evaluation of such a scheduling system is subject of this work.

To this end, we present a concept for designing and evaluating scheduling systems which is based on scheduling policies supplied by the administrator. These policies are used to derive a suitable scheduling objective which is subsequently applied in the design and evaluation process of an algorithm.

The evaluation of such a scheduling system is an import part of the presented design process. In this work, theoretical as well as the experimental approaches are used to evaluate existing and new algorithms. As we will see, conventional algorithms that are proposed by theory are frequently not suitable for practical usage and often focus on the off-line scenario. But in real installations jobs are submitted independently and continuously with an unknown or an user supplied estimated execution time. Additionally, most algorithms do not consider user and administrator objectives at the same time, which is one reason why still list scheduling methods are frequently used for real installations. Therefore, we introduce in this work a concept of fairness to scheduling and present a new algorithm called PFCFS which is fair from a user point of view while also regarding the optimization criteria that are anticipated by administrators. In this case, the makespan as well as the weighted completion time is considered as a performance metric. The selection of a job weight based on the Smith ratio is proposed which has, for instance, the benefit that it does not favor jobs with less or more parallelism. In a first step, the algorithm is examined by theoretical competitive analysis. We prove that the algorithm achieves a constant competitive ratio for both the makespan and the weighted completion time.

However, we also discuss the limitation that the theoretical analysis does not take the workload into account. Therefore the theoretical results are useful to examine the general behavior of an algorithm. However, we need additional evaluation to gain information on the system performance in real workload scenario. Consequently, we additionally apply experimental analysis on the presented algorithm. Simulations are used to achieve information on its practical performance. These results and the comparison with conventional algorithms show the benefit of the introduced algorithms. Based on these results, we make the transition to grid scheduling. The discussed approach on single parallel machine is used to propose several grid scheduling algorithms that have been derived from the conventional methods for single parallel computers. These methods are again evaluated and analyzed by simulation.

By the very nature of grid computing in which resources are owned and maintained by different individuals who are geographically distributed and often do not even know each other, the usage of economic models seem appropriate. These models have been in discussion for quite some time for computational problems. While most of these economic approaches suffer from centralistic concepts with the introduction of markets and auctions which often require the revelation of information on scheduling policies, we present in this work a flexible economic scheduling model that takes

our considerations for grid usage into account. Here, our model supports features as e.g. advance reservation, guarantees, information hiding, or support for multi-site applications. Additionally, users can supply individual scheduling objectives for each submitted job. Similarly, the administrators can also provide individual scheduling objectives for each job and each resource. Moreover, the infrastructure is optimized by a domain-based design which is not focused on a single central scheduler or information service which lead to a performance bottleneck or single-point of failure. Instead it provides site-autonomy and keeps full control of all local resources in the local domain and management realm. For the resource determination and communication we propose a new peer-to-peer approach.
We evaluate the presented economic method by simulations with different workloads that are derived from real traces and show that it produces results in terms of common performance metrics as average weighted response time and utilization in the range of the conventional scheduling algorithms. In several simulations the conventional strategies have even been outperformed. In addition, the economic model has the great advantage that it can deal with arbitrary scheduling objectives and can for example optimize for other criteria, as for instance the cost.

While it is important that a scheduling method produces ‘good’ scheduling results, it is as important as that it is implementable. As a proof of concept, we present a suitable scheduling infrastructure for grid computing which has been implemented in the NWIRE project. This architecture exploits the domain-based concept and the peer-to-peer request architecture. The interfaces are flexible enough to support conventional scheduling algorithms as well as economic variants. Moreover, each domain or resource can have different and specialized schedulers which can still interact with others by submitting requests and returning offers. Therefore, the specification of the description of these messages is of special interest. Within our implementation, we also developed a specification for this description language which is briefly introduced in this work. Note that all of our evaluations have been performed in the scheduling framework that has been implemented in the NWIRE project.

Document Structure

Overall, this work is divided into 8 chapters. We will first discuss the scheduling problem and present our concept for the design process of a scheduling system in Chapter 2. The evaluation of a scheduling algorithm is an important part of this process. We will examine the theoretical analysis of such algorithms in Chapter 3. As scheduling for single parallel machines has been subject to research for years, we start our design process in this area and use theoretical competitive analysis for the evaluation process. As we will see, the theoretical approach gives as information on the worst-case behavior of our algorithms. As we are especially interested in the scheduling performance in a real environment, we extend our scope to experimental analysis. Therefore, we use actual workload simulations to evaluate the scheduling algorithms. The method and its results are discussed in Chapter 4.

Based on these results we will make the transition to scheduling for a grid environment and the evaluation by simulation results for this scenario. In Chapter 5 we first present scheduling methods derived from conventional scheduling algorithms for single parallel machine . Next, we propose a scheduling model based on market economic methods in Chapter 6. Both approaches are evaluated and compared by simulation results. Herein, we show that economic methods can compete with the algorithms based on conventional scheduling methods and even outperform them. Furthermore, we discuss the additional advantages of economic algorithms in grid computing in terms of flexibility and features. As a proof of concept, in Chapter 7 we present and discuss a scheduling infrastructure that supports conventional as well as market economic scheduling models. This thesis ends with a brief conclusion and an outlook on possible extensions and future work on the presented models.

Part I

Designing a Scheduling System

Chapter 2

Design of a Scheduling System

In this chapter we first focus on the actual design of a scheduling system. The administrator or maintainer of a parallel machine - participating in a grid environment or not - is interested in a scheduling method that provides "good" scheduling results. To this end, we present a general concept of designing such a scheduling system in this chapter.

Currently, computers are usually not dedicated to the grid. Moreover, we have often single parallel machines that are used by a local user community. And additionally these machines may voluntarily participate in the grid. Overall scheduling in a grid environment is an extension to the scheduling problem on local parallel systems. However, grid scheduling has distinct requirements that we will address later in this work. Nevertheless, general job scheduling for parallel processors has been subject to research for quite some time. Therefore, we will start with the scheduling problem for these single parallel machines. Later, in Chapter 5, the scope is extended to the scenario of meta- and grid-computing.

Before we start with the design of a scheduling system, we give a short explanation of our general job scheduling problem.

2.1 Scheduling Problem

Generally, it is the task of the scheduling system to allocate resources to a certain application request for a certain amount of time. This problem occurs in many areas and, as mentioned before, has been subject to research for a long time. In the context of this work scheduling deals with the allocation of resources like processor nodes or network bandwidth to user requests for mostly computational applications. These requests are usually called *jobs*. A job consists of information about the requirements on the resources that are necessary to execute this particular job. A typical example is a computational job which can only be executed on a specific type of computer system. It is the task of a scheduling system to decide where and when such a job is executed.

In our model, a parallel job schedule S determines for each job i its starting time s_i and the subset of nodes $\mathcal{M}_i$ assigned to this job. The starting time of a job must be greater or equal than its submission time or release date r_i. The node subsets of two jobs executing concurrently must be disjoint. That means that we use space-sharing instead of time sharing. The job i requests m_i number of nodes and needs an execution time p_i until completion. The completion time $t_i(S)$ of a job i in schedule S depends on the execution time and the starting time s_i.

2.1.1 Single Parallel Machine

As mentioned before, we begin with the scheduling problem for a single parallel machine [12]. Scheduling for such a massive parallel processing system (MPP) differs significantly from scheduling for a single processor or for smaller SMP system (symmetric multi-processing). The scheduling in the context of high-performance computing as presented in this work should not be mistaken for processor or instruction scheduling on a single machine. There, instructions, processes and threads are executed quasi-simultaneously by multi-tasking. A task scheduler of the operating system on a single machine usually performs a switching between tasks by variants of round-robin strategies. In this case the resources are used in a time-sharing fashion.

Here, in the area of high-performance computing, jobs are usually assigned exclusively to processors of a MPP system. This is partially due to the fact that these jobs often represent large and complex applications. These applications often require a large amount of memory which makes time-sharing inefficient as several jobs on the same processing node may lead to severe memory swapping. Also a job can run in parallel on different nodes. As these applications typically also require communication between the different job parts it would lead to a drawback in performance if job parts are not active at the same time or if some job parts need significantly longer due to multi-tasking on that node. Therefore MPP systems are predominantly used in a space-sharing fashion instead of time-sharing.

2.1.2 Machine Model

In the following a machine model is given which is used for the later discussions in this work on evaluation of scheduling strategies.

We assume a MPP architecture where each node contains one or more processors, main memory, and local hard disks while there is no shared memory between different nodes. The system may contain different types of nodes characterized e.g. by their type and number of processors, by the amount of memory or by the specific task this node is supposed to perform. An example for the last category are those nodes which provide access to mass storage. This model comes close to an IBM RS/6000 SP parallel computer. In particular, IBM SP2 architectures supports different kind of processors nodes. Currently, there are three types of nodes: *thin nodes*, *wide nodes*, and *high nodes*. Wide or high nodes with a large number of expansion slots are usually associated with server functionalities. They contain more memory than a thin node while there is little difference in processor performance. Recently introduced nodes have a SMP architecture and can contain several processors . However, it should be noted that still most installations are predominantly equipped with thin nodes. In 1997 only 48 out of a total of 512 nodes in the CTC RS/6000 SP2 were wide nodes while no high nodes were used at all. Thin nodes have a better form factor to include more nodes into the available frame space. This is the typical configuration for systems that are designed for computational intense usage. Although in most installations the majority of nodes have the same or a similar amount of memory, a wide range of memory configurations are possible. For instance, the mentioned CTC RS/6000 SP2 contains nodes with memory ranging from 128 MB to 2048 MB with more than 80% of the nodes having 128 MB and an additional 12% being equipped with 256 MB. Additional information on the CTC SP2 machine is given by Hotovy in [44]. We use the CTC example throughout this work as a reference. Additionally, it is assumed that all nodes on a machine are identical. This coheres with the observation mentioned above that large scale MPP systems for computational purposes consists predominantly of homogeneous partitions.

Fast communication between the nodes is achieved via a special interconnection network. In our model, this network does not prioritize clustering of any subset of nodes over others [24] such as in a hypercube or a mesh, i.e. the communication delay and the bandwidth between any pair of nodes is assumed to be constant. These assumptions come also close to the IBM RS/6000 SP2 architecture. The SP2 switch network scales with the number of nodes and does not favor certain node combination.

2.1.3 Job Model

In our examined scenario the scheduling system receives a stream of job submission data and produces a valid schedule. We use the term ‘stream’ to indicate that submission data for different jobs need not arrive at the same time. Also the arrival of any specific data is not necessarily predictable, that is, the scheduling system may not be aware of any data arriving in the future. Therefore, the scheduling system must deal with the so called ‘on-line’ behavior.

Further, we do not specify the amount and the type of job submission data. Different scheduling systems may accept or require different sets of submission data. For us submission data comprise all data which are necessary to determine a schedule. However, a few different categories can be distinguished:

- **User Information**: These data may be used to determine job priorities. For instance, the jobs of some user may receive faster service at a specific location while other jobs are only accepted if sufficient resources are available.
- **Resource Requests**: These data specify the resources which are requested for a job. Often they include the necessary number and type of processors, the amount of memory as well as some specific hardware and software requirements. Some of these data may be estimates, like the execution time of a job,

or describe a range of acceptable values, like the number of processors for a malleable job (see Section 2.1.3).

- **Scheduling Objectives**: These data may help the scheduling system to generate 'good' schedules. For instance, a user may state that he needs the result by 8am the next morning while an earlier job completion will be of no benefit to him. Other users may be willing to pay more if they obtain their results within the next hour.

Of course other submission data are possible as well. Job submission data are supplied by the user and first available to system when a job is submitted for execution. Nevertheless, some systems may also allow reservation of resources before the actual job submission. Such a feature is especially beneficial for multi-site metacomputing [64]. Especially important for scheduling is information on the number of requested resources and on the execution time of a job. The execution time may be an estimate by the user and is often used as a maximum processing time which a job must not exceed. Usually the scheduling system terminates a job if it reaches the supplied execution length. Section 2.5 gives an example for the need to provide information on the job execution length at submission time. The backfilling algorithm presented in that Section needs this information to determine the job execution order.

Note, that some job parameters may depend on the actual job allocation that is generated by the scheduler. As an obvious example the job execution length may depend in real systems on the available number of processors and the CPU speed. More available processing nodes usually result in a decrease of the execution time. This is often taken into account by allowing a generic description of such parameters relative to other job attributes.

In addition, some technical data are often required to start a job. That is for example the name and the location of the input data files of the job. But as these data do not affect the schedule if they are correct, we ignore them here. Finally note that the submission of erroneous or incorrect data is also possible. However, in this case a job may be immediately rejected or fail to run.

In our model, a job *runs to completion* after it is started. That is, the job will not be interrupted after its start. Alternatively, we speak of *preemption* if a job is temporarily stopped on its processor set. Later on, the job is resumed on the same processors. The ability to preempt parallel jobs simultaneously on all processors in the job's current partition is often also referred to as gang-scheduling [25]. Furthermore, if additionally the set of resources can be changed during run-time, we speak of *migration*. However, in our model we assume that the resource set is fixed during job execution.

Moreover, the resource requirement is also fixed. That means that the number of resources that are requested for a job is given by the user and cannot be modified. These jobs are often referred to as *rigid* jobs, see [28]. In comparison, jobs are called *moldable* if the partition size cannot be modified and the job is capable to run on different numbers of processors, see also [28]. The user supplies a range or a list of suitable partition sizes for a job. Nevertheless, this number is fixed after the job is started. For *malleable* jobs, this limitation is also removed. Here, even the partition size may change during run-time.

2.2 Design Process for a Scheduling System

Next, we want to focus on the actual design of a scheduling system. The creation of a scheduling algorithm which produces 'good' schedule results is a complex task. This is especially true for massively parallel processors (MPPs) where many users with a multitude of different jobs share a large amount of system resources. While job scheduling does not affect the results of a job, it may have a significant influence on the efficiency of the system. For instance, a good job scheduling system may reduce the number of MPP nodes that are required to process a certain amount of jobs within a given time frame. Or it may permit to execute more jobs on the resources of a machine in the same amount of time. Therefore, the job scheduling system plays an important role in the management of computer resources. This is especially the case as these resources usually represent a significant investment for a company or institution in the case of MPPs. The difference between a more and a less efficient scheduling system is often comparable to providing additionally hardware resources to achieve the same system performance.

Hence, the availability of a good job scheduling system is in the interest of the owner or administrator of an MPP. It is therefore not surprising that in the past new job scheduling methods have frequently been introduced by institutions which were among the first owners of MPPs like, for instance, ANL [51], CTC [57] or NASA Ames [58]. On the other hand, machine manufacturers often showed only limited interest in this issue as they frequently seem to have the opinion that “machines are not sold because of superior job schedulers”. Moreover, the design of a job scheduling system must be based on the specific environment of a parallel system. Consequently, administrators of MPPs will always remain to be involved in the design of job scheduling systems.

Thus, we first want to take a closer look at the schedule. As mentioned before a schedule is an allocation of system resources to individual jobs for certain time periods. Therefore, a schedule can be described by all the time instances where a change of resource allocation occurs as long as either this change is initiated by the scheduling system or the scheduling system is notified of this change. To illustrate this restriction assume a job being executed on a processor that is also busy with some operating system tasks. Here, we do not consider changes of resource allocation which are due to the context switches between OS tasks and the application. Those changes are managed by the system software without any involvement of our scheduling system.

For a schedule to be valid some restrictions of the hardware and the system software must be observed. For instance, a parallel processor system may not support gang scheduling or require that at most one application is active on a specific processor at any time. Therefore, the validity constraints of a schedule are defined by the target machine. A valid schedule defines resource allocations that do not violate the physical constraints of the supplied job requirements and the available resources. We assume that a scheduling system does not attempt to produce an invalid schedule. However, note that the validity of a schedule is not affected by falsely supplied properties of a submitted job if those do not comply with the actual job execution. For instance, if not enough memory is requested and assigned to a job, the job will simply fail to run. Also, a schedule depends upon other influences which cannot be controlled by the scheduling system, like the sudden failure of a hardware component. But this does not mean that the resulting schedule is invalid. Therefore, the final schedule is only available after the execution of all jobs.

In our model the scheduling system is divided into the following 3 parts [48]:

1. A *scheduling policy*,
2. an *objective function* and
3. a *scheduling algorithm*.

We describe these scheduling parts and their dependencies in more detail in the following sections. Later on, we compare the evaluation of scheduling systems with the evaluation of computer architectures.

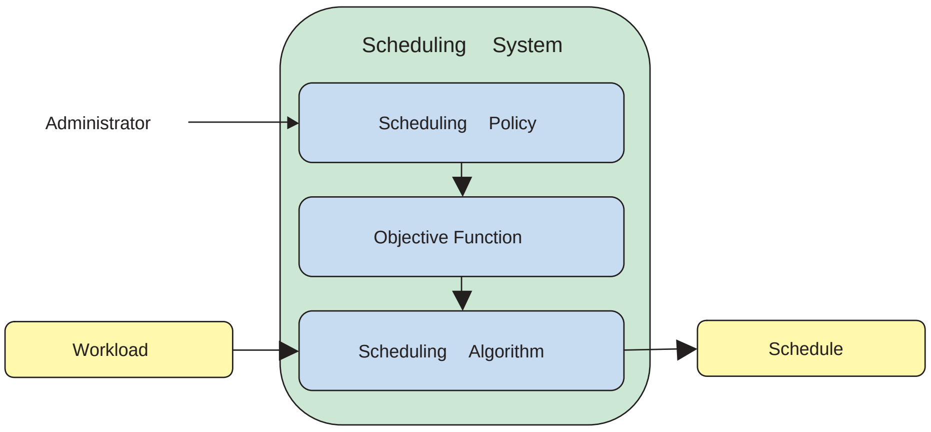

Figure 2.1: Scheduling System.

2.3 Scheduling Policy

The first component of a scheduling system is the scheduling policy on which it is based. This policy is defined by the owner or administrator of a machine as shown in Figure 2.1. In general, the scheduling strategy is a collection of rules to determine the resource allocation for all submitted jobs. There are usually not enough resources available to satisfy all jobs immediately: as a consequence resource conflicts occur. The scheduling strategy is responsible to settle these resource conflicts. To better illustrate our approach, we give an example:

Example 1 *The department of chemistry at University A has bought a parallel computer which was financed to a large part by the drug design lab. The department establishes the following rules for the use of the machine:*

1. *All jobs from the drug design lab have the highest priority and must be executed as soon as possible.*
2. *100 GB of secondary storage is reserved for data from the drug design lab.*
3. *Applications from the whole university are accepted but the labs of the chemistry department have preferred access.*
4. *Some computation time is sold to cooperation partners from the chemical industry in order to pay for machine maintenance and software upgrades.*
5. *Some computation time is also made available to the theoretical chemistry lab course during their scheduled hours.*

Note that these rules are hardly detailed enough to generate a schedule. But they allow a fuzzy distinction between *good* and *bad* schedules. Also, there may be some additional general rules which are not explicitly mentioned, like ‘Complete all applications as soon as possible if this does not contradict any other rule’. There may be also specific rules to the customer or users that has to be obeyed as for example cost modelling or statements on quality of service. These rules have not been included in our example.
Finally, some conflicts between those rules may occur and must be resolved. For instance in Example 1, some jobs from the drug design lab may compete with the theoretical chemistry lab course. Hence, in our view a good scheduling policy has the following two properties:

1. If some rules produce any conflicts, the scheduling policy contains rules to settle these conflicts. This includes for example a prioritization if different rules may apply.
2. It must be possible to implement the scheduling policy in an algorithm.

We believe that there is no general method to derive a scheduling policy. Also there is no need to provide a very detailed policy with clearly defined quotas. In many cases this will result in a reduction of the number of *good* schedules. For instance, it would not be helpful at this point to demand that 5% of the computation time is sold to the chemical industry in Example 1. If there are only a few jobs from the drug design lab then the department would be able to earn more money by defining a higher industry quota. Otherwise, the department must decide whether to obtain other funding for the machine maintenance or to reduce the priority of some jobs of the drug design lab. This issue will be further discussed in Section 2.6.

2.4 Objective Function

Alternatively, a scheduling system can be applied to a multitude of different streams of submission data and the resulting schedules can be evaluated. This requires a method to automatically determine the quality of a schedule. Therefore, an objective function must be defined that assigns a scalar value, the so called *schedule cost*, to each schedule. Note that this property is essential for the mechanical evaluation and ranking of a schedule. In the simplest case all *good* schedules are mapped to 1 while all *bad* schedules obtain the value 0. Most likely however, this kind of objective function will be of little help. To derive a suitable objective function an approach based on multi criteria optimization can be used, see e.g. [71]:

1. For a typical set of jobs determine the Pareto-optimal schedules based on the scheduling policy.
2. Define a partial order of these schedules.
3. Derive an objective function that generates this order.
4. Repeat this process for other sets of jobs and refine the objective function accordingly.

To illustrate Steps 1 and 2 of our approach we consider Rules 1 and 5 of Example 1. Assume that both rules are conflicting for the chosen set of job submission data. Therefore, we determine a variety of different schedules, see Figure 2.2. Note that we are not biased toward any specific algorithm in this step. We are primarily interested in those schedules which are *good* with respect to at least one criterion. Therefore, at first all Pareto-optimal schedules are selected. Those schedules are indicated by bullets in Figure 2.2. Next, a partial order of the Pareto-optimal schedules is obtained by applying additional conflict resolving rules or by asking the owner. In the example of Figure 2.2 numbers 0, 1 and 2 have been assigned to the Pareto-optimal schedules in order to indicate the desired partial order by increasing numbers. Here any schedule 1 is superior to any schedule 0 and inferior to any schedule 2 while the order among all schedules 1 does not matter.

The approach is based on the availability of a few typical sets of job data. Further, it is assumed that each rule of the scheduling policy is associated with single criterion

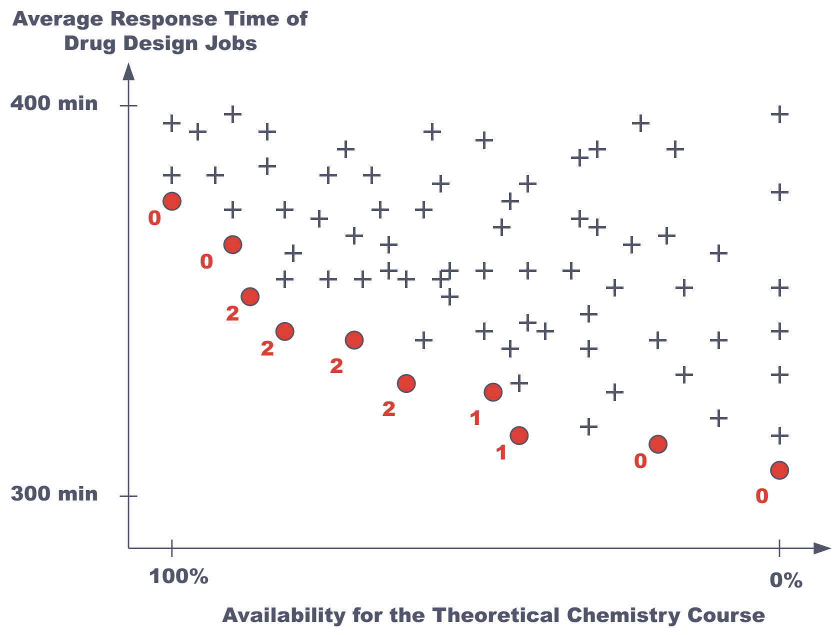

Figure 2.2: Pareto Example for 2 Rules

functions, like Rule 4 of Example 1 with the function 'amount of computation time allocated to jobs from the cooperation partners from industry'. If this is not the case, complex rules must be split.

Now, it is possible to compare different schedules if the same objective function and the same set of jobs is used. Further, there are a few additional aspects which are also noteworthy:

- Schedules can be compared even if they do not have the same target architecture. The partial order of the Pareto-optimal schedules can be generated for different resource configurations. This allows the analysis of different system selections for a given job set.
- An up front calculation of the *schedule cost* may not be possible. Most likely, it will be necessary to execute all jobs first before the correct *schedule cost* can be determined.
- It is also possible to compare two schedules which are based on the same job set. In this case the results for the different schedules can be used to analyze the influence of different submission strategies (e.g. queue configuration or user policies).

Thus, schedules can even be used as a criterion for system selection if desired.

At many installations of large parallel machines simple objective functions are used, like the job throughput, the average job response time, the average slowdown of a job or the machine utilization, see [27]. We believe that it cannot be decided whether those objective functions are suitable in general. For some scheduling policy they may be the perfect choice while they should not be used for another set of rules. Also, it is not clear whether the use of those 'simple' objective functions allows an easier design of scheduling systems.

2.5 Scheduling Algorithm

The scheduling algorithm is the last component of a scheduling system. It has the task to generate a valid schedule for the actual stream of submission data in an online fashion. A good scheduling algorithm is expected to produce very good if not optimal schedules with respect to the objective function while not taking 'too much' time and 'too many' resources to determine the schedule. Overall, the algorithm must be implementable in a real system.

In order to obtain good schedules the administrator of a parallel machine is therefore faced with the problem to pick an appropriate algorithm among a variety of suboptimal ones. He may even decide to design an entirely new method if the available ones do not yield satisfactory results. The selection of the algorithm is highly dependent on a variety of constraints:

- Schedule restrictions given by the system, like the availability of dynamic partitioning or gang scheduling.
- System parameters, like I/O ports, node memory and processor types.
- Distribution of job parameters, like the amount of large or small jobs.
- Availability and accuracy of job information for the generation of the schedule. For instance, this may include job execution time as a function of allocated processors.
- Definition of the objective function.

Frequently, the system administrator will simply take scheduling algorithms from the literature and modify them to his needs. Then, he picks the best one from his algorithm candidates. After making sure that his algorithm of choice actually generates valid schedules, he also must decide whether it makes sense to look for a better algorithm. Therefore, it is necessary to evaluate those algorithms. We distinguish the following methods of evaluation:

1. Evaluation using algorithmic theory
2. Simulation with job data derived from
    - an actual workload
    - a workload model

In general, theoretical evaluation is often not well suited for scheduling algorithms on real systems as it will be discussed in Chapter 3. Occasionally, this method is used to determine lower bounds for schedules. These lower bounds can provide an estimate for a potential improvement of the schedule by switching to a different algorithm. However, it is very difficult to find suitable lower bounds for complex objective functions.

Alternatively, an algorithm can be fed with a stream of job submission data. The actual schedule and its cost are determined by simulation with the help of the complete set of job data. The procedure is repeated with a large number of input data sets. The reliability of this method depends on several factors:

- Availability of correct job data,
- Compliance of the used job set with the job set on the target machine.

Actual workload data can be used if they are recorded on a machine with a user group such that sufficient similarity exists with the target machine and its users. This is relatively easy if traces from the target machine and the target user community are available. Otherwise some traces must be obtained from other sources, see e.g. [21]. In this case it is necessary to check whether the workload trace is suitable. This may even require some modifications of the trace.

Also note that the trace only contains job data of a specific schedule. In another schedule these job data may not be valid as is demonstrated in Examples 2 and 3:

Example 2 *Assume a parallel processor that uses a single bus for communication. Here, independent jobs compete for the communication resource. Therefore, the actual performance of job i depends on the jobs executed concurrently with job i.*

Example 3 *Assume a machine and a scheduling system that support adaptive partitioning for moldable jobs (see Section 2.1.3). In this case, the number of resources allocated to job i again depends on other jobs executed concurrently with job i.*

Also, a comprehensive evaluation of an algorithm frequently requires a large amount of input data that may not be available by accessible workload traces.

If accurate workload data are not available, then artificial data must be generated. To this end a workload model is used. Again conformity with future real job data is essential and must be verified. On the other hand, this approach is able to overcome some of the problems associated with trace data simulation, if the workload model is precise enough. For a more detailed discussion of this subject, see also [27].

Unfortunately, it cannot be expected that a single scheduling algorithm will produce a better schedule than any other method for all used input data sets. In addition the resource consumption of the various algorithms may be different. Therefore, the process of picking the best suited algorithm may again require some form of multi criteria optimization.

In the following several scheduling algorithms are briefly introduced which will be used later in this work for evaluation purposes and comparisons. These are several typical algorithms that are either known from theory (Gary-Graham, Smart, PSRS) and/or from practical use on MPP systems (FCFS and Backfilling)

FCFS

First-Come-First-Serve (FCFS) is a well known scheduling scheme that is used in some production environments. All jobs are ordered by their submission time. Then a greedy list scheduling method is used, that is the next job in the list is started as soon as the necessary resources are available. This method has several advantages:

1. It is fair as the completion time of each job is independent of any job submitted later.
2. No knowledge about the execution time is required.
3. It is easy to implement and requires very little computational effort.

However, FCFS may produce schedules with a relatively large percentage of idle nodes especially if many highly parallel jobs are submitted, as in [78]. Therefore, FCFS has been replaced by FCFS with some form of **backfilling** at many locations including the CTC. Nevertheless, the administrator does not want to ignore FCFS at this time as a theoretical study has recently shown that FCFS may produce acceptable results for certain workloads [61].

Backfilling

The backfilling algorithm has been introduced by Lifka [51]. It requires knowledge of the job execution times and can be applied to any greedy list schedule. If the next job in the list cannot be started due to a lack of available resources, then backfilling tries to find another job in the list which can use the idle resources but will not postpone the execution of the next job in the list. In other words, backfilling allows some jobs down the list to be started ahead of time.

There are 2 variants of backfilling as described by Feitelson and Weil [29]:

EASY backfill is the original method of Lifka. It has been implemented in several IBM SP2 installations. While EASY backfill will not postpone the *projected* execution of the next job in the list, it may increase the completion time of jobs further down the list, see [29].

Conservative backfill will not increase the *projected* completion time of a job submitted before the job used for backfilling. On the other hand conservative backfill requires more computational effort than EASY.

However, note that the statements regarding the completion time of skipped jobs in the list are all based on the provided execution time for each job. Backfilling may still increase the completion time of some jobs compared to FCFS as in an on-line scenario another job may release some resources earlier than assumed. In this case it is possible that a backfilled job may prevent the start of the next job in the list. For instance, while some active job is expected to run for another 2 hours it may terminated within the next 5 minutes. Therefore, backfilling with a job having an expected execution time of 2 hours may delay the start of the next job in the list by up to 1 hour and 55 minutes.

List Scheduling (Garey and Graham)

The classical list scheduling algorithm by Garey and Graham [36] always starts the next job for which enough resources are available. Ties can be broken in an arbitrary fashion. The algorithm guarantees good theoretical bounds in some on-line scenarios (unknown job execution time) [30], it is easy to implement and requires little computational effort. As in the case of FCFS no knowledge of the job execution time is required. Application of backfilling will be of no benefit for this method.

SMART

The SMART algorithm has been introduced by Turek et al. [79]. The algorithm consists of 3 steps:

1. All jobs are assigned to bins based on their execution time. The upper bounds of those bins form a geometric sequence based on a parameter γ. In other words, the bins can be described by intervals of the possible execution time: $]0,1],]1,\gamma^1],]\gamma^1,\gamma^2], \ldots$. The parameter γ can be chosen to optimize the schedule.

2. All jobs in a bin are assigned to shelves (subschedules) such that all jobs in a shelf are started concurrently. To this end the jobs in a bin are ordered and then arranged in a shelf as long as sufficient resources are available.

3. The shelves are ordered using Smith's rule [69], that is for each shelf the sum of the weights of all jobs in the shelf is divided by the maximal execution time of any job in the shelf. Finally, those shelves with the largest ratio are scheduled first.

Schwiegelshohn et al. [60] have presented two variants of ordering the jobs in a bin and assigning them to shelves (Step 2):

SMART-$FFIA$

1. The jobs of a bin are sorted according to the product of execution time and the number of required nodes, also called *area*, such that the smallest area goes first.

2. The next job in this list is assigned to the first shelf with sufficient idle resources, that is, all shelves of this bin are considered.
3. If there is no such shelf, a new one is created and placed on top of the other shelves of this bin.

This approach is called the *First Fit Increasing Area* variant.

SMART-$NFIW$

1. All jobs of a bin are ordered by an increasing ratio of the number of required nodes to the weight of the job.
2. The next job in this list is added to the current shelf if sufficient resources are available on this shelf.
3. Otherwise a new shelf is created, placed on top of the current shelf and then becomes the current shelf itself.

This is the *Next Fit Increasing Width to Weight* variant.

The SMART algorithm has a constant worst case factor for weighted and unweighted response time scheduling. However, it is an off-line algorithm and cannot be directly applied to the scheduling problem of Example 5. It requires a priori knowledge of the execution time for all jobs and assumes that all jobs are available for scheduling at time 0. Therefore, the administrator modifies the SMART algorithm as follows:

1. He does not use the SMART algorithm to determine an actual schedule but to provide a job order for all jobs already submitted but not yet started. Whenever new jobs are submitted the SMART algorithm is started again. Based on this order a greedy list schedule is generated, see FCFS.

2. Instead of the actual execution time of a job the value provided by the user at job submission is used.

In order to reduce the number of recomputations for the SMART algorithm the schedule is recalculated only when the ratio between the already scheduled jobs in the wait queue to all the jobs in this queue exceeds a certain value. In the example a ratio of $\frac{2}{3}$ is used. The parameter γ is chosen to be 2.

As the final schedule is a list schedule the administrator decides to apply backfilling here as well.

PSRS

The PSRS algorithm [59] generates preemptive schedules. It is based on the modified Smith ratio of a parallel job, that is the ratio of job weight to the product of required resources and the execution time of the job. The basic steps of PSRS are described subsequently:

1. All jobs are ordered by their modified Smith ratio (largest ratio goes first).
2. A greedy list schedule is applied for all jobs requiring at most 50% of the machine nodes. If a job needs more than half of all nodes and has been waiting for some time, then all running jobs are preempted and the parallel job is executed. After the completion of the parallel job, the execution of the preempted jobs is resumed.

Similar to SMART, PSRS is also an off-line algorithm and requires knowledge of the execution time of the jobs. In addition it needs support for time sharing. Therefore, it cannot be applied to our target machine without modification.

The off-line problems can be addressed in the same fashion as for the SMART algorithm. Further, it is necessary to transfer the preemptive schedule into a non-preemptive one. To this end, it is beneficial that a job is not executed concurrently with any other job if it causes the preemption of other jobs.

1. First, 2 geometric sequences of time instances in the preemptive schedule are defined, one for those jobs causing preemption (wide jobs) and one for all other jobs (small jobs). In both cases the factor 2 is used with different offsets. These sequences define bins.
2. All jobs are assigned to those bins according to their completion time in the preemptive schedule. Within a bin the original Smith ratio order [69] is maintained.
3. A complete order of jobs is generated by alternatively picking bins from each sequence and starting with the small job sequence.

As with SMART the modified PSRS algorithm guarantees a constant approximation factor for the off-line case (with and without preemption).

2.6 Dependencies

The main dependence between the components of a scheduling system is easy to see: The scheduling policy produces rules which are used to derive an objective function. The application of this objective function to a schedule yields the schedule cost which allows performance measurements for the various algorithms. However, there are also additional dependencies. For instance, some policy rules may not allow efficient scheduling algorithms, see Example 4.

Example 4 *Assume a machine that does not support time sharing. The scheduling policy includes the rule:*

Every weekday at 10am the entire machine must be available to a theoretical chemistry class for 1 hour.

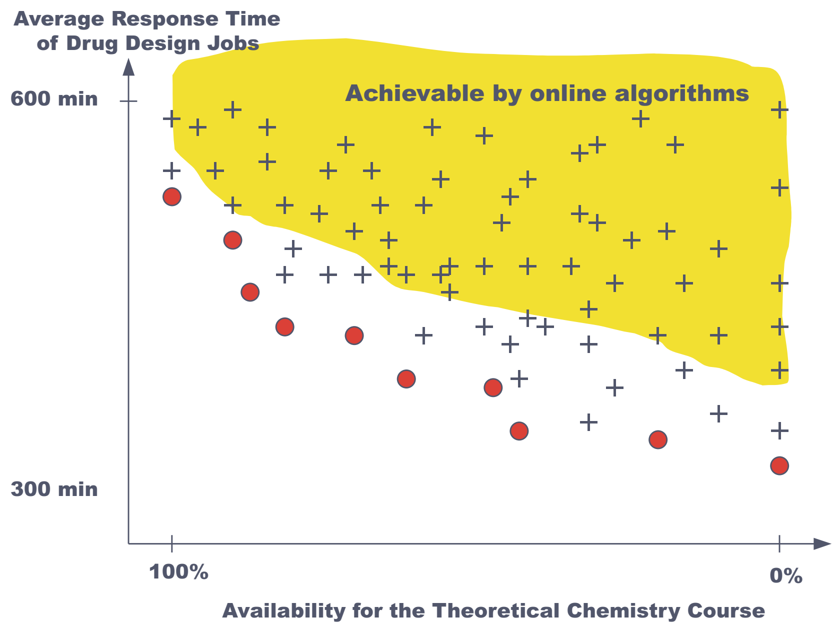

Figure 2.3: On-line versus Off-line Dependence

The Pareto-optimal schedules used for the determination of the objective function show an acceptable (by the owner) amount of idle resources before 10am. However, users are frequently unable to provide accurate execution time estimates for their jobs. Without knowledge on the execution time of jobs it is impossible for a scheduling algorithm to consistently generate good *schedules for different workloads. While, for instance, the first-come-first-serve strategy performs well for jobs without execution time estimates, it results in inefficient system utilization if many highly parallel jobs exist [78].*
Without the ability to preempt a running job (time sharing), an unexpected earlier job completion or a job failure can unnecessarily leave resources idle. The scheduler cannot foresee the actual run-time of jobs and has no ability to change previous scheduling decisions by preemption.

Figure 2.3 illustrates for Example 1 the different schedules if different job knowledge is available. There, it is shown that on-line algorithms can usually achieve a significantly smaller area of schedules in comparison to off-line methods with advance knowledge on the job submissions. Therefore, it may require a review of the conflict resolving strategy if results are compared from the different scheduling evaluation. Consequently, this can affect the considered schedule cost. Unfortunately, this on-line area of schedules will typically be the result of a combination of several on-line algorithms. Therefore, the off-line methods in the approach of Section 2.4 cannot be simply replaced by a single or a few on-line algorithms. The lack of knowledge on current and future jobs makes it usually impossible for on-line algorithms to generate the same schedules as off-line methods.

More of these additional dependencies are listed below:

- The owner may not know upfront his complete scheduling policy and all rules with their consequences. Aspects of the produced schedules may lead to changed rules and policies from the owner point of view. Too many or too restrictive policy rules may prevent schedules that are acceptable by owner and/or users at all. The user perspective and expectations are usually some part of the owner goal.
- Contrary to the case where many or too restrictive rules produce unacceptable results, there may not be sufficient rules to discriminate between *good* and *bad* schedules. Some implicitly assumed rules are frequently not explicitly stated by the owner.
- While there may be a variety of different objective functions which all support the policy rules, a specific objective function may not be suitable as a criterion for an on-line scheduling algorithm. This function may take effects into account that are unappropriate for the examined scenario. This can lead to unacceptable results for owner and/or user.
- The workload model may not be correct if users adapt their submission pattern due to their knowledge of the policy rules.
- The workload model must be modified as the number of users and/or the types and sizes of submitted jobs change over time.

Due to these dependencies a few design iterations may be required to determine the best suitable scheduling algorithms and/or it may be appropriate to repeat the design process occasionally. Nevertheless, it is not guaranteed that the owner succeeds in defining policy rules that leads to acceptable results.

2.7 Comparison

In this section we briefly compare the evaluation of scheduling systems with the well known procedure used for computer architectures. Today, computer architectures are typically evaluated with the help of standard benchmarks, like SPEC95 or Linpack, see [42]. For instance, the SPEC95 benchmark suite contains a variety of programs and frequently no architecture is the best for all those programs. Depending on his own applications the user must select the machine best suited for him. This leads to the question whether a similar approach is also applicable for scheduling systems. With other words, can we provide a few benchmark workloads which are used to test various scheduling systems?

We claim that this cannot be done at the moment and doubt whether this will ever become possible. For computer architectures there is a standard objective function: the execution time of a certain job. As we discussed in the previous sections each scheduling system has its own objective function. Therefore, we cannot really compare two different scheduling systems. On the other hand, the comparison of different scheduling algorithms only makes sense if the same objective function is used. Hence, the evaluation of scheduling algorithms must be based on benchmarks consisting of workloads and objective functions. However, it is not clear to us that there will ever be a small set of objective functions that will more or less cover all scheduling systems.

2.8 Evaluation Example

In this section we give an example for the design and evaluation process of scheduling algorithms. As the focus is on scheduling algorithms we will assume simple scheduling policy rules and only briefly cover the determination of the objective function.

During this example, we will use several typical algorithms as introduced in Section 2.5. We will later use these evaluation results for the comparison of theoretical and experimental evaluation Chapter 3 and Section 3.3.

Example 5 *Assume an Institution B that has just bought a large parallel computer with 288 identical nodes. The institution has established the following policy rules:*

1. *The batch partition of the computer must be as large as possible, leaving a few nodes for interactive jobs and for some services.*
2. *The user must provide the exact number of nodes for each job (rigid job model) and an upper limit for the execution time. If the execution of a job exceeds this upper limit, the job may be cancelled.*
3. *The user is charged for each job. This cost is based on a combination of projected and actual resource consumption.*
4. *Every user is allowed at most two batch jobs on the machine at any time.*
5. *Between 7am and 8pm on weekdays the response time for all jobs should be as small as possible.*
6. *Between 8pm and 7am on weekdays and all weekend or on holidays it is the goal to achieve a high system load.*

According to our model in Section 2.1.2, the machine supports variable partitioning [25] but does not allow time sharing. Further, it is required that all batch jobs have exclusive access to their partition.

The administrator decides that 256 nodes can be used for the batch partition. He further believes that the user community at the Cornell Theory Center (CTC) and at Institution B will be very similar. As the parallel machines at the CTC and at

Institution B are of the same type he decides to use a CTC workload as a basis for the selection of the objective function and the determination of a suitable scheduling algorithm. Due to the interdependence between user community and scheduling policy this decision also requires knowledge of the scheduling policy used at the CTC, see [44]. Only if there is no major disagreement between the scheduling policies at the CTC and at Institution B the profiles of both user communities can be assumed to remain similar.

2.8.1 Determination of the Objective Function

Next, the administrator must determine an objective function. To this end he ignores Rules 1 to 4 because they do not affect the schedule for a specific work load or are only relevant to the on-line situation (Rule 2). As Rules 5 and 6 do not apply at the same time he decides to consider each rule separately.

Rule 4 indicates that all jobs should be treated equally independent of their resource consumption. Therefore, the administrator uses the **average response time** as objective function for the daytime on weekdays (Rule 5). The average response time is the sum of the differences between the completion time and submission time for each job divided by the number of jobs.

For the remaining time (Rule 6) the **sum of the idle times** for all resources in a given time frame seems to be the best choice.

The administrator intends to independently determine an appropriate scheduling algorithm for each objective function and then to address the combination of both algorithms. Note that multi criteria optimization is therefore not necessary in our simple example.

When starting to look for scheduling algorithms the administrator realizes that the sum of idle times is based on a time frame. Therefore, it does not support on-line scheduling. Using the **makespan** instead has the advantage that several theoretical results are available, see e.g. [30], but again the makespan is mainly an off-line criterion [27]. Hence, he decides to use instead the **average weighted response time** where the weight is identical to the resource consumption of a job, that is, the product of the execution time and the number of required nodes, see [62]. It is calculated in the same fashion as the average response time with the exception that the difference between the completion and the submission time for each job is multiplied with the weight of this job. In comparison the job weight is always 1 for the average response time criterion. Note that for the average weighted response time the order of jobs does not matter if no resources are left idle [61]. We will discuss this weight selection later in Section 3.1.

2.8.2 Examined Algorithms

After the objective function has been determined it is necessary to find a suitable scheduling algorithm. Instead of producing an algorithm from scratch it is often more efficient to use algorithms from the literature and to modify them if necessary. In this first step it is frequently beneficial to consider a wide range of algorithms unless previous experiences strongly suggest the use of a specific type of algorithm. Further, there may be algorithms which have been designed for another objective function but can be adapted to the target function.

For Example 5 the administrator examines the algorithms from Section 2.5. On one hand, these are algorithms known from theory (Gary-Graham, Smart, PSRS) with known approximation costs as we will discuss in Chapter 3. On the other hand, typical algorithms (FCFS and Backfilling) are examined which are practically in use on real MPP systems.

The administrator decides to use both types of backfilling, conservative as well as EASY, as it is not obvious that one method is better than the other. In our example the administrator decides to apply backfilling to PSRS schedules as well.

2.8.3 Workload

As already mentioned in Section 2.8 the administrator wants to base his algorithmic evaluation on workload data from the CTC. In addition he decides to use two artificial workloads:

1. Artificial workload based on probability distributions,
2. Artificial workload based on randomization.

The number of jobs in each workload is given in Table 2.1. The reasons for this selection are discussed in the following subsections.

Workload	Number of jobs
CTC	79,164
Probability distribution	50,000
Randomized	50,000

Table 2.1: Number of jobs in various workloads

Workload Trace

In Section 2.8 the administrator has already verified that a CTC workload trace would be suitable in general. He obtains a workload trace from the CTC batch partition for the months July 1996 to May 1997. The workload is on-line available from the standard workload archive [74]. The trace contains the following data for each job:

- Number of nodes allocated to the job

- Upper limit for the execution time
- Time of job submission
- Time of job start
- Time of job completion
- Additional hardware requests of the job: amount of memory, type of node, access to mass storage, type of adapter.
- Additional job data like job name, LoadLeveler class, job type, and completion status.

Those additional job data are ignored as they are of no relevance to the simulation at this point. But the administrator must address two differences between the CTC machine and the parallel computer at his institution:

1. The CTC computer has a batch partition of 430 nodes while the batch partition at Institution B contains only 256 nodes.
2. The nodes of the CTC computer are not all identical. They differ in type and memory. This is not true for the machine at Institution B.

	Total number of jobs	Jobs requiring at most 256 nodes		Jobs requiring at most 128 nodes	
Jul 96	7953	7933	99.75%	7897	99.30%
Aug 96	7302	7279	99.69%	7234	99.07%
Sep 96	6188	6180	99.87%	6106	98.67%
Oct 96	7288	7277	99.85%	7270	99.75%
Nov 96	7849	7841	99.90%	7816	99.58%
Dec 96	7900	7893	99.91%	7888	99.85%
Jan 97	7544	7538	99.92%	7506	99.50%
Feb 97	8188	8177	99.87%	8159	99.65%
Mar 97	6945	6933	99.83%	6909	99.48%
Apr 97	6118	6102	99.74%	6085	99.46%
May 97	5992	5984	99.87%	5962	99.50%

Table 2.2: Number of Jobs in the CTC Workload Data for Each Month (Submission Time)

A closer look at the CTC workload trace in Table 2.2 reveals that less than 0.2% of all jobs require more than 256 nodes. Therefore, the administrator modifies the trace by simply deleting all those highly parallel jobs. Further, he determines that most nodes of the CTC batch partition are identical (382). Therefore, he decides to ignore all additional hardware requests.

Unfortunately, these modifications will affect the accuracy of the simulation. For instance, the simulation time frame of the whole modified CTC workload will most likely exceed the time span of the original trace as less resources are available. This will result in a larger job backlog during the simulation. Therefore, it is not possible to compare the original CTC schedule with the schedules generated by simulation. On the other hand, the administrator wants to separately test for two different objective functions, each of which will typically be valid for half a day. Hence, the present approach is only suited for a first evaluation of different algorithms. Any parametric fine tuning must be done with a better workload.

Besides using the CTC workload with the job submission data described above the administrator also wants to test his algorithms under the assumption that precise job execution times are available at job submission. This simulation allows him to determine the dependence of the various algorithms on the accuracy of the provided job execution times and the potential for improvement of the schedule. For this study the estimated execution times of the trace are simply replaced by the actual execution times.

Workload with Probability Distribution

As we mentioned above, there are some difficulties in applying an actual workload trace of another system. These traces may contain singular effects like for instance down-times. Additionally, necessary modifications of the traces to match the examined configurations may affect the accuracy of the evaluation. In order to overcome some of these difficulties the administrator decides to extract statistical data from the CTC workload trace. These data are then used to generate an artificial workload with the same distribution as the workload trace.

An analysis of the CTC workload trace yields that a Weibull distribution matches best the submission times of the jobs in the trace. It is difficult to find a suitable distribution for the other parameters. Therefore, bins are created for every possible requested resource number (between 1 and 256), various ranges of requested time and of actual execution length. Then probability values are calculated for each bin from the CTC trace. Randomized values are used and associated to the bins according to their probabilities. This generates a workload that is very similar to the CTC data set. Simulations for the CTC machine configurations are performed to analyze if the results are consistent to the original CTC sets. This can be compared for known characteristics like e.g. average weighted response time, backlog, machine utilization for the original CTC and the artificial workload. Once the administrator decided that the artificial workload does model the original CTC workload sufficiently for his purposes (for example by correlation analysis), the probabilistic parameters can be adapted to consider the various differences between the CTC and Institution B.

Randomized Workload

Finally, totally randomized data are used as a third input data set. The administrator is aware of the fact that this workload will not represent any real workload on his machine. But he wants to determine the performance of scheduling algorithms even in case of unusual job combinations. For the workload, jobs are generated with the parameters in Table 2.3 being equally distributed.

Submission of jobs	$\geq$ 1 job per hour
Requested number of nodes	1 – 256
Upper limit for the execution time	5 min – 24 h
Actual execution time	1 s – upper limit

Table 2.3: Parameters for randomized job generation

2.8.4 Evaluation Results

The administrator selects the simulation of FCFS with EASY backfilling to be a reference value as this algorithm is used by the CTC. First he compares the results for the CTC workload trace for the average response time, see Figure 2.4 and Table 2.5. In these figures the different strategies are compared relatively to the FCFS results with backfilling, which mark the 0% reference.

For the unweighted case he comes to the following conclusions:

- All algorithms are clearly better than FCFS even if some form of backfilling is used together with FCFS.
- PSRS and SMART can be improved significantly with backfilling.
- The classical list scheduling produces good results but is inferior to the PSRS and SMART with backfilling.
- Conservative backfilling outperforms EASY backfilling slightly when applied to PSRS and SMART schedules.
- There are little differences between PSRS and SMART schedules when backfilling is used.

The administrator does not give much weight to the absolute numbers as the workload trace has been recorded on a machine with 430 nodes while the simulations are done for a machine with 256 nodes. Although some highly parallel jobs have been removed from the trace a machine with 256 nodes will experience a larger backlog which results in a longer average response time.

Additionally, the weighted case is examined. To this end, the weight selection from Section 2.8.1 is applied, where the weight is identical to the resource consumption (execution time multiplied with the number of required nodes. In this weighted case as shown in Figure 2.5, the results are different:

- The classical list scheduling algorithms clearly outperforms all other algorithms.

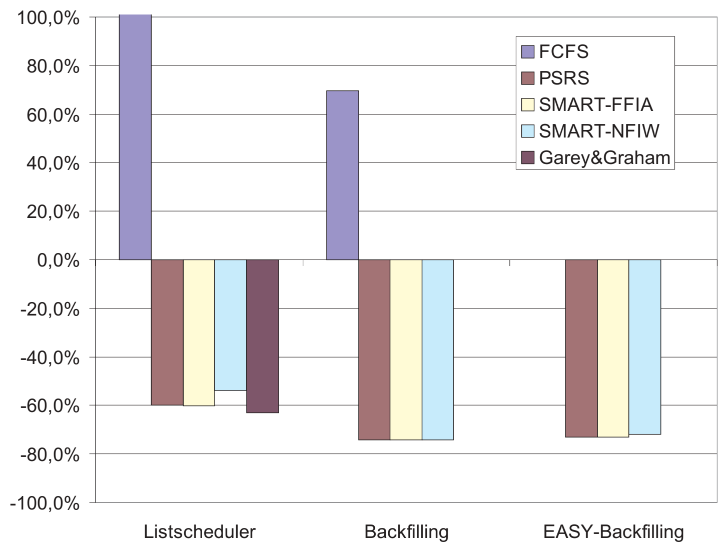

Figure 2.4: Relative Average Response Time for the CTC-Workload to the FCFS result with backfilling (0%)

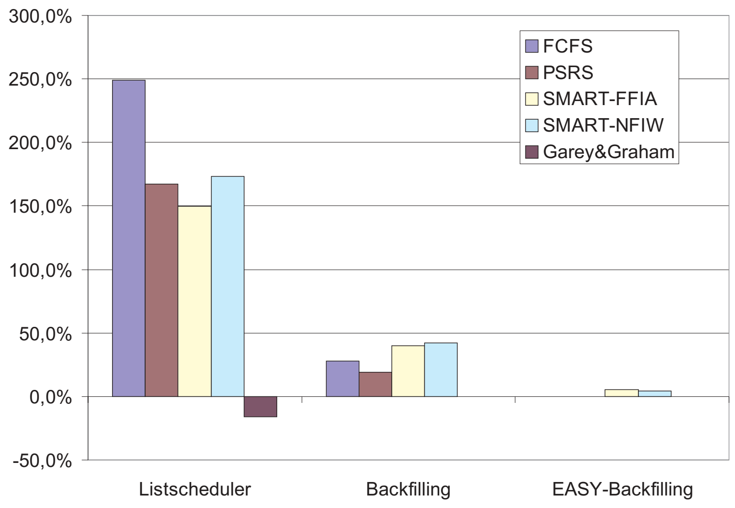

Figure 2.5: Relative Average Weighted Response Time for the CTC-Workload to the FCFS result with backfilling(0%)

		Listscheduler		Backfilling		EASY-Backfilling	
		sec	pct	sec	pct	sec	pct
Unweighted Case	FCFS	4.91E+06	0%	4.05E+05	-39.6%	3.93E+05	-0.5%
	PSRS	1.05E+05	-34.0%	6.35E+04	-37.7%	5.48E+04	-48.3%
	SMART-FFIA	9.07E+04	-42.2%	5.60E+04	-45.1%	5.33E+04	-49.7%
	SMART-NFIW	9.39E+04	-48.4%	5.66E+04	-44.5%	5.34E+04	-51.9%
	Garey&Graham	1.46E+05	0.0%				
Weighted Case	FCFS	4.99E+11	0%	1.14E+11	-37.7%	9.82E+10	-31.3%
	PSRS	3.91E+11	+2.4%	1.15E+11	-32.4%	9.91E+10	-30.7%
	SMART-FFIA	3.03E+11	-15.1%	2.73E+11	+36.5%	2.58E+11	+70.9%
	SMART-NFIW	3.33E+11	-14.8%	2.92E+11	+43.8%	2.68E+11	+79.9%
	Garey&Graham	1.20E+11	0.0%				

Table 2.4: Average Response Time for the CTC-Workload with Knowledge of the Exact Job Execution Time in Comparison to Results with Estimated Execution Times in Table 2.5

		Listscheduler		Backfilling		EASY-Backfilling	
		sec	pct	sec	pct	sec	pct
Unweighted Case	FCFS	4.91E+06	+1143.0%	6.70E+05	-69.6%	*3.95E+05*	*0%*
	PSRS	1.59E+05	-59.7%	1.02E+05	-74.2%	1.06E+05	-73.2%
	SMART-FFIA	1.57E+05	-60.2%	1.00E+05	-74.7%	1.17E+05	-70.4%
	SMART-NFIW	1.82E+05	-53.9%	1.02E+05	-74.2%	1.11E+05	-71.9%
	Garey&Graham	1.46E+05	-63.0%				
Weighted Case	FCFS	4.99E+11	+249.0%	1.83E+11	+28.0%	*1.43E+11*	*0%*
	PSRS	3.82E+11	+167.1%	1.70E+11	+18.9%	1.43E+11	0%
	SMART-FFIA	3.57E+11	+149.6%	2.00E+11	+39.9%	1.51E+11	+5.6%
	SMART-NFIW	3.91E+11	+173.4%	2.03E+11	+42.0%	1.49E+11	+4.2%
	Garey&Graham	1.20E+11	-16.1%				

Table 2.5: Average Response Time for the CTC-Workload with Estimated Execution Times

		Listscheduler		Backfilling		EASY-Backfilling	
		sec	pct	sec	pct	sec	pct
Unweighted Case	FCFS	6.17E+06	+499.0%	1.06E+06	+2.9%	*1.03E+06*	*0%*
	PSRS	2.86E+05	-72.2%	1.71E+05	-83.4%	1.55E+05	-85.0%
	SMART-FFIA	2.67E+05	-74.1%	1.74E+05	-83.1%	1.57E+05	-84.8%
	SMART-NFIW	2.85E+05	-72.3%	1.65E+05	-84.0%	1.64E+05	-84.1%
	Garey&Graham	2.78E+05	-73.0%				
Weighted Case	FCFS	6.17E+11	+108.4%	3.03E+11	+2.4%	*2.96E+11*	*0%*
	PSRS	5.10E+11	+72.3%	3.05E+11	+3.0%	2.91E+11	-1.7%
	SMART-FFIA	4.84E+11	+63.5%	3.33E+11	+12.5%	2.97E+11	+0.3%
	SMART-NFIW	4.86E+11	+64.2%	3.31E+11	+11.8%	3.03E+11	+2.4%
	Garey&Graham	2.72E+11	-8.1%				

Table 2.6: Average Response Time for the Probability Distributed Workload

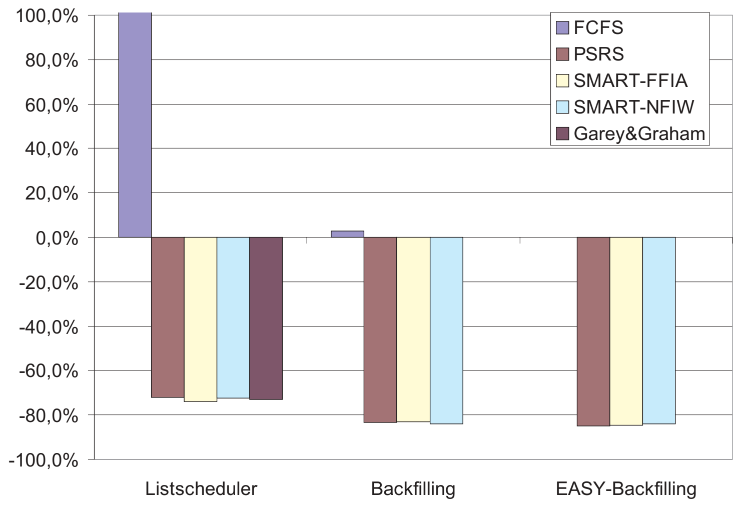

Figure 2.6: Relative Average Response Time for the Unweighted Probabilistic Workload in comparison to the FCFS result (0%)

- PSRS and SMART can be improved with either form of backfilling but are never better than FCFS with EASY.
- EASY is superior to conservative backfilling.
- PSRS is slightly better than either form of SMART.

The artificial workload based on probability distributions basically supports the results derived with the CTC workload, see Figure 2.6 and Table 2.6. This seems to indicate that the larger backlog in the CTC workload does not significantly affect the simulation results. However, it is strange that the absolute values for the average response time are even larger than in the CTC workload case although the number of jobs in the same time frame is significantly less. The only difference to the CTC workload is the fact that EASY is better than conservative backfilling if combined with PSRS or SMART in the unweighted case.

The derived qualitative relationship between the various algorithms is also supported by the randomized workload, see Table 2.7. Therefore, the administrator need not worry if a workload will occasionally deviate from his model.

Next the administrator addresses the simulation using the CTC workload with exact job execution times, see Figure 2.7 and Table 2.4. By comparing those results with the CTC workload simulations (Table 2.5) he wants to determine how much the accuracy of the job execution time estimation affects the schedules. This comparisons yields the following results:

- In the unweighted case the average response time of PSRS and SMART schedules can be improved by almost a factor of 2.

		Listscheduler		Backfilling		EASY-Backfilling	
		sec	pct	sec	pct	sec	pct
Unweighted Case	FCFS	3.40E+08	+96.5%	1.72E+08	-0.6%	*1.73E+08*	*0%*
	PSRS	1.66E+08	-4.0%	1.44E+08	-16.8%	1.32E+08	-23.7%
	SMART-FFIA	1.57E+08	-9.2%	1.41E+08	-18.5%	1.37E+08	-20.8%
	SMART-NFIW	1.61E+08	-6.9%	1.42E+08	-17.9%	1.39E+08	-19.7%
	Garey&Graham	1.73E+08	0%				
Weighted Case	FCFS	9.40E+14	+41.6%	6.66E+14	+0.3%	*6.64E+14*	*0%*
	PSRS	8.66E+14	+30.4%	6.61E+14	-0.5%	6.60E+14	-0.6%
	SMART-FFIA	8.15E+14	+22.7%	7.54E+14	+13.6%	6.96E+14	+4.8%
	SMART-NFIW	9.05E+14	+36.3%	7.96E+14	+19.9%	7.09E+14	+6.8%
	Garey&Graham	6.68E+14	+0.6%				

Table 2.7: Average Response Time for the Randomized Workload

		Listscheduler	EASY-Backfilling
		pct	pct
Unweighted Case	FCFS	-81.6%	*0%*
	PSRS	-76.7%	-33.7%
	SMART	-75.6%	-32.7%
	Garey&Graham	-58.4%	
Weighted Case	FCFS	-80.6%	*0%*
	PSRS	+30.6%	-39.4%
	SMART	-13.7%	-34.3%
	Garey&Graham	-57.2%	

Table 2.8: Computation Time for the CTC Workload

		Listscheduler	EASY-Backfilling
		pct	pct
Unweighted Case	FCFS	-92.1%	*0%*
	PSRS	-88.5%	-79.6%
	SMART	-87.1%	-80.1%
	Garey&Graham	-72.3%	
Weighted Case	FCFS	-91.6%	*0%*
	PSRS	-27.2%	-57.4%
	SMART	-50.5%	-72.7%
	Garey&Graham	-69.2%	

Table 2.9: Computation Time for the Probability Distributed Workload

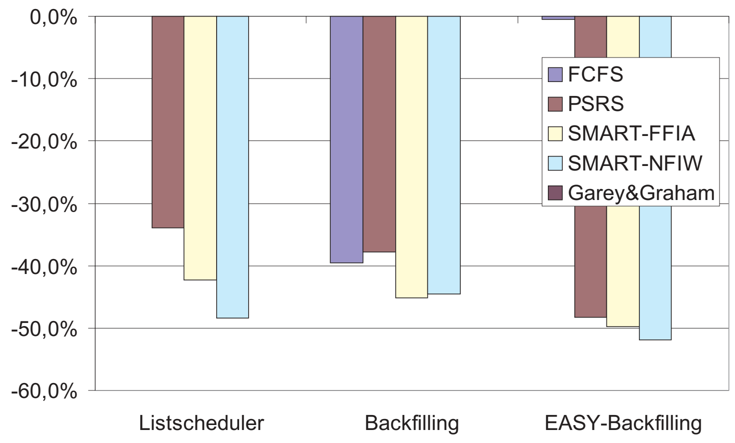

Figure 2.7: Comparison of the Average Response Time for Exact vs. Estimated Job Execution Length

- In the weighted case both forms of backfilling achieve better results than the classical list scheduling if applied to FCFS or PSRS schedules.
- Surprisingly, SMART schedules with backfilling give worse results in the weighted case for the CTC workload using the estimated job execution time than for the original submission data.

Finally, the administrator considers the computation time to execute the various algorithms for the CTC workload (Table 2.8) and the artificial workload based on probability distributions (Table 2.9). In both cases similar results are obtained with a few observations being noteworthy:

- It is surprising that the classical list scheduling algorithm requires a similar computation time for both workloads while the larger number of jobs in the CTC workload results in more computational effort in almost all other cases.
- In the unweighted case SMART and PSRS together with EASY require approximately the same computation time which is significantly less than needed by FCFS and EASY.
- In the weighted case PSRS and SMART need a significant amount of computation time.

As a conclusion, the administrator decides to use the classical list scheduling algorithm for the weighted case. In the unweighted case the results are not that clear. He intends to use either SMART or PSRS together with some form of backfilling. However, he wants to execute more simulations to fine tune the parameters of those algorithms before making the final decision. In addition he must evaluate the effect of combining the selected algorithms. This concludes the evaluation example.

Note that there may be plenty of reasons to consider other algorithms or to modify the simulation model. This section showed practical results for the assumed scenario and allowed the discussion on the examined scheduling algorithms.

2.9 Summary

In this chapter we presented a strategy to design a scheduling system for parallel processors. This strategy was illustrated with the help of an example that addressed the following items in particular:

1. Determination of an objective function from a given simple set of policy rules
2. Selection of a several scheduling algorithms from the literature
3. Modification of the selected algorithms where necessary
4. Evaluation of the algorithms with the help of real and artificial workloads

A main task in designing a scheduling system is the evaluation of a scheduling algorithm. In our design of scheduling algorithms for parallel computers and consequently for grid system we want to address this topic in more detail in the following parts of this work. To this end, we first investigate further into theoretical evaluation of scheduling methods. The theoretical approach will give us information on the worst-case behavior of the algorithms. However, the actual workload and the user demand is not taken into account. As we are especially interested in the practical performance of the scheduling system in a real workload scenario, we later look closer on simulation results for designing scheduling methods in a grid environment.

Part II

Theoretical Evaluation

Chapter 3

Overview on Theoretical Evaluation

During the process of designing a scheduling system in the previous chapter the evaluation of potential algorithm played an important role. As we mentioned before, there are generally two approaches on the evaluation:

- Theoretical evaluation and
- experimental evaluation.

In this part, we examine the theoretical evaluation more closely. Based on observations of previous results for FCFS , we present a new algorithm called PFCFS and its theoretical analysis. We will discuss this approach at the end of this chapter. From these results we make the transition to experimental analysis in the next chapter.

3.1 Performance Metrics

As seen in Section 2.8 different objectives are possible for designing an algorithm and evaluating a schedule. Several common performance metrics are frequently used in theoretical analysis. We will briefly introduce some of them as they are used in the following evaluations.

Makespan This value is defined by the completion time of the last job of a given job set τ, which is $C_{max,S} = \max_{i \in \tau} t_i(S)$.
The makespan is mainly an off-line criterion. It is closely related to the utilization throughput and represents a metric which is often preferred by system owners [26].

Average Weighted Completion/Response Times The average completion time (ACT) is defined by the sum of all completion times divided by the number of jobs. By use of the weight w_i for each job i some jobs may indirectly get a higher priority than others. The sum of the weighted completion times is therefore defined as: $C_S = \sum_{i \in \tau} w_i t_i(S)$. The average weighted response time (AWRT) closely relates to the delay between submission and completion time:

$$\text{AWRT} = \frac{\sum_{i \in \tau} (t_i - r_i) \cdot w_i}{\sum_{i \in \tau} w_i}$$

The response or flow time represents a metric from the user's point of view, as he is most often interested in minimizing the delay between job submission and job completion. However, it is much harder to approximate the optimal weighted flow time than the optimal weighted completion time, as shown by Leonardi and Raz [50]. Therefore, many researchers focused on the completion time problem. This may be justified as both measures differ from each other only by a constant. Therefore, it is sufficient to consider just one of both criteria for the purpose of comparing two schedules.
Note that the completion time is of very limited interest in a practical setting where individual jobs are not submitted at the same time.

3.2 Background

First, we start with a brief overview on the theoretical analysis of scheduling in job scheduling.

Unfortunately, most scheduling problems are NP-complete in the strong sense. This is even true for off-line problems with simple objective functions and few additional requirements, see for instance [37]. As mentioned previously, in the theoretical scheduling community the criteria most frequently used are either the makespan or either the sum of the completion times or flow times of all jobs. Even simple parallel scheduling problems based on these criteria are NP-complete, see Bruno, Coffman, and Sethi [5] or Du and Leung [13].

Previous work in the area of makespan scheduling of parallel jobs includes off-line scheduling [36], scheduling of moldable (malleable) jobs [80] or scheduling with unknown job execution times [30] as well as release date scheduling [85].

Similarly, completion time scheduling algorithms with a constant approximation factor have been provided with respect to off-line scheduling [60], randomized scheduling [8], preemptive scheduling [59], scheduling of moldable (malleable) jobs [78] and scheduling with unknown job execution times [11].

3.2.1 Optimal Solutions

For very simple scheduling problems it is possible to calculate the optimal solution. This is often done by the enumeration of solutions and by some variations of branch-and-bound strategies. For parallel job scheduling the optimal solution is usually not tractable in a moderate amount of time.

Example 6 *A simple branch-and-bound approach for a scheduling problem with 14 jobs on a parallel computer takes several hours to complete on a workstation (depending on the hardware). Computationally hard branch and bound problems have typically exponentially time complexity for increasing problem sets with more variables. In our case, for a larger job set the amount of time necessary to find the optimal solution is not acceptable. This applies for the offline as well as for the online problem.*
In the online case, there are typically a number of jobs in the backlog queue. Additionally, the process has to be applied recurrently, as new jobs arrive and running jobs end ahead of the estimated execution time.

3.2.2 Worst-Case Analysis with Approximation Factors

As optimal solutions are difficult or sometimes impossible to obtain, efforts to deal with these scheduling problems have mainly been devoted to find polynomial time algorithms that have small approximation factors. This factor can be a worst-case bound for an algorithm in comparison to the optimal solution or in comparison to another algorithm. The theoretical analysis may provide an approximation factor for these algorithms. This factor denotes the quality of the solution relative to an optimal solution. In competitive analysis the result of an algorithm is compared to the result of another algorithm for a given input data. In this case the analysis provide a ratio for the criteria between both algorithm.

Makespan and completion time scheduling have been subject to various research efforts:

Makespan Scheduling For the makespan scheduling problem, for instance, Garey and Graham [36] have shown that an arbitrary list schedule on m identical machines for a given job system achieves a tight approximation factor of $2 - \frac{1}{m}$. That means that a list scheduler, as e.g. the algorithm from Section 2.5, provides results for the makespan problem that is less than twice as long as the optimal solution.

Completion Time Scheduling For sequential jobs Kawaguchi and Kyan [47] presented a list scheduling algorithm called LRF with a tight approximation factor of $\frac{1+\sqrt{2}}{2}$. Additionally, McNaugthon proved that every such schedule could not further be improved by preemption [54].

The problem for parallel jobs is more difficult than its sequential counterpart. Turek et al. showed that the SMART algorithm generates shelf based schedules with an approximation factor of 10.45 [79] which has subsequently been reduced to 8.53 [60].

Turek et al. [78] showed that a generalization of Kyan and Kawaguchi's LRF method produces a tight approximation factor of 2 for parallel jobs with unique weights if the resource requirement of each job is at most 50% of the maximum number of processors. If arbitrary jobs are allowed, this method may result in schedules which deviate significantly from the optimum.

While preemption does not improve the results for the sequential scheduling problem, this does not hold for the parallel case. Deng, Gu, Brecht, and Lu [11] also discuss response time scheduling with preemption for jobs with unique weights and variable resource requirements. Schwiegelshohn [59] gave an approximation factor of 2.42 for the preemptive PSRS algorithm from Section 2.5. In the non-preemptive case, PSRS delivers an approximation factor of 7.11 for the completion time.

However, the mentioned works focus on off-line scheduling. Real systems for parallel job scheduling, especially in metacomputing and grid scenarios, have different release times for the submitted jobs. There is also no knowledge on future job submissions. In addition, some provided job properties may not be immediately available or may be incorrect which makes the task for the algorithm even harder, see Section 2. Consequently, the on-line scenario has to be considered.

It must be emphasized that most results in the theoretic research usually address either the makespan or the weighted completion time criterion which is not satisfying. While an optimal makespan will not guarantee a minimal or even small sum of weighted completion times and vice versa [59], Stein and Wein [70] as well as Chakrabarti et al. [8] and Schwiegelshohn [59] showed that there are methods which provide good performance for both criteria. Moreover, all the weighted completion time algorithms mentioned above fit in this category.

In real installations the weighted completion or flow time criterion is almost never used to represent user satisfaction. A *first-come-first-serve* strategy is frequently applied in order to guarantee some kind of *fair* treatment for all users [44]. Unfortunately, this strategy has a very bad worst case behavior, see Section 2.5. We want to address the aspect of fairness related to this scheduling problem in more detail in the following section. In our example for the design and evaluation process in Section 2.8.2 we used several of the here mentioned algorithms. We will refer to these results in our further discussion on the theoretical analysis.

3.3 Fairness in Parallel Job Scheduling

As we saw, theoretic scheduling algorithms mostly focus on off-line scheduling and do not consider both objectives, owner's and user's satisfaction. Neglecting the weighted completion or response time can cause very bad response time behavior for user jobs which is usually not tolerable. This is a reason why several of the mentioned algorithms are not implemented in real installations. In the following we want to introduce a preemptive scheduling algorithm which is based on the concept of fairness in job scheduling [48, 65]. To this end, we first introduce a special selection of job weights.

The weight selection results from the assumption that there should be no gain in splitting a parallel job into many sequential jobs. The same is true for combining many independent short jobs into a single long job. Only the overhead of a parallel job in comparison to its sequential version is considered. Based on the results of Smith [69] this leads to the following weight selection:

Assumption 1 *The weight w_i of a job i is the product of the number of nodes assigned to the job, that is $m_i = |\mathcal{M}_i|$, and the execution time p_i of the job.*

The weight w_i in Assumption 1 may also represent the cost of job i as it equals the total resource consumption of the job. Note that w_i may not be available at the release date of i if the execution time p_i is not known at this moment.

As already mentioned, fairness is further observed in many commercial schedulers by using the *First-come first-serve* principle. But starting jobs in this order will only guarantee fairness in a non-preemptive schedule. We now consider a parallel computer that supports preemption. As mentioned before, preemption allows to temporarily halt a job on its resource set. The job can later be resumed. The ability to preempt parallel jobs simultaneously on all processors in the job's partition is often referred to as gang-scheduling [25]. This context switch is executed by use of the local memory and does not affect the interconnection network. It will cause a preemption penalty which in reality is mainly due to processor synchronization, storing of undelivered messages, saving of job status and page faults [83]. In our model we describe this preemption penalty by a constant time delay s. During this time delay no node of an affected partition is able to execute any part of a job. Note that the context switch does not include job migration, which would change the node subset assigned to a job during its execution.

In a preemptive schedule a job may be interrupted by another job which has been submitted later. In the worst case this may result in job starvation, that is the delay of the job completion for a long period of time. Therefore, we introduce the following parameterized definition of fairness:

Definition 1 *A scheduling strategy is λ-fair if all jobs submitted after a job i cannot increase the flow time of i by more than a factor λ.*

It is therefore the goal to find a method which produces schedules with small values for $\frac{C_{max,S}}{C^*_{max}}$, $\frac{C_S}{C^*}$, and λ where C^*_{max} and C^* denote the optimal makespan and the optimal sum of weighted completion times, respectively.

3.3.1 The Algorithm

In the following, we introduce a new preemptive algorithm called PFCFS that is well suited for fair on-line scheduling of parallel jobs. Fairness is achieved by using the selection of job weights as mentioned above and by limiting the time span a job can be delayed by other jobs submitted after it. Further, we do not require that the processing time of a job is known when the job is released.

At first we consider a simple non-preemptive list scheduling algorithm where the start order of all jobs is determined by their submission times. Although this algorithm is actually used in commercial schedulers, it may produce bad results as can be seen by the example below.

Example 7 *Simply assume m jobs with $p_i = m$, $m_i = 1$ and m jobs with $p_i = 1$, $m_i = m$. If jobs are submitted in quick succession from both groups alternatively, then $C_{max,S} = O(m^2)$ while $C^*_{max} = O(m)$.*

Based on the given scheduling strategy and our job model we are looking for an algorithm producing a λ-fair schedule S with $C_{max,S} \leq \mu \cdot C^*_{max}$ for small values λ and μ. As pointed out by Shmoys et al. [66] the method of Feldmann et al. [30] can be used to achieve $\mu = 4$. But no constant bound can be given for λ. In addition Shmoys et al. showed that no deterministic on-line algorithm has a better competitive ratio than $2 - \frac{1}{m}$ even if preemption is allowed and all jobs are sequential.

On the other hand no gain can be expected from trying to achieve $\lambda = 1$.

Lemma 1 *Any on-line algorithm producing 1-fair schedules S cannot guarantee $C_{max,S} \leq O(\sqrt{m}) \cdot C^*_{max}$.*

Proof *Assume two jobs i and j being submitted in quick succession with $p_i = m$, $m_i = 1$ and $p_j = \sqrt{m}$, $m_j = m$. Obviously both jobs cannot be executed concurrently. Assuming $w_i = p_i \cdot m_i$ and $w_j = p_j \cdot m_j$ we obtain the following results for both orders:*

order	*sum of weighted completion times*	*λ-fairness*
(i,j)	$O(m^{2.5})$	$\lambda = 1$
(j,i)	$O(m^2)$	$\lambda = 1 + \frac{1}{\sqrt{m}}$

Note that this result cannot be improved by introducing preemption.

To solve this problem we introduce preemption and accept $\lambda > 1$. A suitable algorithm for this purpose may be PSRS (Preemptive Smith Ratio Scheduling) [59] which is based on a list and uses gang scheduling. The list order in PSRS is determined by the ratio $\frac{w_i}{m_i p_i}$. As this ratio is the same for all jobs in our scheduling problem (see Assumption 1) any order can be used and PSRS is able to incorporate FCFS (First Come First Serve Scheduling). An adaption of PSRS to our scheduling problem is called PFCFS (Preemptive FCFS) and given in Table 3.1.

Intuitively, a schedule produced by Algorithm PFCFS can be described as the interleaving of two non-preemptive FIFO schedules where one schedule contains at most one (wide) job at any time instant. Note that only a wide job ($m_i > \frac{m}{2}$) can cause preemption and therefore increase the completion time of a previously submitted job. Further, all jobs are started in FIFO order.

Instruction A is responsible for the on-line character of Algorithm PFCFS. We call any time period Δ a period of available resources (PAR), if during the whole period Instruction A is executed and at least $\frac{m}{2}$ resources are always idle. Note that any execution of Algorithm PFCFS will end with a PAR.

```
        while (the parallel computer is active) {
            if ($Q = \emptyset$ and no new jobs have been submitted)
$A$             wait for the next job to be submitted;
            attach all newly submitted jobs to $Q$ in FIFO order;
            Pick the first job $i$ of $Q$ and delete it from $Q$;
            if ($m_i \leq \frac{m}{2}$) {
$B$             wait until $m_i$ resources are available;
                start $i$ immediately;}
            else {
$C$             wait until $\frac{m}{2}$ are available;
$D$             If (job $i$ has not been started)
$E_1$               wait until $m_i$ resources are available or time period $t$ has passed;
                else
$E_2$               wait until the previously used subset of $m_i$ resources is available
                        or time period $t$ has passed;
                If (the required $m_i$ resources are available)
$F$                 start or resume execution of $i$ immediately;
                else {
$G$                 preempt all currently running jobs;
$H$                 start or resume execution of $i$ immediately;
$I$                 wait until $i$ has completed or time period $t$ has passed;
                    If (the execution of $i$ has not been completed)
$J$                     preempt job $i$;
$K$                 resume the execution of all previously preempted jobs;
                    If (the execution of $i$ has not been completed)
                        Goto $D$; }}}
```

Table 3.1: The Scheduling Algorithm PFCFS

Further, we assume that the modification of Q or the selection of resources does not require any time. Therefore, any time instant in a PFCFS schedule corresponds with Instructions A, B, C, E_1, E_2, F, G, H, I, J or K. This is easy to see for any instructions that contain the word *wait*. In addition we assume that preempting a running job or resuming the execution of a preempted job results in a preemption penalty. Therefore, some time passes during the execution of Instructions G, J and K. The same holds for Instructions F and H if the execution of job i is resumed.

3.3.2 Theoretical Analysis

Next, we theoretically examine the proposed PFCFS algorithm. The competitive analysis allows us to give bounds for schedules produced by the PFCFS algorithm.

First, we describe a few specific properties of schedules where $w_i = m_i p_i$ holds for all jobs i. Note that these properties can also be derived from more general statements of other publications.

Corollary 1 *Let τ be a sequential job system such that $w_i = p_i$ holds for all jobs $i \in \tau$. Assume a non-preemptive schedule S where each job i is started time $(k-1)p_i$ after the completion of the previous job (or after time 0 if i is the first job). Then the completion time cost C_S of S is independent of the order of the jobs in S and there is*

$$C_S = kC^* = \frac{k}{2}((\sum_{i\in\tau} p_i)^2 + \sum_{i\in\tau} p_i^2).$$

Proof *Note that each job i is completed time kp_i after its immediate predecessor (or after time 0 if i is the first job in S) in schedule S and that $\frac{w_i}{kp_i} = \frac{1}{k}$ holds for all jobs. Therefore, Smith's rule [69] guarantees that the job order of schedule S does not affect the completion time cost. Moreover, any schedule S with $k = 1$ is optimal.*

We have $C_{max,S} = k\sum_{i\in\tau} p_i$ for $|\tau| = 1$. Therefore, the bound is clearly true for $|\tau| = 1$. Next assume that the statements for C_S and $C_{max,S}$ hold for all job systems τ with $|\tau| = n-1$ and any schedule S for τ with the described structure. Adding a new job i_n to τ and starting this job time $(k-1)p_{i_n}$ after the completion of the last job in such a schedule S produces a job system τ' with $|\tau'| = n$ and a schedule S' with

$$\begin{aligned} C_{max,S'} &= C_{max,S} + kp_{i_n} \\ &= k\sum_{i\in\tau} p_i + kp_{i_n} \\ &= k\sum_{i\in\tau'} p_i \qquad \textit{and} \end{aligned}$$

$$\begin{aligned}
C_{S'} &= C_S + kw_{i_n}p_{i_n} + w_{i_n}C_{max,S} \\
&= C_S + kp_{i_n}^2 + p_{i_n}C_{max,S} \\
&= \frac{k}{2}((\sum_{i\in\tau} p_i)^2 + \sum_{i\in\tau} p_i^2) + kp_{i_n}^2 + kp_{i_n}\sum_{i\in\tau} p_i \\
&= \frac{k}{2}((\sum_{i\in\tau'} p_i)^2 + \sum_{i\in\tau'} p_i^2).
\end{aligned}$$

Corollary 2 *Assume that $w_i = m_ip_i$ holds for all jobs i in a job system τ. A new job system τ' and a new schedule S' are generated from any non-preemptive schedule S for τ by replacing a single job $i \in \tau$ with the successive execution of two jobs i_1 and i_2 such that $m_{i_1} = m_{i_2} = m_i$, $p_i = p_{i_1} + p_{i_2}$, $w_{i_1} = m_ip_{i_1}$ and $w_{i_2} = m_ip_{i_2}$ hold. This results in $C_S - C_{S'} = p_{i_1}p_{i_2}m_i \leq C^*(\tau) - C^*(\tau')$.*

Proof *Note that the transformation always produces a legal schedule S'. The splitting of job i has no effect on the completion time of any other job in τ. Further, $w_{i_1} + w_{i_2} = w_i$, $t_i(S) = t_{i_2}(S')$, and $t_{i_1}(S') = t_{i_2}(S') - p_{i_2}$ hold. Therefore, we have*

$$\begin{aligned}
C_S - C_{S'} &= w_it_i(S) - w_{i_1}t_{i_1}(S') - w_{i_2}t_{i_2}(S') \\
&= (w_{i_1} + w_{i_2})t_i(S) - w_{i_1}(t_{i_2}(S') - p_{i_2}) - w_{i_2}t_{i_2}(S') \\
&= w_{i_1}p_{i_2} = m_ip_{i_1}p_{i_2}.
\end{aligned}$$

As this kind of job splitting also transforms an optimal non-preemptive schedule for τ into a legal but not necessarily optimal schedule, $C^*(\tau) - C^*(\tau') \geq m_ip_{i_1}p_{i_2} = C_S - C_{S'}$ holds as well.

The proof described above also holds in the preemptive case, if the second job i_2 is not preempted in schedule S' and starts immediately after the completion of job i_1. For a more general statement see Corollary 3.

Note, that Corollary 2 leads also to the following relation :

$$\begin{aligned}
\frac{C_S}{C^*(\tau)} &= \frac{C_{S'} + m_ip_{i_1}p_{i_2}}{C^*(\tau)} \\
&\leq \frac{C_{S'} + m_ip_{i_1}p_{i_2}}{C^*(\tau') + m_ip_{i_1}p_{i_2}} \leq \frac{C_{S'}}{C^*(\tau')}
\end{aligned}$$

It is further possible to split a parallel job vertically, that is producing a schedule S' from a schedule S by replacing a job i in S with m_i identical jobs i' such that $m_{i'} = 1$, $p_{i'} = p_i$, and $w_{i'} = \frac{w_i}{m_i}$. Then, we have $C_{S'} = C_S$ and $C^*(\tau) \geq C^*(\tau')$.

Therefore the following relation holds for vertical as well as for horizontal splitting as described above:

$$\frac{C_S}{C^*(\tau)} \leq \frac{C_{S'}}{C^*(\tau')}$$

As already mentioned in the previous section, PFCFS will generate non-preemptive FCFS schedules, if all jobs are small, that is, they require at most 50% of the maximum amount of resources ($m_i \leq \frac{m}{2}$). First, we restrict ourselves to this case and prove some bounds. In Lemma 2 we consider scenarios with a single PAR.

Lemma 2 *Let $m_i \leq \frac{m}{2}$ and $w_i = m_i \cdot p_i$ for all jobs of a job system τ. Also assume that there is no PAR before the submission of the last job. Then Algorithm PFCFS is 1-fair and will only produce non-preemptive schedules S with the properties*

1. *$C_S < 2 \cdot C^*$ and*
2. *$C_{max,S} < 3 \cdot C^*_{max}$.*

Proof *As the whole* else *block starting with Instruction C cannot be executed, Algorithm PFCFS is identical to the non-preemptive FCFS algorithm which is 1-fair.*

Next, we address the bound for the completion times. Note that for the calculation of $C^*(\tau)$ the submission times of all jobs are simply ignored. This cannot increase $C^*(\tau)$.

Now we describe a class of job systems τ' and schedules S' with the following properties:

1. *All jobs $i \in \tau'$ are sequential, that is $m_i = 1$.*
2. *x is the last starting time of any job in τ'.*
3. *τ'_1 is the set of all jobs $i \in \tau'$ that complete no later than time x ($t_i(S') \leq x$).*
4. *All jobs $i \in \tau' \backslash \tau'_1$ start at time x in schedule S'.*
5. *Exactly $\frac{m}{2}$ resources are used at any time instant between 0 and x in schedule S'.*

Note that $\tau' \backslash \tau'_1$ contains at least 1 and not more than m sequential jobs. Further, $\tau' \backslash \tau'_1$ is partitioned into two disjoint sets τ'_2 and τ'_3 which are implicitly defined by use of the following equation where the parameter y is introduced for reasons of convenience:

$$\tau'_3 = \{i \in \tau' \backslash \tau'_1 | p_i > y := \frac{\sum_{j \in \tau' \backslash \tau'_3} p_j}{m - |\tau'_3|} = \frac{x\frac{m}{2} + \sum_{j \in \tau'_2} p_j}{m - |\tau'_3|}\}.$$

Intuitively, $y \cdot (m - |\tau'_3|)$ is the total resource consumption of all jobs in $\tau' \backslash \tau'_3$, see Figure 3.1. Of course, τ'_3 may be empty.

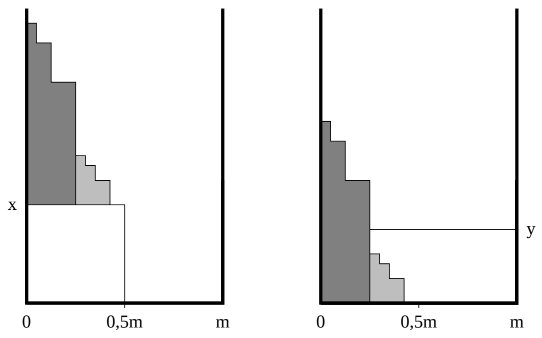

Figure 3.1: Schedule S' (left) and the optimal schedule for τ' (right)

Using these definitions and Corollary 1 we can calculate $C_{S'}$:

$$\begin{aligned} C_{S'} &= C_{S'}(\tau_1') + C_{S'}(\tau_2') + C_{S'}(\tau_3') \\ &= \frac{x^2 m}{4} + \frac{1}{2}\sum_{i\in\tau_1'} p_i^2 + \sum_{i\in\tau_2'} p_i(p_i + x) + \sum_{i\in\tau_3'} p_i(p_i + x) \end{aligned}$$

To determine the optimal schedule $C^*(\tau')$ for τ' on m resources we again use Corollary 1 and obtain

$$\begin{aligned} C^*(\tau') &\geq \frac{y^2(m - |\tau_3'|)}{2} + \frac{1}{2}\sum_{i\in\tau_1'} p_i^2 + \frac{1}{2}\sum_{i\in\tau_2'} p_i^2 + \sum_{i\in\tau_3'} p_i^2 \\ &= \frac{x^2 m^2}{8(m - |\tau_3'|)} + \frac{(\sum_{\tau_2'} p_i)^2}{2(m - |\tau_3'|)} + \frac{\sum_{\tau_2'} p_i x m}{2(m - |\tau_3'|)} + \frac{1}{2}\sum_{i\in\tau_1'} p_i^2 + \frac{1}{2}\sum_{i\in\tau_2'} p_i^2 + \sum_{i\in\tau_3'} p_i^2. \end{aligned}$$

Finally, we compare the costs of the schedule S' and the optimal schedule. This yields

$$\begin{aligned} 2C^*(\tau') - C_{S'} &\geq \frac{x^2 m^2}{4(m - |\tau_3'|)} + \frac{(\sum_{\tau_2'} p_i)^2}{m - |\tau_3'|} + \frac{\sum_{\tau_2'} p_i x m}{m - |\tau_3'|} + \frac{1}{2}\sum_{i\in\tau_1'} p_i^2 + \\ &\quad \sum_{i\in\tau_3'} p_i^2 - \frac{x^2 m}{4} - \sum_{i\in\tau_2'} p_i x - \sum_{i\in\tau_3'} p_i x \end{aligned}$$

$$\begin{aligned}
&> \frac{x^2m}{4}\left(\frac{m}{m-|\tau_3'|}-1\right)+\sum_{i\in\tau_2'}p_ix\left(\frac{m}{m-|\tau_3'|}-1\right)+\sum_{i\in\tau_3'}p_i^2-\sum_{i\in\tau_3'}p_ix \\
&\geq \frac{x^2m}{4}\frac{|\tau_3'|}{m-|\tau_3'|}+\sum_{i\in\tau_3'}p_i^2-\sum_{i\in\tau_3'}p_ix \\
&\geq |\tau_3'|\frac{x^2}{4}+|\tau_3'|\left(\frac{\sum_{i\in\tau_3'}p_i}{|\tau_3'|}\right)^2-\sum_{i\in\tau_3'}p_ix=|\tau_3'|\left(\frac{x}{2}-\frac{\sum_{i\in\tau_3'}p_i}{|\tau_3'|}\right)^2\geq 0.
\end{aligned}$$

Note that for $\tau_3'=\emptyset$ the term $(\frac{\sum_{i\in\tau_3'}p_i}{|\tau_3'|})^2$ disappears. Now, we transform job system τ and the PFCFS schedule S into such a job system τ' and a schedule S' by repeatedly applying

- *vertical splitting,*
- *horizontal splitting as addressed in Corollary 2,*
- *and by moving all jobs which violate the resource constraints to the end of the schedule.*

Remember that no job requires more than $\frac{m}{2}$ resources and that there is no PAR before the submission of the last job. The moving of jobs will increase the cost of a schedule while $C^*(\tau)$ remains unchanged. Therefore, we have

$$\frac{C_S}{C^*(\tau)}\leq\frac{C_{S'}}{C^*(\tau')}.$$

This concludes the proof for the completion time bound. The same bound for $\frac{C_S}{C^*}$ can also be derived by a generalization of Lemma 4.1 in a paper by Turek et al. [78].

For the makespan first note that $C_{max,S'}\geq C_{max,S}$ and $C^*_{max}(\tau')\leq C^*_{max}(\tau)$ hold with respect to all transformations described above. Also, the optimal makespan is bound by $C^*_{max}(\tau')\geq\max_{i\in\tau'}\{p_i\}$ and $C^*_{max}(\tau')\geq\frac{\sum_{i\in\tau'}p_i}{m}>\frac{x}{2}$. This results in

$$C_{max,S'}=x+\max_{i\in\tau_2'\cup\tau_3'}p_i<2C^*_{max}(\tau')+C^*_{max}(\tau')=3C^*_{max}(\tau').$$

The bounds for PFCFS given in Lemma 2 are tight for $m\gg 1$. Note that the bound of 2 for $\frac{C_S}{C^*}$ cannot be increased by jobs in τ_2' and τ_3' independent of m, x, $|\tau_2'|$, and $|\tau_3'|$. In addition, the bound of 2 can only be approached if the processing time of all jobs in τ_1' is as small as possible. These observations are important for the proof of Lemma 4.

Next, we remove the PAR restriction.

Theorem 1 *Let $m_i\leq\frac{m}{2}$ and $w_i=m_i\cdot p_i$ for all jobs. Then Algorithm PFCFS is 1-fair and will only produce non-preemptive schedules S with the properties*

1. *$C_S < 2 \cdot C^*$ and*

2. *$C_{max,S} < 3 \cdot C^*_{max}$.*

Proof *For the given restrictions Algorithm PFCFS is again identical to the* 1*-fair FCFS method.*

The bounds for $C_{max,S}$ and C_S are proven by induction on the number of PARs occurring during the execution of Algorithm PFCFS. As shown in Lemma 2 the bounds hold for all schedules without any inner PAR, that is, any PAR which is ended by the submission of another job. Therefore, we assume that the bounds are valid for all schedules with at most $k \geq 0$ inner PARs and consider schedules with $k+1$ inner PARs.

Now, let job $i \in \tau$ with release date r_i be the job ending the last inner PAR of schedule S. Then job system τ' and schedule S' are generated from job system τ and schedule S by applying the following transformations:

1. *Any job $j \in \tau$ is horizontally split at time $2r_i$ into two jobs j_1 and j_2, if j is processed at time $2r_i$ in schedule S, that is, $t_j(S) - p_j < 2r_i < t_j(S)$. We say that j_1 is the first descendant of job j, if j_1 is executed before j_2 in the resulting schedule S'.*

2. *If $r_j < r_i$ for a job $j \in \tau$ split at Step 1, then we set $r_{j_2} = r_i$ else $r_{j_2} = r_j$. In both cases we have $r_{j_1} = r_j$.*

3. *Job system τ' is partitioned such that*

$$\begin{aligned} \tau'_1 &= \{j \in \tau' | r_j < r_i\}, \\ \tau'_2 &= \{j \in \tau' | r_j \geq r_i \text{ and } t_j(S') \leq 2r_i\}, \text{ and} \\ \tau'_3 &= \{j \in \tau' | r_j \geq r_i \text{ and } t_j(S') > 2r_i\}. \end{aligned}$$

   Similarly, schedule S' is split into schedules S'_1, S'_2, and S'_3.

Note that in any schedule σ for job system τ we have $t_j(\sigma) - r_i > p_{j_2}$ for any job $j \in \tau$ which is split in τ'. With Corollary 2 we therefore obtain $\frac{C_S}{C^*(\tau)} \leq \frac{C_{S'}}{C^*(\tau')}$ and $\frac{C_{max,S}}{C^*_{max}(\tau)} \leq \frac{C_{max,S'}}{C^*_{max}(\tau')}$. Further, no job in schedule S'_3 starts before time $2r_i$. This results in

$$\begin{aligned} C_{S'} &= C_{S'_1} + C_{S'_2} + C_{S'_3}, \\ C^*(\tau') &\geq C^*(\tau'_1) + C^*(\tau'_2) + C^*(\tau'_3), \\ C_{max,S'} &= \max\{C_{max,S'_1}, C_{max,S'_2}, C_{max,S'_3}\}, \\ C^*_{max}(\tau') &\geq \max\{C^*_{max}(\tau'_1), C^*_{max}(\tau'_2), C^*_{max}(\tau'_3)\}. \end{aligned}$$

Schedule S'_1 is a PFCFS schedule with k inner PARs. Due to the induction assumption the bounds therefore hold for job system τ'_1 and schedule S'_1.

Next, we derive job system τ_3'' from job system τ_3' by subtracting $\min\{r_j, 2r_i\}$ *from the release dates of each job $j \in \tau_3'$. Note that the release date of each job in τ_3' is reduced by at least r_i. Therefore, we have the relations $C^*(\tau_3'') \leq C^*(\tau_3') - r_i \sum_{j \in \tau_3'} w_j$ and $C^*_{max}(\tau_3'') \leq C^*_{max}(\tau_3') - r_i$. Also, schedule S_3' is transformed into schedule S_3'' by starting each job in τ_3'' time $2r_i$ earlier. Note that S_3'' is a PFCFS schedule with no inner PAR. Using Lemma 2 this yields the following relations:*

$$\begin{aligned}
C_{S_2'} &\leq 2r_i \sum_{j \in \tau_2'} w_j < 2C^*(\tau_2'), \\
C_{S_3'} &= C_{S_3''} + 2r_i \sum_{j \in \tau_3'} w_j < 2C^*(\tau_3'') + 2r_i \sum_{j \in \tau_3'} w_j = 2(C^*(\tau_3'') + r_i \sum_{j \in \tau_3'} w_j) \\
&\leq 2C^*(\tau_3'), \\
C_{max,S_2'} &\leq 2r_i < 2C^*_{max}(\tau_2'), \\
C_{max,S_3'} &\leq C_{max,S_3''} + 2r_i < 3C^*_{max}(\tau_3'') + 2r_i < 3(C^*_{max}(\tau_3'') + r_i) \leq 3C^*_{max}(\tau_3'),
\end{aligned}$$

The combination of the results concludes the proof.

A brief look at the example of the workload for a real machine with 430 nodes (see Table 2.2) shows that wide jobs are rather uncommon. This explains the acceptable performance of FCFS schedules in many cases.

It has already been mentioned that preempting a running job and resuming the execution of a previously preempted job results in a preemption penalty s. For the general analysis of Algorithm PFCFS we assume that s is a constant and that the minimal execution time of any job is p. Further, we introduce the relative preemption penalty $\bar{s} = \frac{s}{p}$, see [59]. Without the definition of a minimal execution time there is no constant competitive factor for Algorithm PFCFS in general. But in a real application there is always a minimal execution time due to the time required for loading of a job, even if the job fails to run. Further, in Algorithm PFCFS we set time period $t = p$.

We start this part of the analysis by introducing a generalization of Corollary 2.

Corollary 3 *Assume that $w_i = m_i p_i$ holds for all jobs i in a job system τ. Further, we have a schedule S with $\frac{C_S}{C^*(\tau)} > \gamma$ for some parameter γ. A new job system τ' and a new schedule S' is generated from S and τ by replacing a single job $i \in \tau$ in S with the successive execution of two jobs i_1 and i_2 such that $m_{i_1} = m_{i_2} = m_i$, $p_i = p_{i_1} + p_{i_2}$, $w_{i_1} = m_i p_{i_1}$, $w_{i_2} = m_i p_{i_2}$, $t_{i_2}(S') = t_i(S)$ and $0 < t_{i_2}(S') - t_{i_1}(S') \leq \gamma p_{i_2}$ hold. This results in $\frac{C_{S'}}{C^*(\tau')} \geq \frac{C_S}{C^*(\tau)}$.*

Proof *Note that the transformation produces a legal schedule S' if any minimal job execution time p is ignored. As in Corollary 2 it does not affect the completion time of any job in $\tau \backslash \{i\}$. Further, the relation $w_{i_1} + w_{i_2} = w_i$ holds. Therefore, we obtain with the help of Corollary 2*

$$\begin{aligned}
C_S - C_{S'} &= w_i t_i(S) - w_{i_1} t_{i_1}(S') - w_{i_2} t_{i_2}(S') \\
&\leq (w_{i_1} + w_{i_2}) t_i(S) - w_{i_1}(t_{i_2}(S') - \gamma p_{i_2}) - w_{i_2} t_{i_2}(S') \\
&= \gamma m_i p_{i_1} p_{i_2}, \\
C^*(\tau) - C^*(\tau') &\geq m_i p_{i_1} p_{i_2}.
\end{aligned}$$

The combination of both inequalities finally yields

$$\begin{aligned}
\frac{C_{S'}}{C^*(\tau')} &\geq \frac{C_S - \gamma m_i p_{i_1} p_{i_2}}{C^*(\tau) - m_i p_{i_1} p_{i_2}} \\
&\geq \frac{C_S - \frac{C_S}{C^*(\tau)} m_i p_{i_1} p_{i_2}}{C^*(\tau) - m_i p_{i_1} p_{i_2}} \\
&= \frac{\frac{C_S}{C^*(\tau)}(C^*(\tau) - m_i p_{i_1} p_{i_2})}{C^*(\tau) - m_i p_{i_1} p_{i_2}} \\
&= \frac{C_S}{C^*(\tau)}.
\end{aligned}$$

Next, we take a look at the part of a schedule in which a wide job i causes preemptions of other jobs. We define a so called *preemptive time period* T that starts with the end of Instruction C and ends with the next beginning of Instructions A, B or C. Two cases are possible:

1. Wide job i completes in Instruction I while other jobs are preempted.
2. Enough resources become available to run wide job i to completion after it has already preempted other jobs (Instructions F).

In both cases time period T can be divided into different kinds of subperiods. Table 3.2 describes the length and the minimum resource use for those subperiods with $t = p$. In Table 3.2 the preemption penalty is represented by penalty subperiods a and b. Note that the first and the last penalty subperiod in T relates to either the start or the completion of a gang. As the time requirement to load a job or store its final results is included in the processing time of the job those subperiods last only $\frac{s}{2}$ (subperiod a) while in all other cases the full penalty of s is encountered.

For the first case period T can be represented by the sequence

$$c - a - d - b - e - b - \ldots - d - b - e - b - d - a.$$

Further, we set $k = \lfloor \frac{p_i}{p} \rfloor$ if p_i is not an integer multiple of p and $k = \frac{p_i}{p} - 1$ otherwise. Then, the subsequence $d - b - e - b$ is repeated k times. The last subperiod d has a length of $p' = \hat{p} = p_i - kp \leq p$ while for all other subperiods d and e we have $p' = p$. Also note that in all subperiods e a single long running sequential job may

Subperiod identifier	Length	Minimum resource consumption
a	$\frac{s}{2}$	–
b	s	–
c	p	$p(m - m_i + 1)$
d	$p' \le p$	$p'm_i$
e	$p' \le p$	p'

Table 3.2: Subperiods of a schedule where wide job i causes preemptions

prevent the availability of the necessary resources to execute the rest of job i in a non-preemptive fashion as no job migration is allowed. Hence, the resource-time product in subperiods e is positive but may be very small.

Therefore, the overall length and the minimal resource consumption of T in the first case are

$$\begin{aligned}
\text{Length}(T) &= p + \frac{s}{2} + k(2p + 2s) + \hat{p} + \frac{s}{2} \\
&= (k+1)(2p + 2s) + \hat{p} - p - s \\
&\le (k+1)(2p + 2s) \\
\text{Resource}(T) &> p(m - m_i) + kpm_i + \hat{p}m_i = pm + (k-1)pm_i + \hat{p}m_i \\
&> (k+1)p\frac{m}{2} + \hat{p}\frac{m}{2} \\
&\ge (k+1)p\frac{m}{2} \\
&\ge \frac{m}{4(1+\bar{s})}\text{Length}(T).
\end{aligned}$$

The second case is almost identical to the first case. Only the last subperiod b and the last subperiod d are missing. For this case we therefore have the sequence

$$c - a - d - b - e - b - \ldots - d - b - e - a.$$

Here, the subsequence $d{-}b{-}e{-}b$ is repeated $k \ge 0$ times. Further, the last subperiod e has a length of $p' = \hat{p} \le p$ while $p' = p$ holds for all other subperiods d and e. In contrary to the first case job i will continue execution after period P but not cause any preemption of other jobs anymore. For the overall length and the minimal resource consumption of T we have

$$\begin{aligned}
\text{Length}(T) &= p + \frac{s}{2} + k(2p+2s) + p + s + \hat{p} + \frac{s}{2} \\
&< (k+2)(2p+2s) - p - 2s \\
\text{Resource}(T) &> p(m-m_i) + kpm_i + pm_i \\
&= pm + kpm_i \\
&> (k+2)p\frac{m}{2} \\
&> \frac{m}{4(1+\bar{s})}\text{Length}(T).
\end{aligned}$$

Finally, the case must be considered when enough resources become available to process a wide job i before it causes any preemption (Instruction E_1). In this case between $m - m_i + 1$ and $\frac{m}{2}$ resources are used for a period $p' \leq p$. Here, we define a *degenerated preemptive time period* T' consisting of a subperiod c with length p' and a subperiod that is formed by the first p' processing of i, that is the degenerated preemptive period does not stop with the beginning of the next instruction A, B or C. For such a degenerated preemptive period we have Length(T')$=2p'$ and Resource(T')$> p(m-m_i) + p'm_i = p'm = \frac{1}{2}$Length($T'$).

Note that any PFCFS schedule only consists of PARs, periods with preemptions as described above and periods with a minimum resource consumption of more than $\frac{m}{2}$ at any time. Therefore, we can evaluate the minimum degree of machine utilization for the general case.

Lemma 3 *At least $\frac{1}{4+4\bar{s}}$ of the resources are used on average during any time frame of a schedule produced by Algorithm PFCFS if the time frame*

- *does not include any part of a PAR and*
- *does not start or end during a preemptive period or a degenerated preemptive period.*

The first condition simply prevents underutilization due to a lack of jobs. Due to the second condition any time frame allowed by Lemma 3 cannot only partially include a preemptive or degenerated preemptive period to avoid any temporary underutilization.

Proof *At any time instance not belonging to a preemptive or degenerated preemptive period (Instructions A, B or C) more than 50% of the resources are used as no PAR is allowed. During any preemptive or degenerated preemptive period at least $\frac{1}{4+4\bar{s}}$ resources are used on average as has been shown above.*

The average usage of resources can be significantly increased in a realistic setting if we allow the concurrent execution of small jobs together with a preemption causing wide job. This is also true if migration is allowed. However, those architectural modifications may also cause a larger preemption penalty.

Further note that after the execution of a job is resumed following its preemption, the job will either run to completion or execute for a time period of at least p before it is preempted again. Then the job will stay inactive for at most $p + 2s$. Therefore, we can apply Corollary 3 with $\gamma = 2 + 2\bar{s}$ to a PFCFS schedule provided any job within a preemptive period can only be split when its execution is resumed following a preemption. Then the first descendent of the split job has a remaining execution time of 0 after the preemption. Outside of those preemptive periods any job may be split at any time.

To determine the competitive factors for preemptive PFCFS schedules we again consider at first schedules without any inner PARs, that is, schedules where a PAR only occurs after the submission of the last job.

Lemma 4 *Let $w_i = m_i \cdot p_i$ for all jobs. Also assume that there is no PAR before the submission of the last job. Then Algorithm PFCFS is $(2 + 2\bar{s})$-fair and produces schedules S with the properties*

1. *$C_S < (3.562 + 3.386\bar{s})C^*$ and*
2. *$C_{max,S} < (4 + 3\bar{s})C^*_{max}$.*

Proof *As already mentioned above the completion time of a job i may increase at most by $p_i(1 + 2\bar{s})$ due to a wide job which is submitted later and causes preemption. This yields the fairness result.*

Note that the application of Corollary 3 is useful as $\gamma = 2 + 2\bar{s} < 3.562 + 3.386\bar{s}$.

As in the proof of Lemma 2 we start by describing a class of job systems τ' and schedules S' with the following properties:

1. *τ' is partitioned into 3 subsets τ'_1, τ'_2 and τ'_3.*
2. *For every job $i \in \tau'_1$ there is $m_i = \frac{m}{2}$.*
3. *All jobs of τ'_1 are scheduled first in S' such that job i starts $(1 + 2\bar{s})p_i$ after the completion of the preceding job (or after time 0 if i is the first job).*
4. *τ'_2 contains $\frac{y}{p}$ wide jobs i with $p_i = p$ and $m_i = \frac{m}{2}$ and $\frac{ym}{2p}$ sequential jobs j with $p_j = p$ and $m_j = 1$.*
5. *1 wide job of τ'_2 is always started concurrently with $\frac{m}{2}$ sequential jobs of τ'_2 in schedule S'. Such a block of jobs is started $2p + 3s$ after the completion of the previous block (or after the last job of τ'_1 if it is the first block).*
6. *τ'_3 only contains sequential jobs, that is jobs i with $m_i = 1$.*

7. *In schedule S' all jobs in τ'_3 are executed after the jobs in τ'_2 such that at any time instant between the completion of the last job in τ'_2 and the completion of the last job in τ'_3 exactly $\frac{m}{2}$ resources are used.*

Note that τ' only consists of sequential jobs and wide jobs which use 50% of the resources. For reasons of convenience we introduce the variables $x = \sum_{i\in\tau'_1} p_i$ and $z = \frac{2\sum_{i\in\tau'_3} p_i}{m}$. With the help of Corollary 1 we can now calculate the cost of schedule S' to be

$$\begin{aligned}
C_{S'} &= \frac{(2+2\bar{s})m}{4}(x^2 + \sum_{i\in\tau'_1} p_i^2) + \\
&\quad \frac{(3+3\bar{s})m}{2}(y^2+yp) + my(2+2\bar{s})x + \\
&\quad \frac{m}{4}z^2 + \frac{1}{2}\sum_{i\in\tau'_3} p_i^2 + \frac{m}{2}z((2+2\bar{s})x + (3+3\bar{s})y).
\end{aligned}$$

For the optimal schedule we demand that no two wide jobs can be executed concurrently. Using the same convexity argument as in the proof of Lemma 2 we obtain

$$C^* = \begin{cases} \frac{m}{2}(y^2+yp) + \frac{m}{4}(x^2 + \sum_{i\in\tau'_1} p_i^2) + \frac{m}{2}xy + \frac{m}{4}z^2 + \frac{1}{2}\sum_{i\in\tau'_3} p_i^2 + \frac{m}{2}zy & \text{if } x \geq z \\ \frac{m}{2}(y^2+yp) + \frac{m}{4}(2x^2 + \sum_{i\in\tau'_1} p_i^2) + mxy + \frac{m}{8}(z-x)^2 & \\ +\frac{1}{2}\sum_{i\in\tau'_3} p_i^2 + \frac{m}{2}(z-x)(x+y) & \text{if } x < z. \end{cases}$$

If we consider only cases with $\frac{C_{S'}}{C^*(\tau')} \geq 3 + 3\bar{s}$ the proof idea of Corollary 3 can be used to obtain

$$\begin{aligned}
\frac{C_{S'}}{C^*(\tau')} &\leq \frac{C_{S'} - (3+3\bar{s})(\frac{m}{4}\sum_{i\in\tau'_1} p_i^2 + \frac{m}{2}yp - \frac{1}{2}\sum_{i\in\tau'_3} p_i^2)}{C^*(\tau') - \frac{m}{4}\sum_{i\in\tau'_1} p_i^2 - \frac{m}{2}yp - \frac{1}{2}\sum_{i\in\tau'_3} p_i^2} \\
&\leq \frac{C_{S'} - (2+2\bar{s})\frac{m}{4}\sum_{i\in\tau'_1} p_i^2 - (3+3\bar{s})\frac{m}{2}yp - \frac{1}{2}\sum_{i\in\tau'_3} p_i^2}{C^*(\tau') - \frac{m}{4}\sum_{i\in\tau'_1} p_i^2 - \frac{m}{2}yp - \frac{1}{2}\sum_{i\in\tau'_3} p_i^2}.
\end{aligned}$$

Therefore, it is sufficient to determine the upper bound for the ratios

$$\begin{array}{ll}
\dfrac{(1+\bar{s})(2x^2+6y^2+8yx+4zx+6zy)+z^2}{x^2+2y^2+2xy+z^2+2zy} & \text{for } x \geq z \text{ and} \\
\dfrac{(1+\bar{s})(2x^2+6y^2+8yx+4zx+6zy)+z^2}{2x^2+2y^2+4xy+\frac{1}{2}(z-x)^2+2(z-x)(x+y)} & \text{for } x < z.
\end{array}$$

The solution of this algebraic optimization problem produces

$$\begin{aligned}
\frac{2x^2+6y^2+8yx+z^2+4zx+6zy}{x^2+2y^2+2xy+z^2+2zy}|_{z\geq 0,y\geq 0,x\geq z} &\leq \\
\frac{2x^2+6y^2+8yx+z^2+4zx+6zy}{x^2+2y^2+2xy+z^2+2zy}|_{y=0,x=1.281z} &= 3.562 \\
\frac{2x^2+6y^2+8yx+4zx+6zy}{x^2+2y^2+2xy+z^2+2zy}|_{z\geq 0,y\geq 0,x\geq z} &\leq \\
\frac{2x^2+6y^2+8yx+4zx+6zy}{x^2+2y^2+2xy+z^2+2zy}|_{y=z,x=1.886z} &= 3.386
\end{aligned}$$

$$\begin{aligned}
\frac{2x^2+6y^2+8yx+z^2+4zx+6zy}{2x^2+2y^2+4xy+\frac{1}{2}(z-x)^2+2(z-x)(x+y)}|_{x\geq 0,y\geq 0,z>x} &< 3.562 \\
\frac{2x^2+6y^2+8yx+4zx+6zy}{2x^2+2y^2+4xy+\frac{1}{2}(z-x)^2+2(z-x)(x+y)}|_{x\geq 0,y\geq 0,z>x} &< 3.386.
\end{aligned}$$

Now, it remains to be shown that every job system τ and every PFCFS schedule S with no inner PAR can be transformed into a job system τ' and a schedule S' with $\frac{C_{S'}}{C^*(\tau')} \geq \frac{C_S}{C^*(\tau)}$. To this end we use four elementary transformations each of which cannot decrease the ratio $\frac{C_S}{C^*}$ provided $\frac{C_S}{C^*} > 2+2\bar{s}$ holds:

1. *Horizontal splitting of jobs as described in Corollary 3 with $\gamma = 2+2\bar{s}$*
2. *Vertical splitting of jobs*
3. *Increasing the completion time of some jobs without affecting the completion time of any other job in the schedule*
4. *Earlier start or introduction of empty subperiods (periods in which no job is processed)*

In detail we apply the following sequence of transformations:

1. *If a job executes during a preemptive or a degenerated preemptive period P and starts or terminates outside of P in the PFCFS schedule it is horizontally split at the beginning and/or at the end of P. This separates all those periods from the rest of the schedule.*
2. *In a degenerated preemptive period of length $2p'$ vertical splitting is applied to all jobs in the period such that there is one wide job i' with $m_{i'} = \frac{m}{2}$ and $p_{i'} = p'$ while all other jobs in this period are sequential.*

3. *The preemption causing wide job i in a preemptive period P is horizontally split at the beginning of the second subperiod d if there are at least two subperiods d. This produces a wide job of execution time p and another wide job of execution time $p_i - p$. In case of a degenerated preemptive period there is only a single wide job i with processing time $p_i < p$.*

4. *All other jobs belonging to this preemptive period are horizontally split at the beginning of the first subperiod e.*

5. *All jobs executing during subperiods e are executed at the end of the schedule in a non-preemptive fashion. Note that this does not include jobs executing during subperiod c which complete at the beginning of the first subperiod e.*

6. *Both subperiods a, the first two subperiods b, and the first subperiod e are moved to the beginning of the preemptive period.*

7. *Vertical splitting is applied to the first wide job (executing in the first subperiod d, see Step 3) such that one wide job i' with $m_{i'} = \frac{m}{2}$ and $m_i - \frac{m}{2}$ sequential jobs are generated.*

8. *Vertical splitting is applied to the jobs in subperiod c such that all jobs in this subperiod are sequential.*

9. *$m - m_i$ of the sequential jobs in subperiod c are executed concurrently with wide job i' in the following subperiod d while the remaining jobs are executed at the end of the schedule.*

10. *All subperiods e (except the first one which has been moved to the beginning of the preemptive period P, see Step 6) and all subperiods b (except the first two, see Step 6) are scheduled immediately after the completion of the first wide job in this preemptive period.*

11. *Vertical splitting is applied to the second descendant of wide job i (with execution time $p_i - p$, see Step 3) such that there is a single wide job i'' with $m_{i''} = \frac{m}{2}$ and $m_i - \frac{m}{2}$ sequential jobs. All those sequential jobs are executed at the end of the schedule.*

12. *For each degenerated period of length $2p'$ an empty period of length $p' + 3\bar{s}p'$ is introduced at its beginning. This way a degenerated preemptive period has a resource ratio of $\frac{1}{3+3\bar{s}}$.*

13. *The transformations of Lemma 2 are applied to all jobs not executing during any preemptive period such that always exactly $\frac{m}{2}$ resources are used except for any preemptive period and the final PAR.*

With these transformations (and by possibly introducing additional periods without any resource use) we have generated periods with a use of $\frac{1}{2}$, $\frac{1}{3+3\bar{s}}$, and $\frac{1}{4+4\bar{s}}$ of the available resources.

Next, Smith's rule [69] is generalized to those periods. As $\frac{p_i m_i}{w_i}$ is invariant for all jobs, the ratio of used resources to available resources within a period (resource ratio) corresponds to the ratio $\frac{weight}{processing\ time}$ in uniprocessor scheduling. Therefore, the largest weighted completion time cost of the schedule is obtained by executing periods with the smallest resource ratio first. This produces the ordering in the description of S'.

It remains to be shown that any jobs executing during the final PAR of the schedule cannot result in a larger maximum value for $\frac{C_{S'}}{C^*(\tau')}$. To this end we consider two job systems $\hat{\tau}'$ and τ' and the corresponding schedules $\hat{S}'$ and S'. Both job systems only differ in the subsets $\hat{\tau}'_4 \subseteq \hat{\tau}'_3$ and $\tau'_4 \subseteq \tau'_3$.

1. *Job system $\hat{\tau}'_4$ contains less than $\frac{m}{2}$ sequential jobs that all start at the beginning of the last PAR in schedule $\hat{S}'$. This can be achieved by use of Corollary 2.*

2. *Job system τ'_4 contains $\frac{\sum_{i\in\hat{\tau}'_4} p_i}{p}$ identical sequential jobs with processing time p. Those jobs are executed last in schedule S' such that exactly $\frac{m}{2}$ machines are used for their processing and all machines are idle in the final PAR.*

With the help of Corollary 1 we obtain:

$$\begin{aligned}
C_{\hat{S}'} - C_{S'} &= \sum_{i\in\hat{\tau}'_4} p_i^2 - \frac{m}{4}\left(\frac{2\sum_{i\in\hat{\tau}'_4} p_i}{m}\right)^2 - \frac{1}{2}\frac{\sum_{i\in\hat{\tau}'_4} p_i}{p} p^2 \\
&= \sum_{i\in\hat{\tau}'_4} p_i^2 - \frac{(\sum_{i\in\hat{\tau}'_4} p_i^2}{m}) - \frac{p\sum_{i\in\hat{\tau}'_4} p_i}{2} \\
C^*(\hat{\tau}') - C^*(\tau') &\geq \frac{1}{2}\left(\sum_{i\in\hat{\tau}'_4} p_i^2 - p\sum_{i\in\hat{\tau}'_4} p_i\right) > \frac{1}{2}(C_{\hat{S}'} - C_{S'})
\end{aligned}$$

Using the proof idea of Corollary 3 we can therefore show that any jobs executing in the final PAR cannot increase the ratio of $\frac{C_{S'}}{C^*(\tau')}$ beyond $3.562 + 3.386\bar{s}$.

Finally note that each job i' in τ' with $m_{i'} = \frac{m}{2}$ is a descendant of a job i in τ with $m_i > \frac{m}{2}$. Therefore, any two of those jobs cannot be executed concurrently in any schedule. This justifies the additional constraint in the optimal schedule for τ'.

For the makespan we use the same definitions of x, y, and z. In addition we now assume that τ' also contains jobs which execute during the final PAR. Then, the following bounds hold for the optimal makespan:

$$\begin{aligned}
C^*_{max}(\tau') &\geq p_i \text{ for all jobs } i \in \tau' \\
C^*_{max}(\tau') &\geq x + y \\
C^*_{max}(\tau') &\geq y + \frac{1}{2}(x + z)
\end{aligned}$$

With j being the job that finishes last in schedule S' we obtain:

$$\begin{aligned} C_{max,S} &\leq C_{max,S'} \\ &\leq p_j + z + (3+3\bar{s})y + (2+2\bar{s})x \\ &\leq p_j + 2(y + \frac{1}{2}(x+z)) + (1+3\bar{s})(x+y) \\ &\leq (4+3\bar{s})C^*_{max}(\tau') \leq C^*_{max}(\tau) \end{aligned}$$

Finally, we again remove the constraint on the number of PARs.

Theorem 2 *Let $w_i = m_i \cdot p_i$ for all jobs. Then Algorithm PFCFS will only produce schedules S with the properties*

1. *S is $(2+2\bar{s})$-fair,*
2. *$C_S < (3.562 + 3.386\bar{s})C^*$, and*
3. *$C_{max,S} < (4+3\bar{s})C^*_{max}$.*

Proof *In general the proof is done in the same fashion as the proof of Theorem 1. However, due to the existence of preemptive periods there are some differences. Because of the definition of a PAR time s_i cannot belong to a preemptive period.*

We start by addressing the completion time cost of schedule S. First assume that $r_i \leq 2p + 2s$. We define $\tau_1 = \{j \in \tau | r_j < r_i\}$ and consider two situations:

1. *Every job $j \in \tau_1$ starts at time r_j. Then job system τ' is generated by splitting each job $j \in \tau_1$ into jobs j_1 and j_2 at time r_i if $t_j(S) > r_i$. As in the proof of Theorem 1 we have $t_j(\sigma) - r_i \geq p_{j_2}$ in any schedule σ for job system τ. We can now prove the claims by applying Lemma 4 to job system $\tau'_2 = \{j \in \tau' | r_j \geq r_i\}$.*
2. *Otherwise we have $\sum_{j\in\tau_1} m_j > m$ and $\sum_{j\in\tau \wedge t_j(S) < r_i} m_j p_j > \frac{mp}{2}$. Therefore, we can modify τ and S by replacing the schedule between time 0 and r_i with a period using $\frac{1}{4+4\bar{s}}$ of the available resources as described in the proof of Lemma 4. This will remove all inner PARs and increase $\frac{C_S}{C^*(\tau)}$.*

Therefore, it is sufficient to consider $r_i > 2p + 2s$. As in the proof of Theorem 1 we want to use horizontal splitting to generate a job system τ'_3 with $r_j > r_i$ for all jobs $j \in \tau'_3$ and a corresponding pair of schedules S'_3 and S''_3 such that Lemma 4 can be applied to S''_3. Let us consider time $t = (3.562 + 3.386\bar{s})r_i$. If t does not fall into a preemptive period then each job is split at time t provided it is executing at this moment. Note that

$$p_{j_1} \geq \frac{3.562 + 3.386\bar{s} - 1}{2+2\bar{s}} r_i > r_i$$

holds if $r_j < r_i$. On the other hand if t is a time instant within a preemptive period we first move the final subperiod a in schedule S down to the first subperiod a, see also the proof of Lemma 4, Step 6 of the transformation. This can only increase C_S. Now we must address two cases:

1. *If t belongs to a subperiod a or c then all jobs are horizontally split before the preemptive period, that is, at time $t - p - s > t - \frac{1}{2}r_i$ or later. Therefore, we have*
$$p_{j_1} \geq \frac{3.562 + 3.386\bar{s} - 1 - \frac{1}{2}}{2 + 2\bar{s}} r_i > r_i$$
if $r_j < r_i$.

2. *In all other situations all jobs j with $r_j < r_i$ are split at the end of the preemptive period while the wide job is split at the beginning of the last subperiod d before t. As any jobs executing during subperiods e are not used in the proof of Lemma 4 we are allowed to split those jobs after t thus guaranteeing $p_{j_1} > r_i$. However, wide jobs may be split as early as $t - (2 + 2\bar{s})p > (2.562 + 2.386\bar{s})r_i$. This also does not affect the validity of Lemma 4 for τ_3' and S_3''.*

The proof of Theorem 1 can be directly applied to all jobs $j \in \tau_2'$, that is, jobs j with $r_j \geq r_i$ and $t_j(S') \leq t$.

Finally, we must consider all jobs $j \in \tau'$ with $t_j(S') > r_i$ and $r_j < r_i$. Unfortunately, Lemma 4 can only be applied if those jobs are not preempted after time s_i. Using Lemma 3 it can be assumed that those jobs start with the beginning of the last inner PAR. Again, we must distinguish 3 cases:

1. *j is not preempted.*
2. *$p_j \geq 0.640 r_i$.*
3. *j is preempted and $p_j < 0.640 r_i$.*

As already stated Lemma 4 is valid in the first case. For the second case we use the fairness property to compare the completion time of job j in schedule S' with the earliest completion time of j in any schedule σ:

$$\begin{aligned} t_j(S') &= r_i + (2 + 2\bar{s})p_j \\ &\leq (\frac{1}{0.640} + 2 + 2\bar{s})p_j \\ &\leq (1.562 + 2 + 2\bar{s})t_j(\sigma) \\ &\leq (3.562 + 3.386\bar{s})t_j(\sigma) \end{aligned}$$

Therefore, those jobs can be split at time r_i and considered separately from τ_1'.

Any job j belonging to the third case will complete in schedule S' before time $r_i + (2 + 2\bar{s})0.640 r_i = (2.280 + 1.280\bar{s})r_i < t - r_i \leq t - (2 + 2\bar{s})p$, that is, before the start of any wide job assigned to τ_3'. To address such a job j we first vertically split each wide job κ that preempts j to generate a job k with $m_k = m_j$ that preempts j. If the completion time of any such job k is larger than $t_j(S)$ then k is horizontally split after the preemption following $t_j(S)$. The set of all those jobs k is denoted *by τ_k. Next, we rearrange the schedule such that j is only preempted just before its completion by possibly increasing the completion time of k'. Therefore, the completion time cost of the resulting schedule S'' cannot be smaller than $C_{S'}$ and $t_j(S'')-p-s \leq \max_{k\in\tau_k}\{t_k(S'')\} \leq t_j(S'')-p-s$. Finally, we horizontally split job j at the beginning of its first preemptive period after r_i. Then job j_1 belongs to Case 1 while we can use $r_{j_2} = p_{j_1}$ and $r_k = r_i$ for all $k \in \tau_k$. From the proof of Lemma 4 we know that $\sum_{k\in\tau_k} p_k = \frac{2a}{3}p_{j_2}$ with $\frac{3}{2} \geq a \geq 1$. Now, we can determine the contribution of job j_2 and jobs $k \in \tau_k$ to the completion time cost of S'' (upper bound) and the cost of the optimal schedule $C^*(\tau')$ (lower bound):*

$$
\begin{aligned}
C_{S''} \; : \; & m_j p_{j_2} t_{j_2}(S'') + m_j \sum_{k\in\tau_k} p_k t_k(S'') \leq \\
& m_j(p_{j_2}(r_i + p_{j_1}) + p_{j_2}^2(2+2\bar{s}) + (\sum_{k\in\tau_k} p_k)(r_i + p_{j_1} + \frac{3+3\bar{s}}{2}\sum_{k\in\tau_k} p_k) + \\
& \frac{3+3\bar{s}}{2}\sum_{k\in\tau_k} p_k^2) = \\
& m_j p_{j_2}(r_i + p_{j_1} + p_{j_2}(2+2\bar{s}) + \frac{2a}{3}(r_i+p_{j_1}) + (1+\bar{s})\frac{2a^2}{3}p_{j_2}) + \\
& \frac{3+3\bar{s}}{2}\sum_{k\in\tau_k} p_k^2) = \\
& m_j(p_{j_2}(\frac{2a+3}{3}(r_i+p_{j_1}) + \frac{2a^2+6}{3}(1+\bar{s})p_{j_2})) + (3+3\bar{s})\frac{1}{2}\sum_{k\in\tau_k} p_k^2)
\end{aligned}
$$

$$
\begin{aligned}
C^*(\tau') \; : \; & m_j(p_{j_2}(p_{j_1}+p_{j_2}) + (\sum_{k\in\tau_k} p_k)(r_i + \frac{1}{2}\sum_{k\in\tau_k} p_k) + \frac{1}{2}\sum_{k\in\tau_k} p_k^2) = \\
& m_j(p_{j_2}(p_{j_1} + p_{j_2} + \frac{2a}{3}p_{j_2} r_i + \frac{2a^2}{9}p_{j_2}) + \frac{1}{2}\sum_{k\in\tau_k} p_k^2) = \\
& m_j(p_{j_2}(\frac{2a}{3}r_i + p_{j_1} + \frac{2a^2+9}{9}p_{j_2}) + \frac{1}{2}\sum_{k\in\tau_k} p_k^2) \\
Ratio \; : \; & p_{j_2}(\frac{\frac{2a+3}{3}(r_i+p_{j_1}) + \frac{2a^2+6}{3}(1+\bar{s})p_{j_2}) + \frac{3+3\bar{s}}{2}\sum_{k\in\tau_k} p_k^2}{p_{j_2}(\frac{2a}{3}r_i + p_{j_1} + \frac{2a^2+9}{9}p_{j_2}) + \frac{1}{2}\sum_{k\in\tau_k} p_k^2}) < 3.562 + 3.386\bar{s}
\end{aligned}
$$

The proof for the optimal makespan follows the proof from Theorem 1 and Lemma 4. The relations $C_{max,S'} \geq C_{max,S}$ and $C^*_{max}(\tau') \leq C^*_{max}(\tau)$ hold with respect to all transformations described above. The addition of inner PARs in Theorem 2 does not alter the lower bound for the optimal makespan as used in the proof for Lemma 4: $C^*_{max}(\tau') \geq \max_{i\in\tau'}\{p_i\}$, and $C^*_{max}(\tau') \geq \frac{\sum_{i\in\tau'} p_i}{m}$, and $C^*_{max}(\tau') \geq r_i$

For the makespan in S and S' we only have to consider the completion of the last job j. Following the assumption and modifications of the completion time proof for

$j \in \tau_1$ and $r_i \leq 2p + 2\bar{s}$ and $t_j(S') > r_i$, the third case on Page 59 is considered for the worst-case. Lemma 1 applies accordingly for $C_{max,S'} \leq (4 + 3\bar{s})C_{max^*}(\tau_1')$. For $r_i > 2p + 2\bar{s}$ and the job beginning with the last inner PAR, we get an upper bound:

$$C_{max,S'} \leq r_i + p_j(2 + 2\bar{s}) \leq (4 + 3\bar{s})C^*_{max}(\tau')$$

3.4 Summary

The developed PFCFS algorithm is an example for the design of an algorithm as shown in Chapter 2. Our algorithm improves the FCFS strategy by introducing preemption on the parallel machine. As mentioned in Section 3.3, preemption is used to temporarily interrupt a job and continue it later. Our algorithm is well suited for fair on-line scheduling of parallel jobs. Fairness is achieved by selecting job weights to be equal to the resource consumption of the job and by limiting the time span a job can be delayed by other jobs submitted after it. Further, the processing time of a job is not known when the job is released. It is proven that the algorithm achieves a constant competitive ratio for both the makespan and the weighted completion time

We addressed the issue of fairness in parallel job scheduling. We used a selection of job weights that do not result in a prioritization of small jobs over wide jobs and vice versa. The presented algorithm prevents job starvation and achieves constant competitive factors for both, the makespan and the weighted completion time criteria with the given weight selection. As an extension it is possible to use different job priorities by supporting different values of the ratio *weight* to *resource consumption*. To each group of jobs with the same ratio the algorithm can be applied separately. The final schedule is obtained by interleaving the different schedules and assigning time frames to them in accordance with their ratio values.

The theoretical approach on the evaluation gives us valuable information on the worst-case behavior. Nevertheless, competitive factors are worst case factors which are frequently not acceptable for practical use. For instance, a competitive factor of 2 for the machine load of a schedule denotes that in some cases 50% of the resources are not used. On the other hand, the corresponding input data to achieve this worst case result typically cannot be found in real job traces. We can compare the results from the approximation analysis of known theoretical algorithms as mentioned in Section 3.2.2 with our example simulation results from Section 2.8. Here, we see that the theoretical approximation factors give bounds that address the question if the scheduling system can guarantee that it *always* produces schedules with a certain quality. However, the theoretical analysis does not take workload and user demand into account. Therefore, it does not indicate how this algorithm performs under real workload compared to some other algorithms. This shows a common limitation for the theoretical analysis in the evaluation process for the algorithm design. Frequently, this is also true if randomization is used for the analysis. Additionally, it is often very complex or even impossible to apply theoretical competitive analysis to an algorithm. For example, heuristic algorithms, or very complex strategies, or economic models may be difficult to examine by theoretical analysis if the number of dependencies and different cases grow. It may require a lot of effort to prove a relevant or even tight bound.

Overall, the competitive analysis can provide valuable information for the design of a scheduling system with real workload. In our example, we now know worst-case guarantees and fairness properties of our algorithm. Nevertheless, we do not yet know the actual usability in a real installation of the PFCFS algorithm. Theoretical analysis does not help us in this kind of evaluation. Thus, we apply experimental analysis to further evaluate the algorithms in the following.

Part III

Experimental Evaluation

Chapter 4

Scheduling Simulations

We have seen in the previous chapter that performance evaluation is a difficult task. Theoretical worst case analysis is only of limited help as typical workloads on production machines never exhibit the specific structure that will create a really bad case. Further, there is no random distribution of job parameter values, see e.g. Feitelson and Nitzberg [23]. Hence, a theoretical analysis of random workloads will not provide the desired information either. A trial and error approach on a commercial machine will be tedious and may affect system performance in a significant fashion. Therefore, most users will probably object to such an approach except for the final fine tuning. This just leaves simulation for all other cases.

Simulation may either be based on real trace data or on a workload model. While workload models, see e.g. Jann et al. [46] or Feitelson and Nitzberg [23], enable a wide range of simulations by allowing job modifications, like a varying amount of assigned processor resources, the consistence with real workloads cannot always be guaranteed. This is especially true in the case of workloads whose characteristics change over time. On the other hand, trace data restrict the freedom to select different allocation and scheduling strategies in the case of simulations as the performance of a specific job is only known under the given circumstances. For instance, trace data specifying the execution time of a batch job do not provide similar information, if the job would be assigned to a different subset of processors. Therefore, the selection of the base data for the simulation depends on the circumstances determined by the scheduling strategy and the MPP architecture. There are already a variety of examples for evaluation via simulation based on a workload model, see e.g. Feitelson [20], Feitelson and Jette [22] or on trace data, see e.g. Wang et al. [84].

We want to use simulations to further analyze the PFCFS algorithm from the previous chapter. Here, we describe the use of trace data for the evaluation of different parameter settings for a preemptive scheduling strategy.

4.1 The Workload Data

As already mentioned, we use the workload traces originating from the IBM SP2 of the Cornell Theory Center. We have already used this data for the example in Section 2.8.3. In our simulation we again assume batch partitions of 128 and 256 nodes respectively. Jobs with an node allocation exceeding 128 or 256 nodes are ignored.

In our simulation experiments for the PFCFS scheduler a wide job only preempts enough currently running jobs to generate a sufficiently large partition as is typically done in real parallel computers using preemption. Those small jobs are selected by use of a simple greedy strategy. This modification of Algorithm PFCFS does not affect the theoretical analysis but improves the schedule performance for real workloads significantly.

We generated our own FCFS schedule as a reference schedule and did not use the CTC schedule as, for instance, some jobs were submitted in October and started in November. Using the CTC schedule would not allow to evaluate each month separately. As the preemption penalty for the commercial IBM gang scheduler is less than a second it is neglected ($\bar{s} = 0$).

A large number of simulations with different preemption strategies have been done. For each strategy and each month we determined the makespan, the total weighted completion time and the total weighted flow time with the described selection of weights and compared these values with the results of a simple non-preemptive FCFS schedule. For a better understanding in Table 4.1 three selected examples of the examined preemption strategies are explained with certain parameters. Additional simulations with variations on the parameters have also been executed and are presented below.

Note that $PFCFS_1$ is similar to the strategy used for the theoretical analysis except for the selection of the gang length. As already mentioned the gang length has been selected to be p in the theoretical analysis to guarantee a constant competitive factor. In a practical setting the gang length is a parameter of the scheduling algorithm.

In order to obtain information on the influence of the parameters and to determine good parameter values, various simulations have been done. For each strategy and each month we determined the makespan, the total flow time, and the total weighted flow time using our weight selection and compared these values with the results of a simple non-preemptive FCFS schedule. To compare the algorithm performance with a non-FCFS strategy, we made simulations with a backfilling scheduling policy.

Further, we examined variations on the specification of wide jobs. We also simulated with different gang lengths x and start delays δ before the start of the preemption.

Although the work traces of the Cornell Theory Center reflect a batch partition comprising 430 nodes, there are only very few jobs in these traces that use between 215 ($\frac{R}{2}$) and 430 nodes, as seen in Table 2.2. A reason is the tendency to use a number of processors which is equal to a power of 2. Taking into account that a noticeable gain by PFCFS can only be expected for a suitable number of wide jobs, we assume

$PFCFS_1$	A wide job causes preemption after it has been waiting for at least 10 min. Then the two gangs are preempted every 10 min until one gang becomes empty.
$PFCFS_2$	A wide job causes preemption after it has been waiting for at least 1 min. After running for 1 min the wide job is preempted and the previously preempted jobs are resumed. Finally, the wide job waits until the partition becomes empty (no further preemption).
$PFCFS_3$	A wide job causes preemption after it has been waiting for at least 10 min. Then it runs to completion. Afterwards, the preempted jobs are resumed.

Table 4.1: Different Preemption Strategies

parallel computers with 128, respectively 256 resource. This is also reasonable as most installations only have a smaller number of nodes, and it can be assumed that there the percentage of wide jobs will be larger than for the CTC. This approach prevents the direct comparison of the simulation results with the original schedule data of the CTC.

In order to evaluate our scheduling strategy we assume a homogeneous MPP, that is a computer with identical nodes. As already mentioned, the batch partition of the CTC SP2 consists mainly of thin nodes with 128 or 256 MB memory. Therefore, our reference computer is almost homogeneous in this respect. Special hardware requests of the jobs are currently ignored as we assume a homogeneous MPP system. It is easy to include such requests into the scheduling process. Additional constraints limit the number of available resources for a job while the scheduling process for this reduced set of resources follow our homogeneous model.

Table 4.2 shows the parameter spectrum of our simulations.

The theoretical analysis [61] suggests $x = 50\%$. We chose several values around 50% to determine the sensitivity of the schedule quality on this parameter. While good theoretical results require a potentially unlimited number of preemptions we wanted to determine whether in practice a restriction of this parameter is sufficient. The time period between two context switches Δ is selected to span a wide range. However, to prevent a significant affect of the preemption penalty a minimum value of 60 seconds was used. The theory also suggests to set the gang length to $\delta = \Delta$. We additionally used two fixed values for δ.

Finally, we ignored any preemption penalty in our simulations although it can easily

x Wide job spec.	n Maximum preemptions	Δ Start Delay	δ Gang Length
40%	1	60 sec	1 sec
45%	2	120 sec	60 sec
50%	3	240 sec	Δ
55%	10	600 sec	
60%	11	1800 sec	
		3600 sec	

Table 4.2: Different Parameter Settings for Simulation Experiments

be included into the simulator. Moreover, as migration is not needed and only local hard disks are used for saving of job data and job status, the strategy allows gang scheduling implementations with a relatively small preemption penalty. Note that a preemption penalty of 1 second as assumed in [84] will still be small compared to the minimal value of Δ. The gang length δ of 1 sec is considered for comparing the influence of this parameter for shorter and more frequent gang switches. In a real implementation such a small value has to be considered in comparison to an actual preemption penalty.

Finally, our experiments were conducted for each month separately to determine the variance of our results. This is another reason why the CTC schedule data could not be used for comparison as in the original schedule there are some jobs which were submitted in one month and executed in another.

4.2 Analysis of the Results

First we examine the results for the different example strategies described in Table 4.1. As evaluation criteria the makespan, completion time and flow time in comparison to standard FCFS are used in Tables 4.3 to 4.5. We can see, that the $PFCFS_2$ consistently outperforms FCSFS and the other of the sample PFCFS strategies. We noticed that the schedule quality is very sensitive to the preemption strategy. While significant improvements over FCFS are possible, FCFS may still outperform some preemptive methods. Although there may be general explanations for some phenomena, like the short execution times of wide jobs which fail to run due to a bug, the workload certainly plays a significant role. This can be best demonstrated by looking at the total weighted flow time where strategy $PFCFS_2$ produced almost 50% improvement over FCFS for November 1996 while it performed more than 9% worse than FCFS in January 1997, see Table 4.5.

Figure 4.1 shows that the preemptive FCFS strategy was able to improve FCFS for all scheduling criteria if the best parameter setting was used, see Table 4.6. The results for the FCFS denote 100%. The preemptive FCFS strategy also outperformed the backfilling scheduling policy in all cases. This algorithm does not utilize

	$PFCFS_1$	$PFCFS_2$	$PFCFS_3$
Jul 96	-3.5%	-16.3%	+20.0%
Aug 96	-8.3%	-22.2%	+14.3%
Sep 96	-10.0%	-25.4%	+8.7%
Oct 96	-4.9%	-23.2%	+12.2%
Nov 96	-18.0%	-30.0%	+11.2%
Dec 96	-12.5%	-22.3%	+20.3%
Jan 97	-4.9%	-16.9%	+16.4%
Feb 97	-4.1%	-15.4%	+32.3%
Mar 97	-5.6%	-14.1%	+15.4%
Apr 97	-14.3%	-26.7%	+8.6%
May 97	-15.8%	-29.2%	+3.0%
Sum	-9.5%	-22.1%	+14.4%

Table 4.3: Results for Makespan and 128 Nodes

	$PFCFS_1$	$PFCFS_2$	$PFCFS_3$
Jul 96	-7.1%	-19.7%	+13.8%
Aug 96	-6.1%	-20.4%	+13.2%
Sep 96	-10.9%	-25.5%	+6.5%
Oct 96	-3.3%	-20.9%	+11.5%
Nov 96	-21.0%	-33.4%	+5.7%
Dec 96	-15.0%	-25.6%	+20.7%
Jan 97	-4.6%	-13.4%	+12.1%
Feb 97	-5.8%	-18.3%	+25.0%
Mar 97	-5.3%	-14.2%	+16.7%
Apr 97	-11.8%	-24.5%	+11.8%
May 97	-14.0%	-28.4%	+2.7%
Sum	-9.6%	-22.2%	+12.3%

Table 4.4: Results for Completion Time and 128 Nodes

	$PFCFS_1$	$PFCFS_2$	$PFCFS_3$
Jul 96	-10.4%	-30.6%	+22.0%
Aug 96	-22.6%	-41.9%	+2.0%
Sep 96	-15.9%	-37.3%	+9.4%
Oct 96	-5.2%	-32.9%	+17.4%
Nov 96	-29.7%	-47.3%	+8.1%
Dec 96	-22.4%	-39.1%	+31.7%
Jan 97	+28.1%	+9.6%	+63.0%
Feb 97	-8.2%	-28.8%	+38.3%
Mar 97	-8.4%	-22.3%	+26.2%
Apr 97	-18.4%	-38.3%	+18.4%
May 97	-19.5%	-39.5%	+3.7%
Sum	-13.6%	-33.2%	+19.1%

Table 4.5: Results for Flow Time and 128 Nodes

Resources	128	256
Wide job spec. x	40%	45%
Max. preemptions n	1	1
Start delay δ	60 sec	60 sec
Gang length Δ	not applicable	

Table 4.6: Parameter Setting for the Results Shown in Figure 4.1

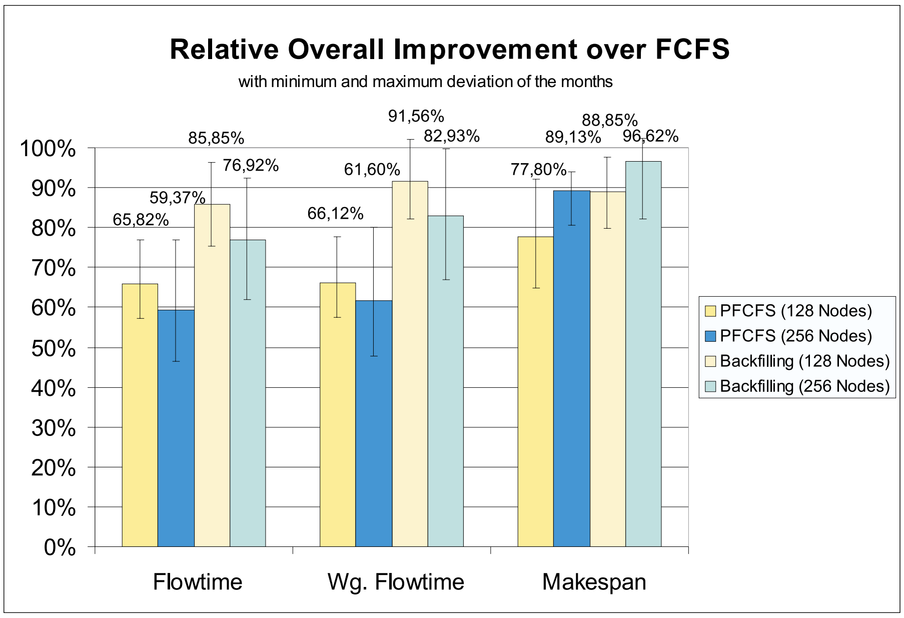

Figure 4.1: Comparison of PFCFS with FCFS for Different Criteria

preemption, but requires the user to provide a maximum execution time for each job.

However, it can be noticed that the effect on the flow or weighted flow time is more pronounced for a 256 node parallel computer while the smaller 128 node machine achieves a better result in terms of the makespan criteria. It can also be seen that in every month a noticeable gain can be achieved with this set of parameters although the actual gains depend on the workload. Note that in the selected setting there is no fixed time period between two context switches. With $n = 1$ a wide job will interrupt some small jobs and then run to completion. A similar approach was also subject by Chiang et al. [9] in the context of dynamic jobs and application characteristics.

Also, there is only very little difference between the weighted flow time and the flow time results. This can be attributed to the large amount of short running jobs which do not require a large number of nodes. The stronger emphasis of larger highly parallel jobs in weighted flow time criterion does not have a significant impact on the evaluation of the strategy. However, this can be expected to change if the number of those jobs would increase.

Next, we examine the result for different limits on the number of preemptions. The results in Figure 4.2 and Figure 4.3 show that there are always improvements for odd values of n. Note that an odd n gives preference to the execution of the wide job once the maximum number of preemptions is reached. However, the gains change only little if the number of preemptions is increased. This indicates that the desired improvement of the scheduling costs can already be obtained by using a simple preemption for a wide job. On the other hand, even values of n result in significantly

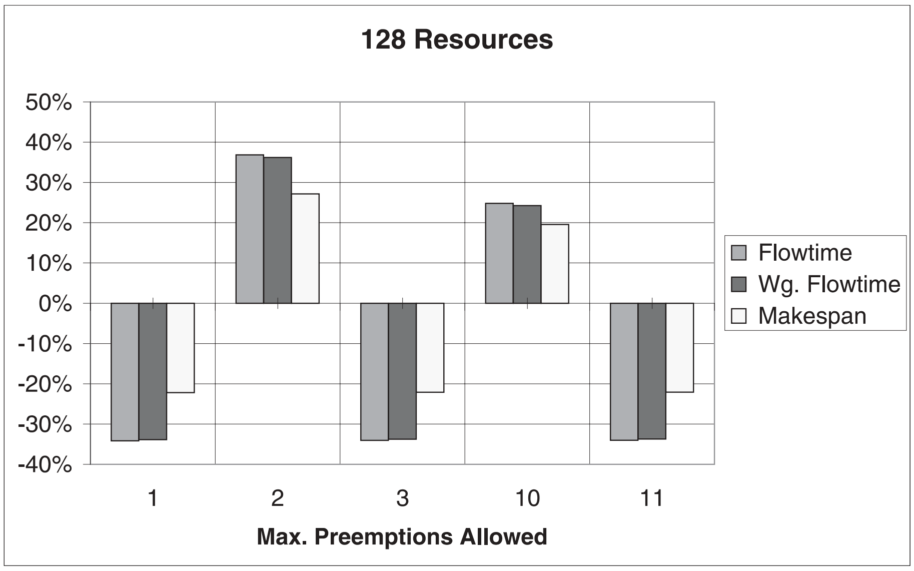

Figure 4.2: Results for Different Limits on the Number of Preemptions for the 128 Node Machine

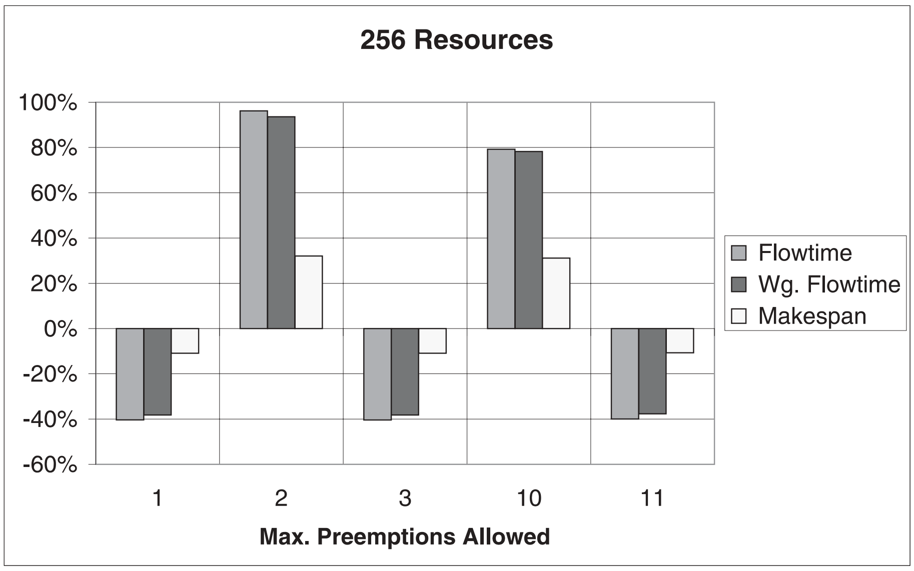

Figure 4.3: Results for Different Limits on the Number of Preemptions for the 256 Node Machine

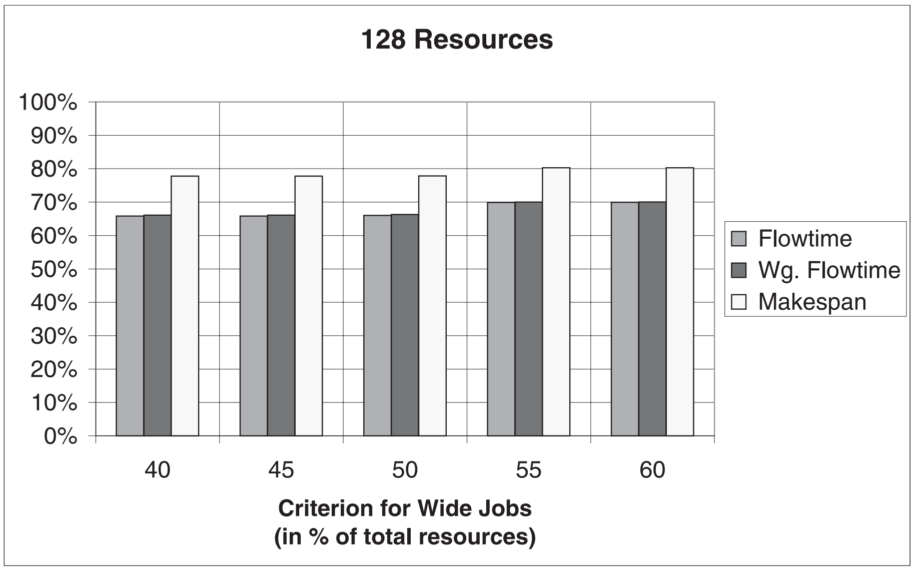

Figure 4.4: Results for Different Characterizations of Wide Jobs for the 128 Node Machine

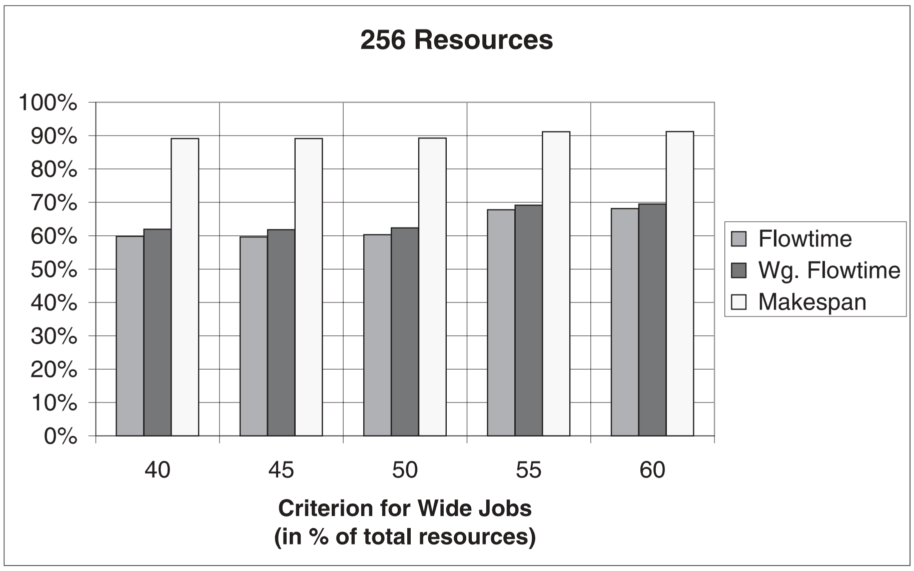

Figure 4.5: Results for Different Characterizations of Wide Jobs for the 256 Node Machine

worse schedules in comparison to FCFS. This is due to the fact that the completion time of the wide job may actually be further increased over the FCFS schedule as its node allocation is fixed at the start time of the job and no job migration is allowed. This may then lead to a larger degree of fragmentation.

The schedule quality is not as sensitive as on the other parameters. Our theoretical studies [61] indicate that x=50% would be the best choice to separate wide jobs from small ones. Figure 4.4 and Figure 4.5 show that the schedule cost increases for all three criteria if x is selected to be larger than 50%. For the presented workloads a further slight improvement can be obtained by selected values for x which are less than 50%.

Δ is irrelevant if n is selected to be 1. For all other odd values of n the schedule cost varies only little with Δ, as seen in Figures 4.2 and 4.3. There, $\Delta = 120$ sec is frequently the best choice. Additional simulations showed for even n that the results improve with a larger Δ as the chances to reach the preemption limit become smaller.

As the choice of Δ has little influence on the schedule quality, it is sufficient to just consider the cases δ= 1 sec and δ=60 sec. From theory we would expect a better makespan for δ= 1 sec as fragmentation will be reduced if wide jobs are executed as early as possible. On the other hand, δ= 60 sec will permit some short running small jobs to complete before they are interrupted by a wide job. However, in our simulation, switching from 1 sec to 60 sec changed the schedule costs by less than 0.01%. Therefore, δ can also be ignored.

4.3 Comparison to Theoretical Results

The presented evaluation by simulation provides information on the schedule quality that is actually achievable for certain workloads. Our workloads were derived from real traces and used to resemble a real application scenario with the mentioned modelling. We noticed that the schedule performance is very sensitive to the preemption strategy. While significant improvements over FCFS are possible, FCFS may also outperform some preemptive methods. The evaluation showed that the results are sensitive to the workload and can be fine tuned by several heuristic parameters. Nevertheless, the results showed also that the presented PFCFS algorithm can consistently outperform the FCFS strategy if the scheduling parameters are selected correctly.

The results from the theoretical analysis guarantee a certain degree of fairness and competitive ratios under all circumstances for our algorithm. Nevertheless, the theoretical analysis did not give us information on the actual performance of the algorithm for real workloads. The theoretical analysis provides a competitive ratio of > 3.562 for the completion time which suggests that a large percentage of resources would be idle in the worst-case. However, our algorithms showed efficient results for the workload simulations in terms of completion and flow time as well as the makespan in comparison to the FCFS algorithm. Even backfilling has been outperformed in certain scenarios. This leads to the assumption that from a performance point of view the presented algorithm is actually recommendable for real installations. This example shows the importance of a experimental simulation in the process of a scheduling system design. Note, that a parameter-optimization and fine-tuning of the algorithm can be performed by simulation studies. Such studies and examinations on a production system are usually cost-inefficient and impracticable. Note, that the theoretical analysis of the impact of certain parameter selections, as e.g. presented in our evaluation, would require quite some effort while the relevance on the real system is very limited. From the results for the PFCFS algorithms, we suggest that the parameters are determined on-line in an adaptive fashion based on former workload. While potentially beneficial for a real installation, the theoretical analysis may not provide additional practically relevant information for a scheduling system designer.

After these results we make the transition from the design and evaluation of scheduling strategies for single parallel machines to the grid scheduling problem in the next chapters. There, we use the same approach on the design process. The components of a grid are typically single parallel machines with local management instances, therefore we can use and adapt the above discussed models and strategies.

Chapter 5

Grid-Scheduling

In this work we have suggested a strategy and a process to design job scheduling systems. According to this concept the evaluation of the schedule quality takes an important part. We want to focus on the actual performance of grid scheduling strategies for real installation scenarios. Based on our previous results, we choose the experimental evaluation for this purpose. The theoretical analysis of such algorithms gives worst-case bounds and guarantees of certain properties like the introduced concept of fairness. For the schedule quality in a real system scenario these bounds are of minor interest. Based on the results and discussions in the previous chapters we will therefore focus on experimental evaluation by simulation.

However, all previous chapters dealt with analysis and results for scheduling systems on single parallel machines. As we want to develop scheduling strategies for grid computing, we have to extend the previous results to the grid environment.

Here, a computational grid (see [14, 39]) or metacomputer [63] consists of numerous independent computing resources which are linked by interconnection networks. The term *independent* emphasizes the fact that the participating resources are not controlled by a single entity and may be geographically distributed. While this definition for a grid does not include further details on the individual resources in this network, these resources are often high performance computing components like large parallel processors, networks with high speed and high bandwidth or huge databases. Only the component at the user access point may not necessarily belong to this category. However, the user of such a grid system may not be aware of the internal system structure but has the illusion of a single virtual machine. The grid management system may use several components of the system concurrently to solve a large problem.

The job scheduling for a single parallel computer significantly differs from scheduling for a metacomputer. As we have seen, the scheduler of a parallel machine usually arranges the submitted jobs in a certain order to achieve a high utilization. The task of scheduling for a metacomputer is more complex as many machines are involved with mostly local scheduling policies. The metacomputing scheduler forms a new level of scheduling which is implemented on top of the job schedulers. Also, it is likely that a large metacomputer may be subject to more frequent changes as individual resources may join or exit the grid at any time. Note that some users will utilize the ability to solve single large problems by combining many different resources. Altogether, this requires solutions for several challenges in hard- and software in several areas. One of these topics is the scheduling problem.

As one of its benefit, grid computing provides the user with access to locally unavailable resource types. Furthermore, there is the expectation that a larger number of resources are available. It is also expected that this will result in a reduction of the average job response time. Also the utilization of the grid computers and the job-throughput is likely to improve due to load-balancing effects between the participating systems.

However, parallel computing resources are usually not exclusively dedicated to grid computing. Furthermore, they are typically not owned and maintained by the same administrative instance. Research institutes are an example for such resource owners, as well as laboratories and universities. Without grid computing local users are usually only working on the local resources. The owners of those computing systems are interested in the impact of participating in a computational grid and whether such a participation will result in better service for the users by improving the job response time. Therefore, first we want to examine the practical benefit of collaboration between computing sites.

5.1 Multi-Site Applications

The usage of multi-site applications has been theoretically discussed for quite some time [4]. Multi-site computing is the execution of a job in parallel at different sites. This results in a larger number of totally available resources for a single job. The effect on the average job response time is yet to be determined as there are only few real multi-site applications. The lack of real multi-site applications may be the result of an absence of a common grid computing environment which is able to support the parallel allocation of resources at remote sites. In addition many users fear a significant adverse effect on the computation time due to the limitations in network bandwidth and latency over wide-area networks. The overhead depends on the communication requirements between the job parts of a particular application. As WAN networks become faster, the overhead may decrease over time. Therefore, we determine which amount of overhead will still result in an overall user benefit.

To evaluate the effect of multi-site applications in a grid environment, we want to examine the usage of multi-site jobs in addition to job sharing. To this end, discrete event simulations on the basis of workload traces have been executed again for sample configurations. The potential benefit is evaluated if a computing site participates in a computational grid. This evaluation is focused on the question whether sharing jobs between sites and/or using multi-site applications could provide advantages in mastering the existing workload.

5.2 Grid Site Model

We assume a computational grid consisting of independent computing sites with their local workloads. That means that each site has its own computing resources as well as a local user community that submits jobs to the local job scheduling system. In a typical single site scenario all jobs are only executed on the local resources.

The sites may combine their resources and share incoming job submissions in a grid computing environment. Here, jobs can be executed on local *and* remote machines. The computing resources are expected to be completely committed to grid usage. That is job submissions of all sites are redirected and distributed by a grid scheduler. This scheduler exclusively controls all grid resources. For a real world application this may be a difficult requirement to fulfill. There are other possible implementations where site-autonomy is maintained, while the results from our model can still be applied. Nevertheless, the scheduling process becomes more complex. Scalability to large scale grids with many resources is challenging as users and administrators have high expectations in terms of security, fault-tolerance and performance.

5.3 Grid Machine Model

We assume massive parallel processor systems (MPP) as the grid computing resources where each site has a single parallel machine that consists of several nodes. Each MPP machine coheres to our machine model from Section 2.1.1. A parallel job can be allocated to any subset of nodes of a machine. This model comes reasonably close to a grid environment consisting of real systems like IBM RS/6000 Scalable Parallel Computers, a Sun Enterprise 10000 or a HPC clusters.

For simplicity all machines and all nodes in this study are identical. The machines at the different sites only differ in the number of nodes. The existence of different resource types would additionally limit the number of suitable machines for a job. In a real implementation a preselection step is part of the grid scheduling process and is normally executed before the actual scheduling takes place. After the preselection phase the scheduler can ideally choose from several resources that are all suitable for the job request. In this study we neglect this preselection step and focus on the remaining scheduling task. Therefore, in the following it is assumed that all resources are of the same type and all jobs can be executed on all nodes.

Further, we keep the model for the execution of a job from the previous chapters as introduced in Section 2.1.2. That is, the jobs are not preempted nor is time-sharing used. Thus, once started a job runs until completion. Furthermore, we do not consider the case that a job exceeds its allocated time. After its submission, a job requests a fixed number of resources that are necessary for starting the job. This number is not changed during the execution of the job. That is, jobs are neither moldable nor malleable [19, 28].

5.4 Grid Job Model

We model a grid scenario with independent sites with workload generated by local user groups. Jobs are submitted by these independent users at the local sites. This produces an incoming stream of jobs over time. Thus, we still deal with an on-line scheduling problem without any knowledge on future job submissions. Furthermore, we restrict our simulations on batch jobs, as we did in our discussion on single parallel machines.

In our model the scheduling system allocates resources for the jobs and determines its starting time. The job is executed without further user interaction. In a real implementation the job data must be transferred to the remote site before the execution. This transport of data requires additional time. This effect can often be hidden by prefetching before the execution or returning output data after the execution time (postfetching). In this case the resulting overhead is not necessarily part of the scheduling process. It is the task of a data management to optimize the transport of data. Note, that for some applications, e.g. high-energy physics many users work on the same input data sets. The intelligent management of these data files can reduce the network overhead. In our grid computing scenario, a job can be transmitted to a remote site without any additional overhead. Data management of any files is neglected in this study. Nevertheless, in a real implementation an intelligent data management would be part of the scheduling system and should be taken into consideration in future work.

In a grid environment we now assume the ability of jobs to run in multi-site mode. That means a job can run in parallel on a node set distributed over different sites. This allows the execution of large jobs that require more nodes than available on a single machine in the grid environment. The impact of bandwidth and latency has to be considered as wide-area networks are involved. In the simulations, we will address this subject in Section 5.6.3 by increasing the job length if multi-site execution is applied to a job.

5.5 Grid Scheduling System

As we have seen in previous parts of this work, first-come-first-serve (FCFS) strategies can result in poor quality if more jobs with large node requirements are submitted. The results already showed that backfill is usually preferable against FCFS in our scenarios. To this end, we assume the backfill strategy for the local machines as a reference implementation, see Section 2.5. It requires knowledge of the expected job execution time and can be applied to any greedy list schedule.

In our scenario for grid computing, the scheduling task is delegated to a grid scheduler. The local scheduler is only responsible for starting the jobs after the allocation by the grid scheduler. Note, that we use a central grid scheduler for this study. In a real implementation the architecture of the grid scheduler will be different as single central instances are usually associated with drawbacks in performance, fail-safety or acceptance of resource users and owners. Nevertheless, distributed architectures can be designed in such a way that they act logically similar to a central grid scheduler as presented in this study.

5.6 Conventional Grid Algorithms

Three scenarios are compared in our study for a small grid environment:

1. *local job* processing as a reference scenario with the local execution;
2. *job-sharing* between cooperating computing sites;
3. *multi-site* computing with jobs split over different sites.

We briefly illustrate these scenarios in the following.

5.6.1 Local Job Processing

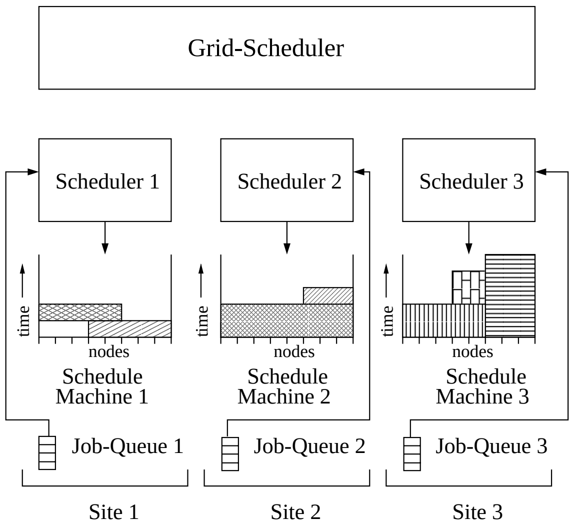

Figure 5.1: Sites executing all jobs locally

This scenario refers to the common situation where the local computing resources at a site are dedicated only to its local users (see Figure 5.1). A local workload is generated at each site. This workload is not shared with other sites and the local submission queues are independent from each others. In our examination the forementioned backfilling scheduler is applied. In this work we just present the results for EASY backfill algorithm as its performance proved to be more effective in our simulations than conservative backfilling. The EASY backfill scheduler is the original method presented by Lifka [51]. It has been implemented for several IBM SP2 installations [67].

5.6.2 Job Sharing

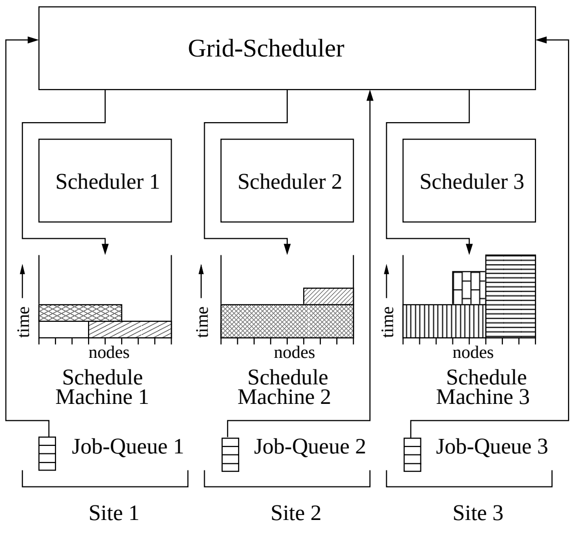

Figure 5.2: Sites sharing jobs and resources

In the *job-sharing* scenario all jobs submitted at any site are delegated to the grid scheduler as seen in Figure 5.2. In our examination the scheduling algorithms in grid computing consist of two steps. In the first step the machine is selected and in the second step the allocation in time for this machine takes place.

Machine Selection: There are several methods possible for selecting machines. Other simulation results (presented in [41]) showed good results for a selection strategy called *BestFit*. Here, the machine is selected on which the job leaves the least number of free resources if started.

Scheduling Algorithm: Here, the backfilling strategy is applied for the single machines as well. This algorithm has shown best results in previous studies.

5.6.3 Multi-Site Computing

This scenario is similar to *job sharing*: the local schedulers forward all jobs to a grid scheduler. This time, the grid scheduler does not only select a single site for job execution but can also split jobs to be executed crossing site boundaries (see Figure 5.3).

There are several strategies possible for multi-site scheduling. For this work we use a scheduler which first tries to find a site that has enough free resources for starting the job. If such a machine is not available, the scheduler tries to allocate the jobs to resources from different sites. To this end the sites are sorted in the descending order of free resources and allocating the free resources in this order for a multi-site

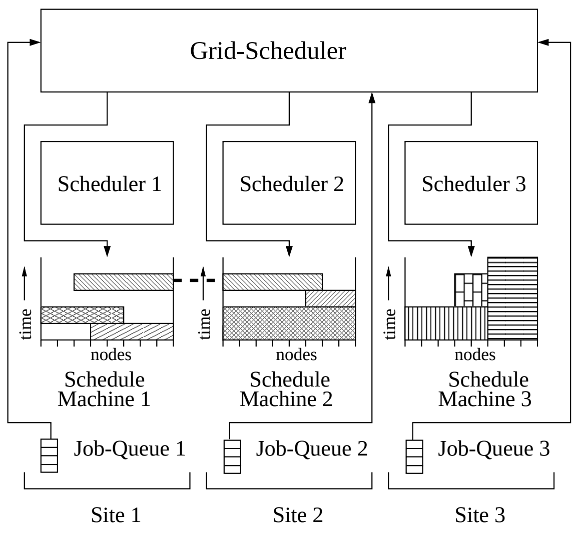

Figure 5.3: Support for multi-site execution of jobs

job. In this case the number of combined sites is minimized. If there are not enough free resources available for a job, it is queued and normal backfilling is applied.

Spawning job parts over different sites usually produces an additional overhead. This overhead is due to the communication over slow networks (e.g. a WAN). This overhead applies to the time necessary to transfer job data to and from a resource at job start and end. Additionally, this overhead applies to the process communication during run-time. Consequently, the overall execution time of the job will increase depending on its particular communication pattern. For those jobs with limited communication demand there is only a small impact. Note, without the introduction of any penalty for multi-site execution, the grid would behave like a single large computer. Hence, multi-site scheduling will ideally outperform all other scheduling strategies. In this study we examine the effect of multi-site processing on the schedule quality under the influence of a communication overhead. To this end, we model the influence of the overhead by extending the required execution time r_i to r_i^* for a job i that runs on multiple sites by a constant factor:
$r_i^* = (1 + p) \cdot r_i$ with p = 0 .. 40 % in steps of 5%.
The overhead in a real system scenario highly depend on the job communication pattern and the network configuration between the sites. In our model we do not regard these effects, as for example the number of sites involved for a multi-site jobs. All mentioned overhead contributions are summarized in the extension of the job run-time.

5.7 Simulation Results on Multi-Site Computing

For the evaluation of the different structures and algorithms discrete event simulations have been performed, see also [15, 18]. Several machine configurations have been examined for the forementioned algorithms.

5.7.1 Machine Configurations

All configurations use a total of 512 resources. Those resources are partitioned in the various machine configurations as shown in Table 5.1.

identifier	configuration	max. size	sum
m64	$4 \cdot 64 + 6 \cdot 32 + 8 \cdot 8$	64	512
m64-8	$8 \cdot 64$	64	512
m128	$4 \cdot 128$	128	512
m256	$2 \cdot 256$	256	512
m256-5	$1 \cdot 256 + 4 \cdot 64$	256	512
m384	$1 \cdot 384 + 1 \cdot 64 + 4 \cdot 16$	384	512
m512	$1 \cdot 512$	512	512

Table 5.1: Resource Configurations

The configurations *m64-8*, *m128* and *m256* represent several sites with equal machines. They are balanced as there is an equal number of resources at each machine. They differ in the number of machines and in the number of resources at each machine. The configurations *m384* and *m256-5* are examples of a large computing center with several client sites. In the latter configuration, the biggest machine is smaller than the biggest machine within the configuration *m384* and all other machines have an equal size in contrast to *m384*. The configuration *m64* is a cluster of several sites with smaller machines.

Finally, a reference configuration *m512* consists of a single site with one large machine. In this case no grid computing is used and a single scheduler can control the whole machine without any need to split jobs.

5.7.2 Workload Model

Unfortunately, no real workload is currently available for grid computing. For our evaluation we derived a suitable workload from real machine traces. To this end, we use the CTC workload which has already been used for the analysis in the previous chapters, see Section 2.8.3.

In order to use these traces for our study it is necessary to modify the traces to simulate submissions at independent sites with local users. To this end, the jobs from the real traces have been assigned in a round-robin fashion to the different sites. However, it is typical for many known workloads to favor jobs requiring a power of 2 number of nodes. The CTC workload shows the same characteristic, see Figure 5.4. The modelling of configurations with smaller machines would put these machines into disadvantage if the number of nodes is not a power of 2. To this end, our configurations consist of 512 nodes altogether. Nevertheless, the traces provide enough workload to keep a sufficient backlog on all systems (see results from [41]). The backlog is the workload that is queued at any time instant if there are not enough free resources to start the jobs. A sufficient backlog is important as a small or even no backlog indicates that the system is not fully utilized. In this case there is not enough workload available to keep the machines working. Many schedulers, e.g. the mentioned backfilling strategy, depend on the availability of enough jobs for its backfilling in order to utilize idle resources. Therefore, a small or no backlog usually leads to bad scheduling quality and unrealistic results.

As we already discussed in our previous evaluations in this work, the quality of a scheduler is highly dependent on the workload. We again assume a network of homogeneous machines that comply with our previous models of an IBM RS/6000 SP and the CTC job traces for such a machine. To minimize the risk that singular and abnormal workload characteristic affect the validity of our results, the simulations have been done for 4 workload sets. This allows to examine deviation for different workloads. Nevertheless, all used workloads are derived from the actual user group from the same real installation. As proposed in our presented scheduling design process, we assume a grid scheduling scenario where the user workload is similar as in the CTC workload. As we already explained, the schedule quality is usually highly depend on the workload. Additional evaluations have to be done for a specific other workload to select and adapt an algorithm.

Therefore, the following workload sets have been used:

- 3 extracts of the original CTC traces.
- A synthetic probabilistic generated workload on the basis of the CTC traces.

The synthetic workload is very similar to the CTC data set from Section 2.8.3. It has been generated to prevent that potential singular effects, e.g. a down-time of the real system, in real traces affect the accuracy of the result. Also the 3 extracts of real traces are examined to get information on the consistency of the results for the CTC workload. Each workload set consists of 10000 jobs which corresponds in real time to a period of more than three months.

A problem of such simulations is the handling of wide jobs which are contained in the original workload traces. The widest job in the CTC traces for example requests 336 processing nodes. On one hand these jobs can be used in simulations to examine the benefit of multi-site applications. Here jobs can be split over different sites to get more resources than available at a single site. On the other hand, some of these jobs cannot be started in simulations of scenarios with only local execution or job sharing.

To permit a valid comparison of the simulation results, no job must be neglected. Therefore we assume that the corresponding workloads of wide jobs are still generated at single sites. The wide jobs are split up into several parts of the local machine size to allow their execution. Accordingly, the job size is limited by the size of the largest machine in the *job-sharing* scenario. Here, users can submit jobs that are wider than the local machine size. We also use a modification were all jobs are split up into several parts with maximum 64 nodes to allow their execution on all examined configurations. Every configuration has a machine that consists of at least 64 nodes.

To allow the comparison of different scenarios for job sharing the following modifications have been applied to each forementioned workload.

Workload Modification:

1. Wide Jobs are split in parts of local machine size,
2. Wide Jobs are split in parts of largest machine size in the configuration,
3. Wide jobs are split in parts of 64 nodes,
4. Wide jobs are unchanged.

The workloads with modification *1* and *3* were executed in all 3 scenarios. The workloads with modification *2* were simulated for scenario *job-sharing* and *multi-site* while modification *4* was only used for the *multi-site* scenario. Note, that all of these modifications do not alter the overall workload. In our model, we assume that a certain amount of workload exists at a local site. With our modifications larger jobs are split up, but are still generated locally at the corresponding site. Depending on the scenario a user may submit jobs larger than the local machine. The simulations allow the examination of the impact caused by wider multi-site jobs on the schedule.

The examined workloads are summarized in Table 5.2 and an identifier is introduced for each workload.

5.7.3 Results

Job-Sharing

The simulation results show that job-sharing provides significant improvement over local job execution for the user. We use the average weighted response time as the measure in this study. Note that the weight is selected according to the definition in Section 3.3 which prevents any prioritization of small over wider jobs in terms of the average weighted response time if no resources are left idle [61].

The usage of job-sharing improved the average weighted response time in all examined machine configuration and in all workloads. In the *m128* configuration for example the improvement is over 50% (Figure 5.5). The results are similar in the other simulations. Note, that the results for the reference scenario in which jobs stay local on its site of origin depend on the modelling of the local workload. The EASY backfilling strategy is used in both scenarios, single-site execution and job-sharing. In contrast to job-sharing single-site execution is restricted to keep the workload

identifier	description
10_20k_org	An extract of the original CTC traces from job 10000 to 20000.
10_20k_max64	The workload 10_20k_org split into jobs with at most 64 processors.
30_40k_org	An extract of the original CTC traces from job 10000 to 40000.
30_40k_max64	The workload 30_40k_org split into jobs with at most 64 processors.
60_70k_org	An extract of the original CTC traces from job 60000 to 70000.
60_70k_max64	The workload 60_70k_org split into jobs with at most 64 processors.
syn_org	The synthetically generated workload derived from the CTC workload traces.
syn_max64	The workload syn_org split into jobs with at most 64 processors.

Table 5.2: The used workloads

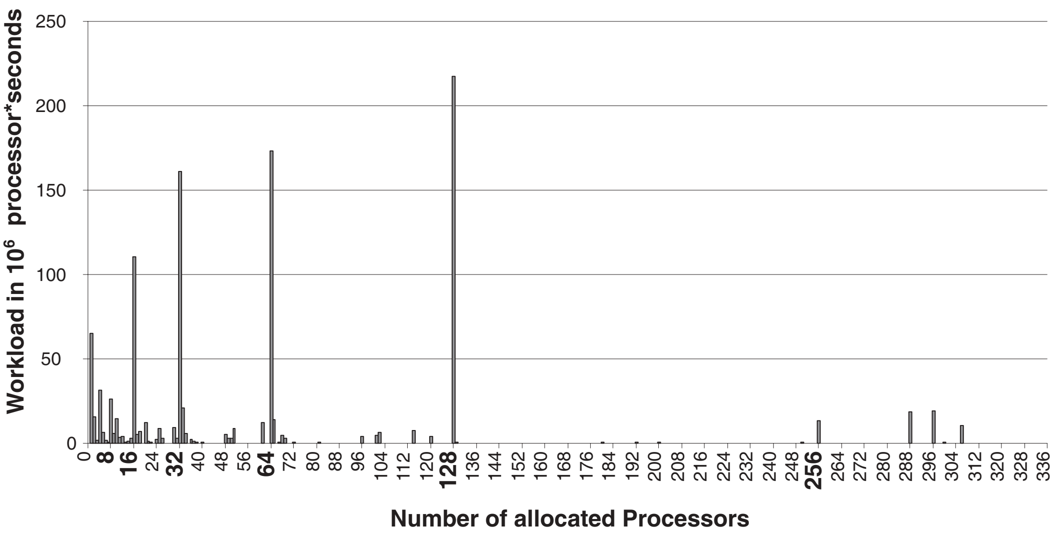

Figure 5.4: Workload distribution for the syn_org workload

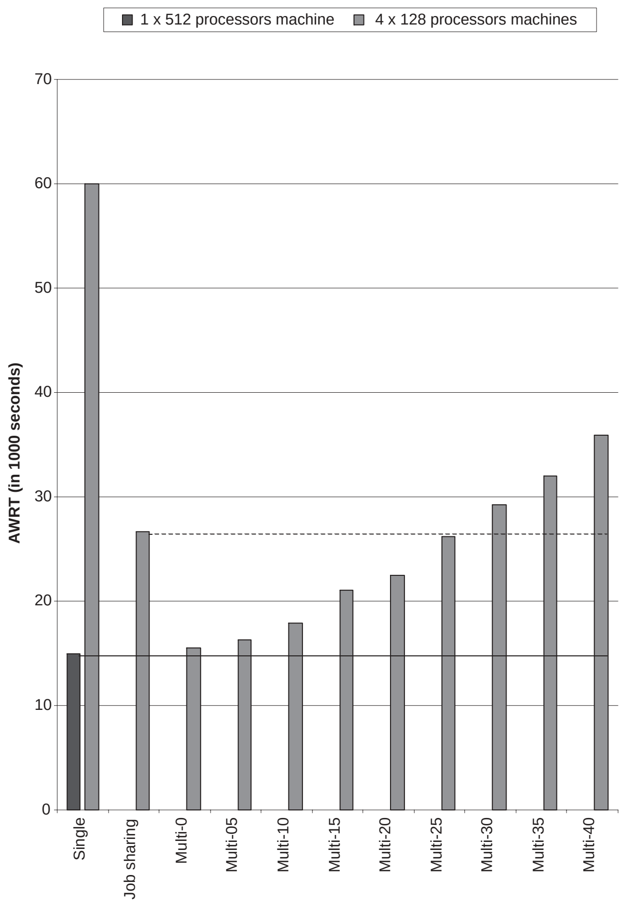

Figure 5.5: Average Weighted Response Time for the *m128* configuration and workload *ctcsyn* with modification *2*

locally. No job is transferred to a remote site. As mentioned before large jobs that are wider than the local machine have been split up into smaller jobs which are sequentially executed on the local system. This leads to an increase of the AWRT as the workload of these wide jobs are still generated and submitted at the same time as the original wide job, but the sequential execution leads to a delay of job parts. Job-sharing on the other hand allows the transfer of jobs to remote machines.

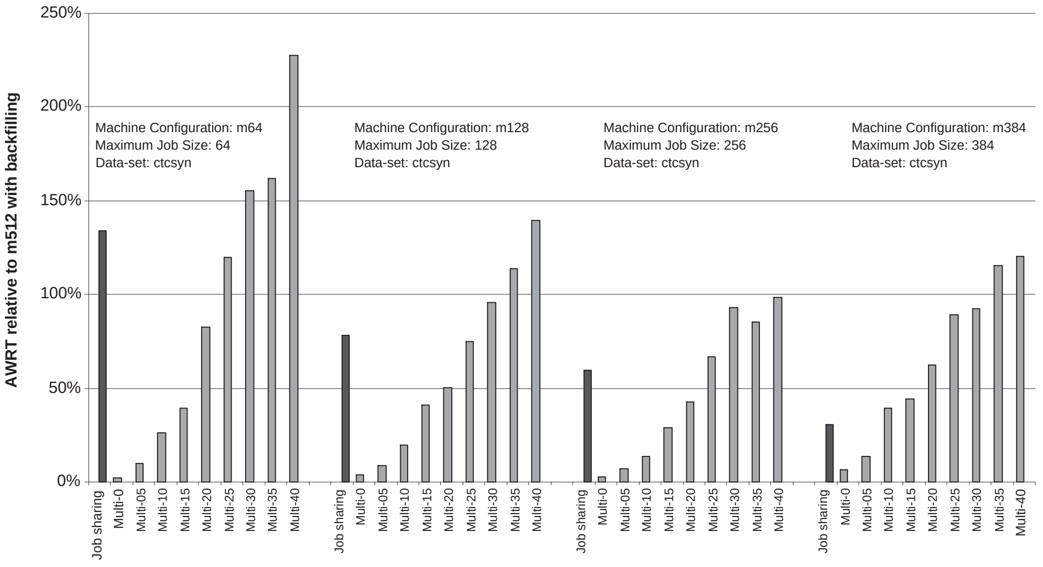

Figure 5.6: Results for different resource configurations compared to configuration *m512* with backfilling (equals 0%)

Multi-Site Computing

Next, we examine the influence of using multi-site execution. The results show that further improvement on the AWRT can be achieved in comparison to job-sharing. As a reference the result for a single machine with 512 nodes is used (Figure 5.5). The result of this *m512* configuration gives the lower bound for the backfilling algorithm. In this configuration no machine partitioning had to be taken into account contrary to any other configuration. As expected, the average weighted response time without overhead for multi-site is near to the *m512* result, see Figure 5.5. In this case splitting a job for multi-site execution causes no penalty.

Moreover, multi-site execution is beneficial compared to job-sharing even for an overhead on execution time of about 25% (Figure 5.5).

Similar results are achieved for other configurations. As an example Figure 5.6 shows the improvement for the workload with the modification to limit the job width to the maximum machine size. Note, that the configurations with equal sized machines show better results than for the *m384* or *m64* configurations. Here, on the equal-sized machines, a larger overhead is tolerable for multi-site to still improve the AWRT in comparison to job sharing.

The mentioned improvements can also be verified in different workload scenarios. Moreover, the previously mentioned results, as given in Figure 5.5, showed the least effective improvements in comparison to other examined workloads, see Figure 5.7. The average weighted response time in other configurations delivered even better results. The tolerable overhead on multi-site executed jobs can even be larger.

All mentioned results are achieved for the workload in which all original wide jobs

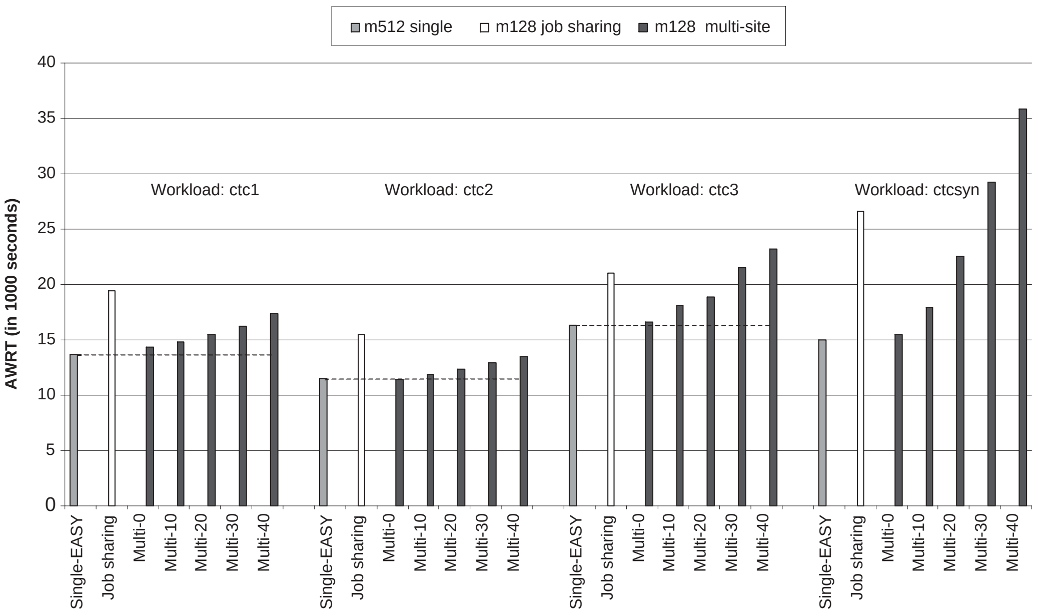

Figure 5.7: Results for different workloads

are split into job parts with the size of the machine on which it has been generated. In our example as shown in Figure 5.5 jobs with node requirements larger than 128 were split up into jobs requesting 128 or less nodes. This allowed us to compare the job-sharing and multi-site scenario with the single-site scenario. We now want to examine the influence on the results if wide jobs are submitted locally that require the execution on remote resource by multi-site. To this end, we used the original wide jobs from workloads without the modification in our simulations. Note, that this simulation cannot be computed for the job-sharing scenario as these wide jobs can only be executed in a multi-site scenario. The results show that submitting these wide jobs does not increase the average weighted response time significantly (10-20%). An example is presented in Figure 5.8. There is still an advantage of multi-site over job-sharing, although these wide jobs are actually more difficult to schedule. The allocation of such a wide job requires synchronously free resources at different sites. This may require a longer delay for the wide job and leaves less possibilities for allocating other jobs in comparison to the previous workloads where wide jobs are split and the different parts do not have to be executed concurrently. Nevertheless, the addition of wide jobs in multi-site did not significantly increase the AWRT.

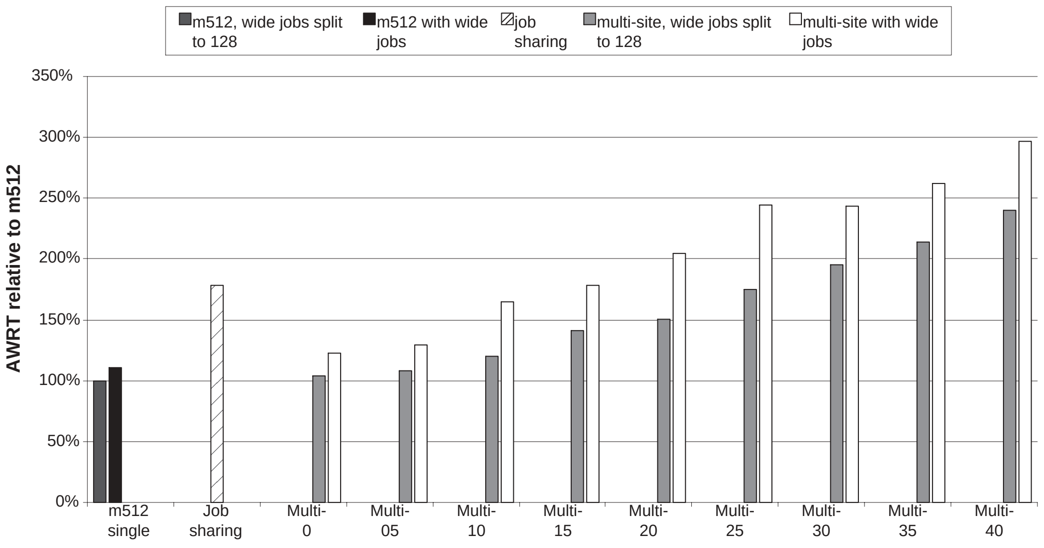

Figure 5.8: Comparing workloads with original jobs split to 128 parts against keeping wide jobs

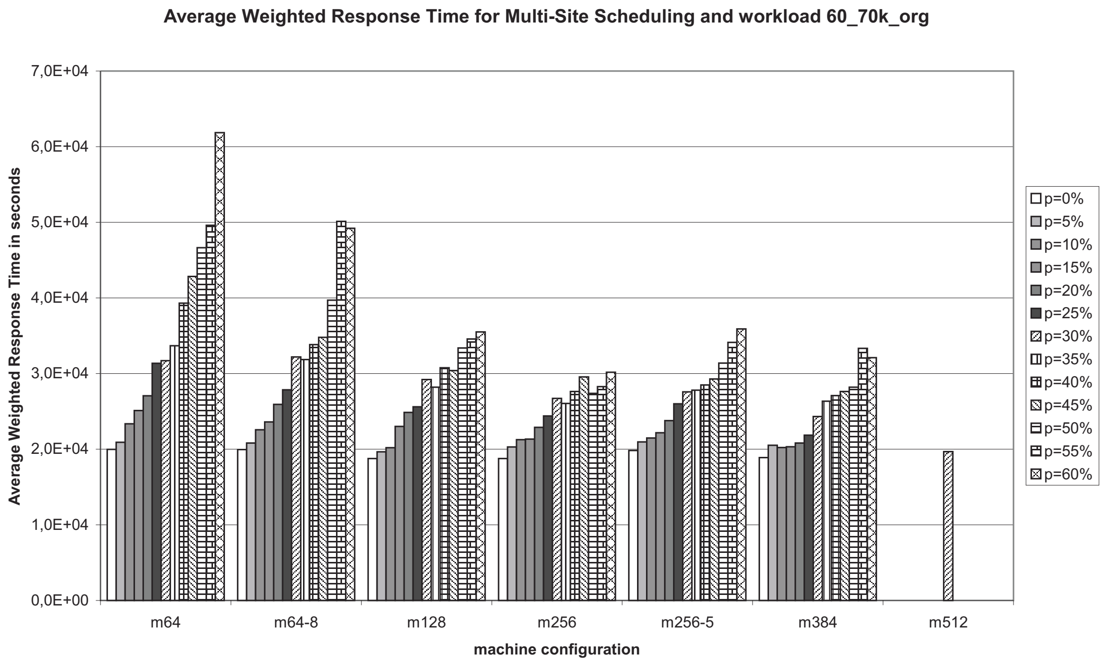

Figure 5.9: The average weighted response time in seconds for workload 60_70k_org and all machine configurations and Multi-Site Scheduling

Machine configurations

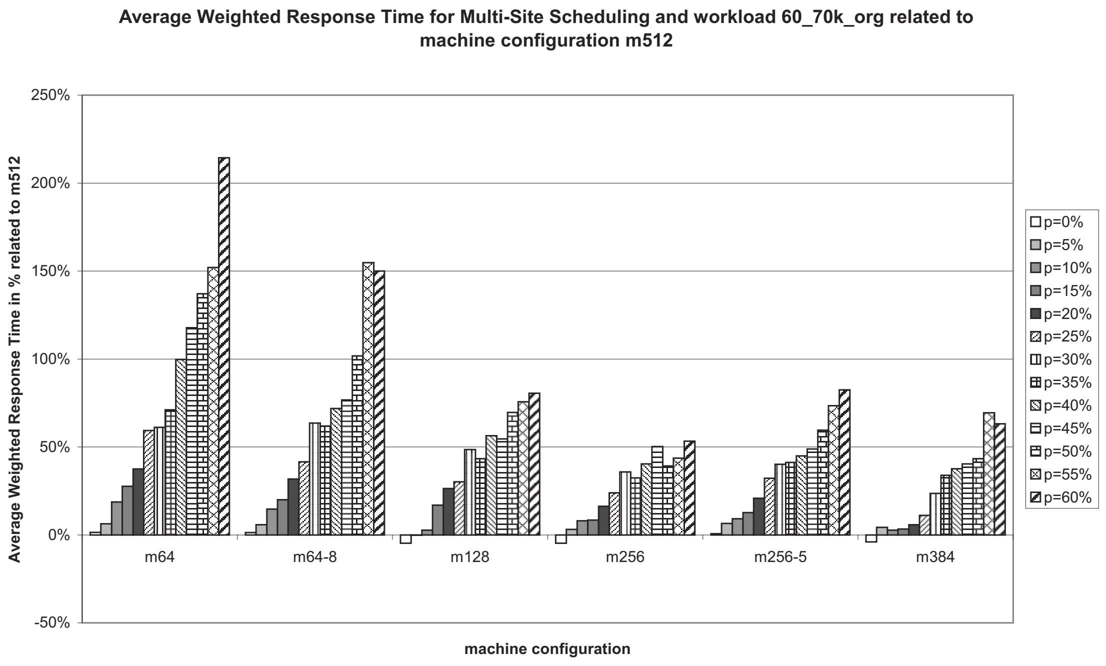

Figure 5.10: The average weighted response time for workload 60_70k_org and all machine configurations relative to machine configuration m512 for Multi-Site Scheduling

Next, different machine configurations have been examined to determine if the results can be verified on different grid configurations and in which way they are sensitive on the machine sizes. The simulation results show that configurations with equal sized machines provide significant better scheduling results than machine configurations that are not balanced, see Figures 5.9 and 5.10.

Overall, machine configurations with less but smaller machines produce better scheduling results under the precondition that each configuration has the same amount of resources in the sum. As an example for the scheduling quality the average weighted response time for the workload 60_70k_org and all resource configurations is given in Figure 5.9. The other workloads show a similar behavior. The AWRT for the machine configuration m512 is constant for all parameters, due to the fact that no multi-site scheduling is applied in this configuration. In the example in Figure 5.9 the AWRT decreases from m64-8 over m128 to m256. These three machine configurations have equal sized machines but the sizes of the machines increase between the configurations, as previously described in Section 5.7.1 and Table 5.1. The AWRT decreases from machine configuration m64 to m64-8, because the configuration m64-8 consists of more larger machines than m64. As more jobs are executed in multi-site mode in configurations with more smaller machines, an increase of the communication overhead (p) has a higher impact on the AWRT. The same effect can be

file	10_20k_org $[jobs]\hat{=}[10^{-2}\%]$	30_40k_org $[jobs]\hat{=}[10^{-2}\%]$	60_70k_org $[jobs]\hat{=}[10^{-2}\%]$	syn_org $[jobs]\hat{=}[10^{-2}\%]$
p=0%	539	431	840	774
p=5%	543	436	850	769
p=10%	582	448	928	827
p=15%	572	490	917	906
p=20%	583	546	946	946
p=25%	601	567	951	1042
p=30%	622	521	979	1026
p=35%	637	523	1036	1063
p=40%	647	534	1106	1012
p=45%	673	597	1008	1181
p=50%	755	579	1029	1188
p=55%	748	578	1086	1233
p=60%	746	638	1114	1177

Table 5.3: Number of Multi-Site Jobs for different workloads and different parameters using machine configuration m128

observed between m256 and m256-5. Here the configuration m256-5 is not balanced and contains some smaller machines, which turns out to be a disadvantage. The results are very sensitive to the characteristic of the workload as can be seen in the comparison between m384 and m256. In this example m384 outperforms m256 for a communication overhead up to 45%. Note, that the AWRTs of the configurations m64 and m64-8 are almost similar for values of the multi-site overhead parameter between 0% and 35%. If the overhead exceeds 35% the AWRT of configuration m64 increases dramatically and is at least 25% bigger than the AWRT for configuration m64-8. The difference of the AWRT for parameters under 35% is not significant. Similar results can be found by comparing configuration m256 and m256-5. The AWRT increases for overheads over 45% significantly.

The increase of the AWRT results from two effects. First, the overall workload increases as all multi-site jobs have a longer execution time due to the overhead. Second, we can make the observation that the number of multi-site jobs increases with additional overhead. Table 5.3 shows the number of multi-site jobs for machine configuration m128 for different workloads and different multi-site parameters. The effect can be observed for all workloads. This process is not monotone, but the difference between the number of multi-site jobs for p=0% and p=60% is always at least 30%.

This behavior results from the scheduling policy to schedule all jobs as soon as possible after submission. Because of an increased execution time of all multi-site jobs the number of free time slots within the schedule decreases and the probability of free time slots within one machine decreases as well. Therefore jobs are started as soon as possible if free time slots from different machines are combined.

Table 5.4 indicates that the increasing number of multi-site jobs corresponds to an

file	10_20k_org	30_40k_org	60_70k_org	syn_org
p=0%	22,19%	23,97%	34,01%	40,75%
p=5%	22,87%	25,09%	36,36%	41,21%
p=10%	24,71%	27,21%	37,28%	46,85%
p=15%	25,30%	28,24%	38,97%	47,95%
p=20%	26,05%	31,36%	40,43%	46,91%
p=25%	29,40%	31,93%	41,02%	52,81%
p=30%	29,11%	30,84%	44,53%	53,62%
p=35%	30,24%	31,01%	44,38%	54,01%
p=40%	31,21%	32,82%	48,10%	56,01%
p=45%	33,22%	37,20%	45,87%	59,77%
p=50%	37,15%	34,18%	46,25%	59,99%
p=55%	37,87%	36,05%	49,81%	62,72%
p=60%	41,46%	39,17%	52,38%	60,46%

Table 5.4: The amount of work by Multi-Site jobs related to the amount work by the whole workload for different workloads and multi-site parameters using machine configuration m128

increased ratio of the amount of work, the processor time product, caused by the multi-site jobs and the whole workload. This multi-site part increase at least 50% (up to 90% in our example) from jobs without overhead to jobs with 60% execution time increase in multi-site computing.

Tables 5.3 and 5.4 also indicate that jobs running in multi-site mode are in majority bigger jobs. This can be concluded because 5% to 10% of all jobs are multi-site jobs while they are responsible for about 20% to 40% of the whole amount of workload depending on the used job trace. This effect even increases for a higher multi-site overhead and is responsible for the mentioned results.

The AWRT for the machine configurations m128, m256 and m384 is smaller in comparison to m512 for the multi-site parameter p=0%, as it can be seen in Figure 5.10. This effect results from the algorithm for multi-site scheduling as described in Section 5.1. A job that is considered for multi-site execution can be started earlier as previously submitted but not yet started jobs. This strategy allows that backfilling is outperformed in some of our simulations. The results for for different workloads while using multi-site scheduling with p=50% lead to the assumption that the explained effects apply in general, see Figure 5.11.

After examining the influence of the machine configuration on the multi-site scheduling we address the influence on the job sharing scenario. Here, we see the jobs with limited resource demand result only in a minor drawback for configurations with smaller machines, see Figure 5.13. In this example only workloads with the modification to split jobs to a maximum of 64 processors are used to allow the comparison between the scenarios. The increase of the average weighted response time stays generally within 20% for all partitioned configurations except m64 compared to a single large machine with 512 nodes. The workload syn_max64 as an exception shows that

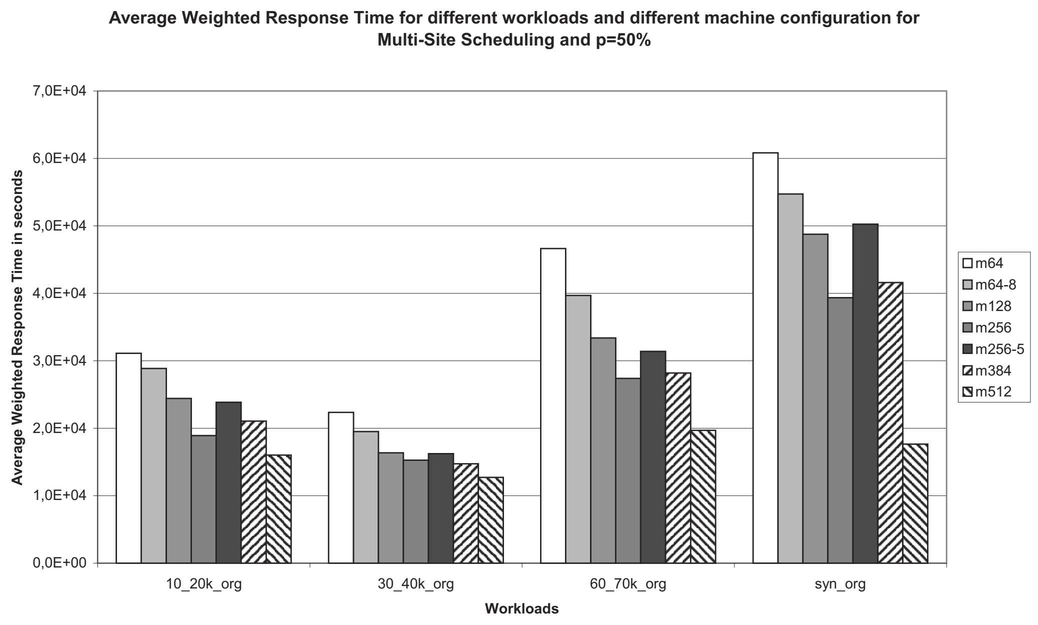

Figure 5.11: The average weighted response time in seconds for all workloads and all machine configurations for Multi-Site Scheduling and p=50%

the results are highly dependent on the workload characteristics. Similar results can be found for other workloads where the job width is limited to other sizes. This leads to the observation that the AWRT of the schedules suffers if the job processors requirement is large (e.g. 50%) in comparison to the largest available machines.

The results for job sharing correspond to the statements made for multi-site scheduling. The AWRT for all workloads and machine configuration m64 is higher than the AWRT for machine configuration m64-8, see Figure 5.13. This indicates that balanced systems with bigger machines is advantageous to unbalanced system with some smaller machines. In our example the m64-8 configurations produces to upto 25% better AWRT results in comparison to m64. The comparison between the configurations m64-8, m128 and m256 shows that the use of bigger machines produces favorably better scheduling results. The actual improvement depends on the workload. The job-sharing scheduling for the m256 and m256-5 configurations produce the same results as for multi-site scheduling. Again, the use of a system with two big machines is preferable to the use of a system with one big and several smaller machines. The configurations m384 and m256 provide similar results without a clear advantage to one of them.

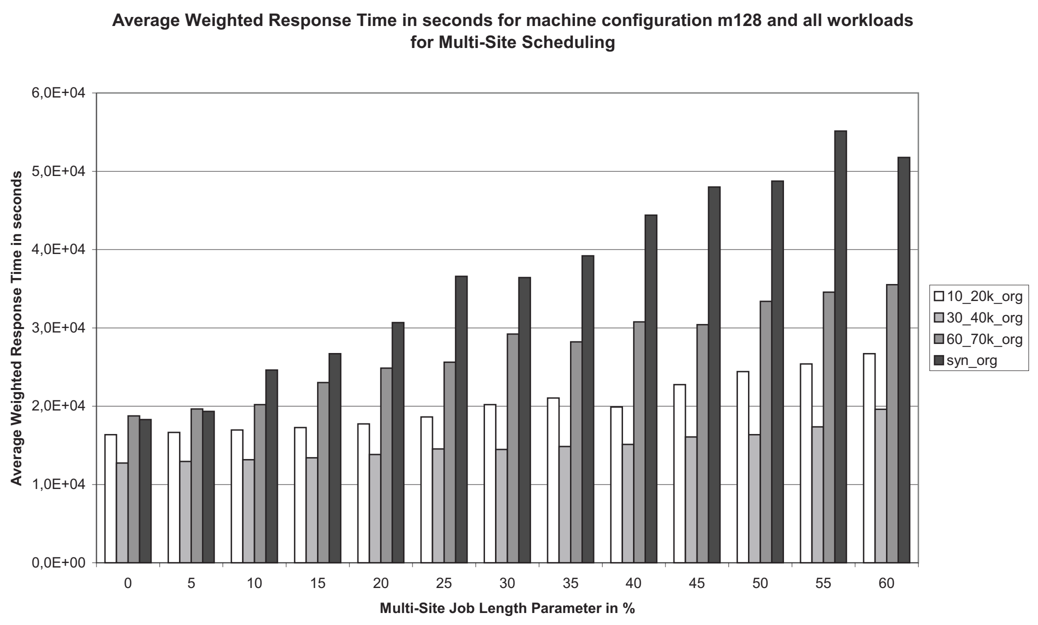

Figure 5.12: The average weighted response time in seconds for machine configuration m128 and all workloads for Multi-Site Scheduling

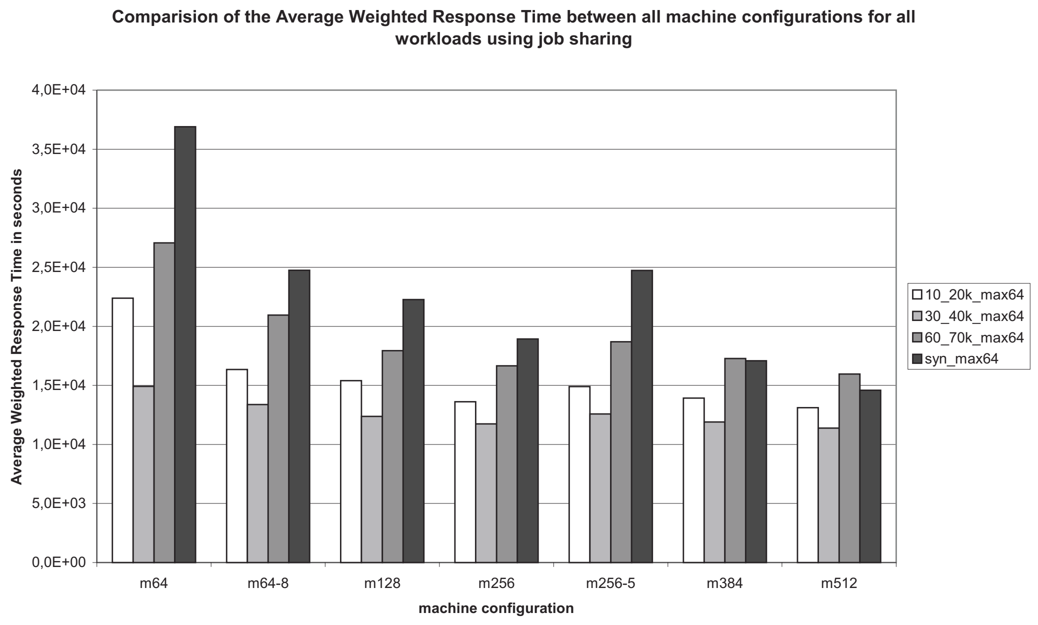

Figure 5.13: The average weighted response time in seconds for all workloads and all machine configurations for Job Sharing

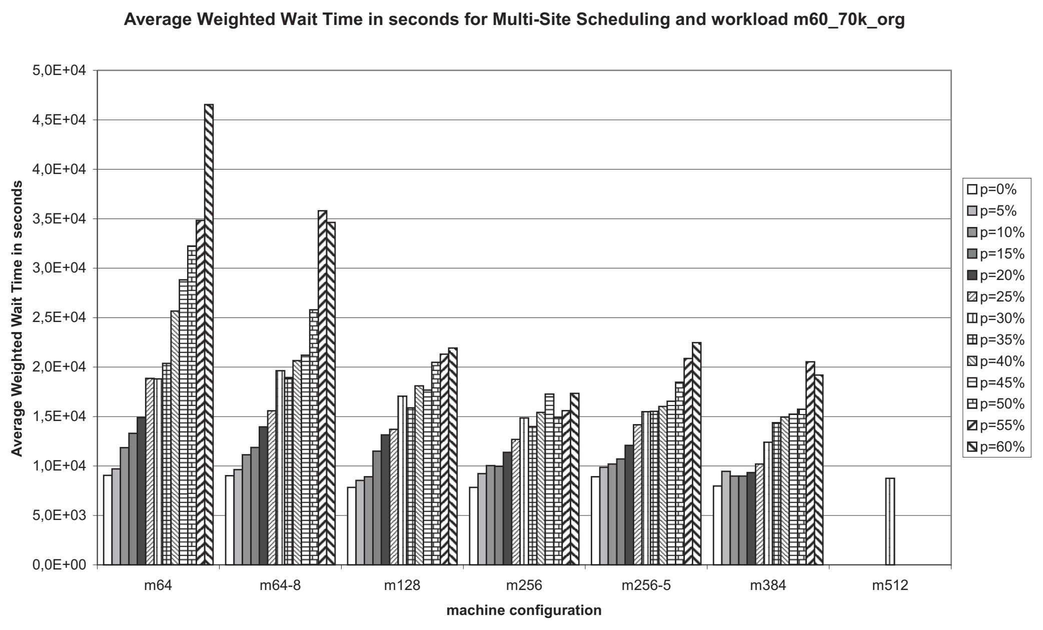

Figure 5.14: The average weighted wait time in seconds for workload 60_70k_org and all machine configurations for Multi-Site Scheduling

Backlog Examination

As we already mentioned, the evaluation of scheduling algorithm is dependent on the examined workload. According to our concept for the design process for a scheduling algorithm, the administrator defines the workload for the simulations. In this examination we assume a grid scenario in which the user community and the workload correspond to the CTC setting. The grid size has been selected accordingly. In the following we want to check if the workload modelling is appropriate. The workloads have been derived from traces that have been recorded at a different machine configuration with a different scheduling system. There are at least two potential risks in workload modelling by traces: on one hand, the jobs in the generated schedule may not utilize enough resources. In such a case the average weight time can be very small as there are frequently free resources and submitted jobs can be started immediately. On the other hand, there is a risk that constantly more workload is submitted as it can be executed. In this case the wait time grows on average during the scheduling process.

Therefore, we want to examine the backlog. As a criterion the wait time of the jobs is suitable. We use the average weighted wait time (AWWT) with our presented weight selection, which additionally takes the amount of workload in the backlog into account.

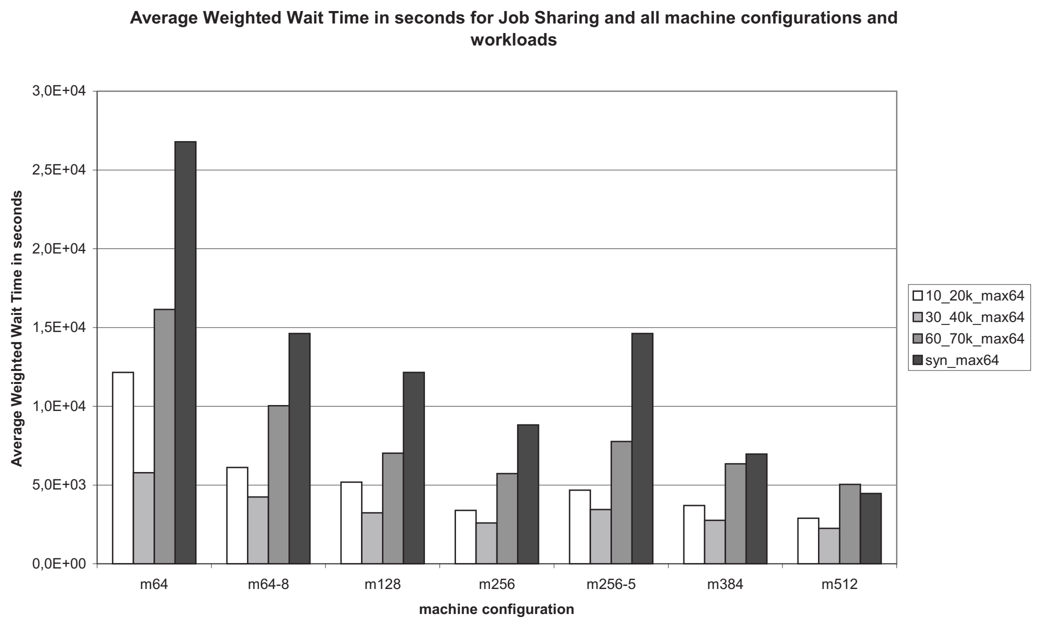

Figure 5.15: The average weighted wait time in seconds for all workloads and all machine configurations for Job Sharing

The AWWT is defined as follows:

$$\text{AWWT} = \frac{\sum_{i \in \tau} (t_i - s_i) \cdot w_i}{\sum_{i \in \tau} w_i}$$

In comparison, the average wait time without the weight would relate to the number of jobs in the backlog and as a measure would favor small jobs. However, both criteria can be used as a mean for the backlog. As we used the weight selection for the AWRT to not prioritize smaller over larger jobs, we also decided to use AWWT for the backlog examination. Note, that the splitting of a long job to several shorter jobs with the same submission time change the AWWT due to the nature of wait-time in comparison to the response-time. Note also, that the AWWT differs only by a factor from the AWRT. This is the weighted execution time of all jobs as can be seen from the AWRT definition on Page 38 .

The simulation results show that the AWWT is at least two hours. As an example the AWWT for simulations with the workload 60_70_org for all used machine configurations and multi-site scheduling is given in Figure 5.14. The average weighted waiting time of about two hours indicates the existence of an appropriate backlog. Such a backlog is for example important for the backfilling algorithm as previously mentioned. The examination of the actual schedule also showed that there was no job starvation in the long run of job submissions. That is the backlog did not grow significantly during the simulation. The examination of the different scheduling strategies, machine configurations and workloads generated similar results. Figure 5.15 shows the results for the job sharing scenario for all workloads and all machine configurations. The minimal AWWT is about 45 minutes and so there is also evidence for a sufficient backlog.

Generally, the results for the AWWT, as presented in Figure 5.14, show a similar behavior as in the results in Figure 5.9. As mentioned, this is the case because the AWWT differs from the AWRT only by the weighted execution time of all jobs.

5.8 Summary

The results show that the collaboration between sites by exchanging jobs even without multi-site execution significantly improves the average weighted response time. This is already achieved with a simple algorithm for a central scheduler as used in this work. Note, that the applied algorithms for scheduling are simple extensions of backfilling and node selection strategies.

Furthermore, the usage of multi-site applications leads to even better results under the assumption of a limited increase of the job execution time due to communication overhead. Even an increase of their execution time by 25% multi-site proved to be beneficial compared to job-sharing. Note, we do not conclude that multi-site is suitable for all applications. WAN networks are in terms of latency in the order of 2-3 magnitudes slower than common fast interconnection networks between nodes inside a parallel computer, e.g. an IBM SP Switch. Thus, the actual overhead caused by multi-site may be much higher. The question if multi-site execution is suitable, depends on many factors as e.g. the actual communication pattern, the requirement in data I/O of input/output data. Nevertheless, the results indicate that multi-site execution may be beneficial in terms of response time reduction for applications with a limited demand in communication as presented in our results.

Overall, as expected the results show that our scheduling strategies produce smaller average weighted response times for configurations with large machines in comparison to configurations with more but smaller machines. For example, the increase for the average weighted response time stays in general below 15% for a configuration with four machines of 128 nodes compared to a single large machine with 512 nodes. Comparing the results with the workload characteristics, it can be observed that job systems with jobs limited in resource demand (e.g. 50% of the largest machine) result in a minor drawback in configurations with smaller machines. It can be expected that an adequate ratio between the workload of large jobs and the available computing power of the large machines is necessary to guarantee an acceptable response time. As long as this requirement is met, the remaining resources may consist of smaller machines without implying a significant drawback. This shows how the evaluation can be useful for system selection and configuration.

In contrast to job-sharing the usage of multi-site scheduling allows the execution of jobs that are not limited by the maximum machine size. The results show that the resource configurations and the overhead due to multi-site scheduling have a strong impact on the AWRT of the schedule. Configurations of larger machines are superior to those with smaller machines. If there is only a small overhead for multi-site execution ($p < 20\%$), balanced large machine configurations show a slight advantage compared to unbalanced systems with small machines. Especially on smaller resource configuration a large overhead results in a very steep increase of the AWRT, as more jobs are executed in multi-site mode in configurations with a higher number of smaller machines. Nevertheless, there are some values for the overhead that lead to a decrease or at least no increase of the AWRT compared to larger overheads in the same scenario. This may be caused by two effects. First, the number of jobs that are executed in multi-site mode increases corresponding to the size of the overhead, while the amount of work by these jobs shows a much lower increase than the overhead. Therefore, each job must be smaller in average. This leads to the assumption, that jobs used for multi site scheduling differ in each scenario for each size of the overhead. Second, the increase of the communication overhead leads incidentally to a more suitable job size as certain pattern of job execution times are more common than others. Recent results showed [16], that the results for multi-site computing can be further improved by introducing additional bounds for jobs size and the allowed number of job parts.

As grid environments and networks are becoming more and more common, it seems reasonable for resource owners to participate in such initiatives. Simple strategies like job sharing significantly improve the average weighted response time and therefore the quality of service to the users. Also the research and effort in developing multi-site programs for suitable applications with limited demand in network communication can provide even better results. Furthermore, multi-site applications can effectively use more resources for a single job than available at any single machine. The drawback on the overall schedule quality in regards on the AWRT due to submitting a wider instead of several smaller jobs was limited in our simulations, about 10 - 20%. Of course this may vary with the amount of wide jobs in a workload. It has still to be kept in mind, that the quality of a schedule always depends on the actual configuration and workload. The presented improvements were achieved using example configurations and workloads derived from a real trace. Nevertheless, the results show that job-sharing and multi-site execution in a grid environment are capable of improving the scheduling quality for the users significantly.

Nevertheless, our model has limitations as we consider a central scheduling instance. The actual implementation may use a different and distributed architecture that can act from a logical point of view still similar to our model. But overall the scalability and reliability remain important issues which have to be considered. We will address these aspects in the next chapter.

Our evaluations showed good results for response time. Other examinations not presented here showed also that high utilization of the machines were achieved. But we have to ask if the simple assumptions for the user and owner objectives in regards to the response time and utilization suffices. Up to now we assumed that minimizing the response time is the main objective. In the next chapter we will present and examine a different scheduling method which is based on an economic model. This new model has several features that seem appropriate for grid computing. One of them is the support of variable objective formulation.

Chapter 6

Economic Scheduling Model

In the first chapters of this work we examined scheduling algorithms and their applications for the management of single parallel machines. Later, we adapted them for the application in grid environments. There, we showed that the participation in a grid environment may be beneficial for users and owners of such systems.

As already mentioned the challenges for scheduling methods differ from single machine scheduling as the resources are geographically distributed and owned by different individuals. The scheduling objective on a single parallel machine is usually the minimization of the completion time of a computational job. We used this objective in our previous evaluations of this work by examining the average weighted response time.

In a grid environment there are much more different users and owners and these individuals will typically not know each other. In comparison to accessing local resources where user may not consider costs and cannot change system properties, in a grid environment other requirements and expectations may apply. Consequently, other objectives have to be considered as e.g. cost, quality of service or additional time constraints, e.g. given start and end time limits. The examined algorithms in this work are mostly derivations of list-scheduler that are usually predestined for a single overall objective. To this end, other scheduling approaches are necessary that may deal better with different user objectives as well as owner and resource policies. For instance, the grid scenario with independent user or resource owner can be compared to a network in which independent parties trade for goods; in our case a good is the resource allocation for a certain time. This is a typical application scenario in which economic models are used.

In the following, we want to address the idea of applying economic models to the scheduling task. To this end a new market-economic method for grid scheduling is presented and analyzed. In our study we use the evaluation process from the previous chapter. The quality of an economic scheduling is more difficult to measure as the former single objective is now dependent on the specific and variable objective formulation and the corresponding cost-metric. Additionally, the process for offer and request generation has to be taken into account. While there is a high degree freedom in this processes, heuristics are often applied to limit the time consumption for the execution of market methods. This is one reason why theoretical analysis is difficult to apply. In our example we want to focus on the practical performance for a real system scenario as presented in the previous chapter. Because economic methods support different objectives, it is difficult to compare and evaluate such methods to other approaches. In this work, we therefore examine the efficiency of this economic approach by the means of response-time minimization by simulations with real workload traces [17]. Note, we assume for this study that the response time should be minimized despite the additional features by economic models as the mentioned support for variable objective formulation. This allows the examination of the performance that can be achieved by economic approaches in comparison to the conventional methods as presented in the previous chapter.

6.1 Market Methods

Market methods for computational tasks have been subject of research for some time. An overview of such models is, for example, given by Buyya in [6]. In this work, we give a brief introduction on the background needed for our scheduling setting.

We just give a brief introduction on the background needed for our scheduling setting.

6.1.1 Considerations on Market Methods

Market methods, sometimes called *Market oriented programming* in combination with Computer Science, can be used to solve the following problems which occur in real scheduling environments [10]:

- **The site autonomy problem** arises as the resources within the system are owned by different individuals, companies or institutions.
- **The heterogeneous substrate problem** that results from the fact that different installations use different resource management systems.
- **The policy extensibility problem** means that local management systems can be changed without any effects for the rest of the system.
- **The co-allocation problem** addresses the aspect that some applications need several resources of different companies at the same time.
- **The online control problem** is caused by the fact that the system works in an online environment.

We address these problems from an architectural point of view in more detail in Chapter 6.

The supply and demand mechanisms provide the possibility to optimize different objectives of the market participants under the usage of costs, prices and utility functions. It is expected that such methods provide high robustness and flexibility in case of failures and a high adaptability during changes.

As a general definition, a *market* can be defined as a virtual market or from an economical point of view as follows: *"Generally any context in which the sale and purchase of goods and services takes place."* [76]. The minimal conditions to define a virtual market are: *"A market is a medium or context in which autonomous agents exchange goods under the guidance of price in order to maximize their own utility."* [76]. The main aspect is that autonomous agents exchange voluntarily their goods in order to maximize their own utility.

A *market method* is given by Berman and Tucker in [77] as follows: *"A market method is the overall algorithmic structure within a market mechanism or principle is embedded."* It has to be emphasized that a market method is an equilibrium protocol and not a complete algorithm.

The definition of an *agent* can also be found in [77]: *"An agent is an entity whose supply and demand functions are equilibrated with those of others by the mechanism, and whose utility is increased through exchange at equilibrium ratios."*

It is now the question how this equilibrium can be obtained. One possible method is the application of *auctions*: *"An auction is a market institution with an explicit set of rules determining resource allocation and price on the basis of bids from the market participants"* [82]. The most common auctions are: the English Auction, the Dutch Auction, the Sealed Bid Auction and the Double Auction that are described in detail by Walsh et.al. [77]. More details about the general equilibrium and the existence of the general equilibrium is given by Ygge in [86].

6.1.2 Economic Scheduling in existing systems

Economic methods have been applied in various contexts. Besides the references given by Buyya in [6], we want to briefly mention some other typical algorithms of economic models.

WALRAS

The *WALRAS* method is a classic approach by translating a complex, distributed problem into an equilibrium problem [3]. One of the assumptions is that agents do not try to manipulate the prices with speculation, which is called a *perfect competition.* To solve the equilibrium problem the *WALRAS* method uses a *Double Auction.* During that process all agents send their utility functions to a central auctioneer who calculates the equilibrium prices. A separate auction is started for every good. At the end, the resulting prices are transmitted to all agents. As the utility of goods may not be independent for the agents, they can react on the new equilibrium prices by re-adjusting their utility functions. Subsequently, the process starts again. This iteration is repeated until the equilibrium prices are stabilized.

The *WALRAS* method has been used for transportation problems in the area of processor scheduling and rental, see also [43]. The transportation problem requires to transport different goods over an existing network from different start places to different end places. The processor rental problem consists of allocating one processor for different processes, while all processes have to pay for the utilization.

Enterprise

Another application example for market methods is the *Enterprise* [75] system. Here, machines create offers for jobs to be run on those machines. To this end, all jobs describe their necessary environment in detail. After all machines have created their offers the jobs select between these offers. The machine that provides the shortest response time has the highest priority and will be chosen by the job. All machines have a priority scheme where jobs with a shorter run time have a higher priority.

Additionally economic concepts have been examined for different application fields, e.g. the Mariposa project which is restricted to distributed database systems [72]. The presented methods are examples on market economic methods. In the next sections we present our infrastructure and scheduling method for grid job scheduling.

6.2 Economic Scheduling

This section includes a description of the scheduling algorithm that has been implemented for the presented infrastructure. The general application flow can be seen in Figure 6.1. In contrast to Buyya et.al. in [7] and [6] or Waldspurger et.al. in [81], our scheduling model does not rely on a single central scheduling instance. Moreover, each domain acts independently and may have different objective policies, see also Chapter 7. Additionally the job requests of the users can contain individual objective functions. The scheduling model has the task to combine these objectives to find the equilibrium of the market. This is a derivation of the previously presented methods of WALRAS and Enterprise. Our method also supports the co-allocation of distributed resources in different domains. This feature is useful for multi-site job execution.

In our scheduling model all users submit their job requests to the local scheduler of the domain. The scheduler first analyzes those requests and, if possible, creates new offers for all local machines. After this step, a first selection takes place where only the best offers are selected. The request is forwarded to the schedulers of other known domains. This is possible as long as the number of *hops* (search depth of involved domains) for this request is not exceeded and the time to live for this request is still valid. In addition none of the domains must have received this request before. The selection which domain is queried for offers can be implemented by different strategies. In a peer-to-peer approach, in which also network-locality can be exploited, each domain keeps a list of domains it asks. This list can be set up manually based on network structure or according some logical hierarchy. Furthermore, it is possible to setup information services that provides addresses for domains and information

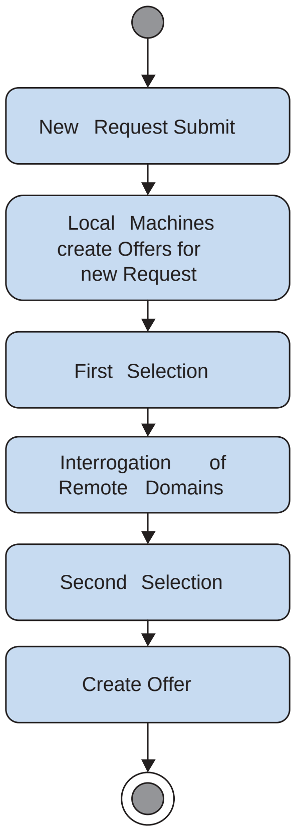
New Request Submit
Local Machines create Offers for new Request
First Selection
Interrogation of Remote Domains
Second Selection
Create Offer

Figure 6.1: General application flow.

on the available resources. Such a directory service can be used to find domains in which suitable resource for a specific request are installed.

The remote domains create new offers and return their best to the requesting domain. If a request for a job has already been processed before, no further offers are generated. That is, a domain answers on requests for a job only once as the same request may be forwarded from different domains. Afterwards, a second selection process takes place in order to find the best offers among the returned result of this particular domain.

Note, that this method is an auction with neither a central nor a decentral auctioneer. Moreover, the different objective functions of all participants are used for equilibration. For each potential offer o for request i the utility value $UV_{i,o}$ is evaluated and returned within the offer to the originating domain that received the user's request. The utility values are calculated by the user supplied utility function UF_i which can be formulated with the job and offer parameters. Additionally to this parameter set $\vec{P_u}$ the machine value $MV_{i,j}$ of the corresponding machine j can be included.

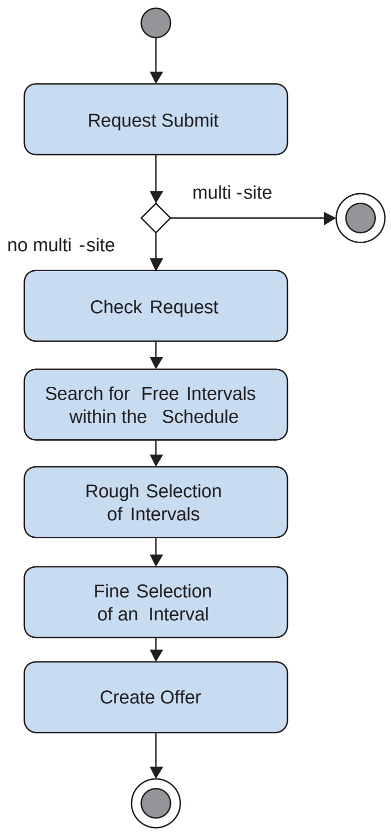
Request Submit
multi -site
no multi -site
Check Request
Search for Free Intervals within the Schedule
Rough Selection of Intervals
Fine Selection of an Interval
Create Offer

Figure 6.2: Local offer creation.

$$
\begin{aligned}
UV_{i,o} &= UF_i(\vec{P_u}, MV_{i,j}) \\
MV_{i,j} &= MF_i(\vec{P_m})
\end{aligned}
$$

The machine value results from the machine objective function MF which can depend on a parameter set $\vec{P_m}$.

The originating MetaManager selects the offer with the highest utility value $UV_{i,o}$. In principle this MetaManager serves the tasks of an auctioneer.

A more detailed explanation of the local offer generation is given in Figure 6.2.

Within the *Check Request* phase it is determined if either the best offer will be automatically selected or if the user is going to select the best offer interactively among a given number of possible offers.

In the same step it is checked whether the user‘s budget is sufficient in order to process the job at the local machines. Additionally, it is determined whether the local resources meet the requirements of the request. Next, the necessary scheduling parameters are extracted which are e.g. the earliest start time of the job, the deadline (end time), the maximum search time, the time until the resources will be reserved for the job (reservation time), the expected run time and the number of required resources. Another parameter is the utility function which is applied in the further selection process.

If not enough resources can be found during the *Check Request* phase, but all other requirements can be fulfilled by the local resources, a *multi-site scheduling* will be initiated unless explicitly forbidden by the used. In this case additional and modified offers are requested from remote domains to meet in combination the original job requirements.

The next step *Search for free intervals within the schedule* tries to find all free time intervals within the requested time frame on the suitable resources. For a simple example assume a parallel computer with dedicated processors as the resources. The example schedule is given in Figure 6.3. The black areas within the schedule are already allocated by other jobs. Now a new incoming job requests three processors and has a start time A, an end time D and a run time less than $(C - B)$. First, free time intervals are extracted for each processor. Next, the free intervals of several processors are combined in order to find possible solutions. To this end, a list is created with triples of the form {time, processor number, +/-1} which means that the processor with the specified processor number is free (+1) or not free (-1) at the examined time.

The generated list is used to find possible solutions as shown in the following pseudocode:

```
list tempList; LOOP: while(generatedList not empty) {
    get the time t of the next element in the sourceList;

    test for all elements in tempList whether the difference between
    the beginning of the free interval and the time t is bigger or
    equal to the run time of the job;
    if(number of elements in tempList, which fulfill the time
       condition, is bigger or equal the needed number of processors)
       {
         create offer from the elements of the tempList;
       }
    if(enough offers found)
       {
         finish LOOP;
       }
    add or substract the elements of the sourceList to or from
       tempList which have time entry t; }
```

The given algorithm creates possible offers that are specified by start, end, run time and the requested number of processors.

Note, that we did not yet show how the offer is created from the elements of this list. This is achieved by the following algorithm. The goal is to find areas of enough

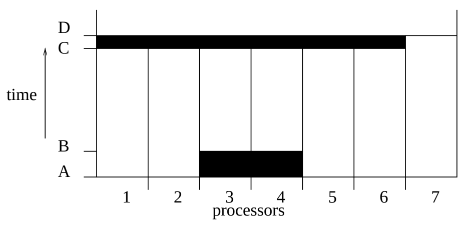

Figure 6.3: Start Situation.

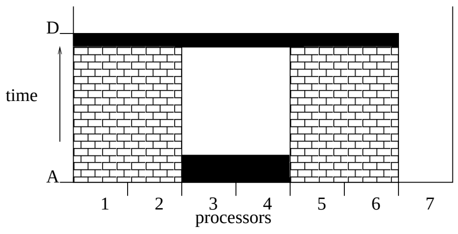

Figure 6.4: Bucket 1.

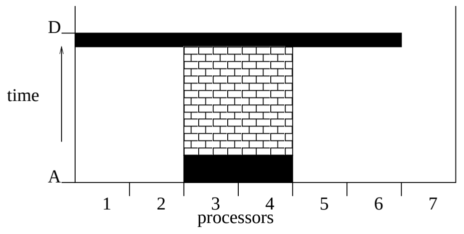

Figure 6.5: Bucket 2.

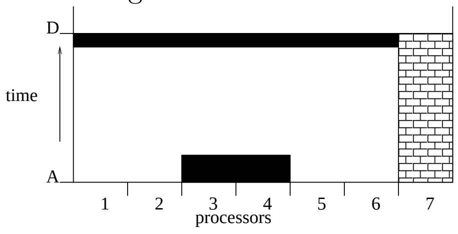

Figure 6.6: Bucket 3.

resources within the schedule for a given list of free time intervals. This has to take into account that idle times at resource elements may have different start and end times. The resulting areas are characterized by the earliest start and latest end time. To this end a derivation of a bucket sort is used. In the first step all intervals with the same start time are collected in the same bucket. In the second step for each bucket the elements with the same end time are collected in new buckets. At the end each bucket has a list of resources available between the same start and end time.

For the example above, the algorithm creates three bucket as shown in Figures 6.4, 6.5 and 6.6. After this bucket creation, suitable offers are generated either with elements from one bucket if the bucket includes enough resources or by combining elements of different buckets. Additional care must be taken as elements from different buckets can have different start and end times. The maximum start and the minimum end time must be calculated. In our example only bucket 1 can fulfill the requirements alone and therefore an offer can be built e.g. with resources 1, 2 and 5.

In order to generate different offers buckets for which an offer could be generated by using only its own elements are modified to contain one resource less than the required number. Afterwards, the process of offer generation is repeated. If the number of elements within a bucket is less than the necessary number, all elements of this bucket are taken into the collection bucket. For our example this is true for the buckets 2 and 3. If yet not enough solutions are found and no further bucket can fulfill the request by itself, and the number of remaining elements of all buckets is greater or equal to the requested resource number, new solutions are generated by combinations of bucket elements with appropriate intersecting time frames.

In our example, combined with the solution built from bucket 1 the whole set of solutions would be: {{1,2,5}, {1,2,3}, {1,2,4}, {1,2,7}, {1,3,4}, {1,3,7}, {1,4,7}, {2,3,4}, {2,3,7}, {3,4,7}}.

After finishing the phase *Search for free intervals within the schedule* from Figure 6.2, a rough selection of one of these intervals takes place in the next step. In principle a large number of solutions may be possible due to small changes for the start and end time for a job in every combination and then selecting the interval with the highest utility value. In practice this is not applicable as the run-time of the algorithm must be considered. Therefore, a heuristic is used by first selecting several combination among all possible combinations. The start and end time for these combinations are modified to improve this utility value. The modification with the highest utility value is selected as the resulting offer (in phase "fine selection of an interval" in Figure 6.2). A parameter for the number of steps can be given which defines the number of different start and end times within the given time interval. Note, that the utility function is not constrained in terms that is must be monotone. Therefore, this selection process is also based on heuristics.

After this phase the algorithm is finished and possible offers are generated.

The utility functions of the machine owner and the user have not been specified yet. This method allows both parties to define their own utility functions. In our implementation, presented in Chapter 7, any mathematical formula, using any valid time and resource variables, is supported. Overall, the resulting objective value for the user's utility function is maximized among the available offers. Note, that we do not obtain by this process the general maximum. Instead, from the available offers the solution with the maximum value is selected. The linkage to the objective function of the machine owner is created by the price for the machine usage which equals the machine owner's utility function. The price may be included in the user's utility function.

The owner of the machine can formulate the utility function in which additional variables can occur that depend on the resulting schedule. Figure 6.7 shows variables that are used in our implementation. The variable *under* specifies the area in the schedule in which the corresponding resources (processors) are unused before the job allocation. The variable *over* determines the area of unused resources after the job to the start of next job start on the according resources or to the end of the schedule. The variable *left_right* specifies the concurrently idle resource area during the allocation of the job. In a graphical depiction of a schedule as in Figure Figure 6.7, this is the idle area on the left and right side of the job for all processors of the machine. The variable *utilization* specifies the utilization of the machine if the job is allocated. This is defined by the relation between the sum of all allocated areas to the whole available area from the current time instance to the end of the schedule.

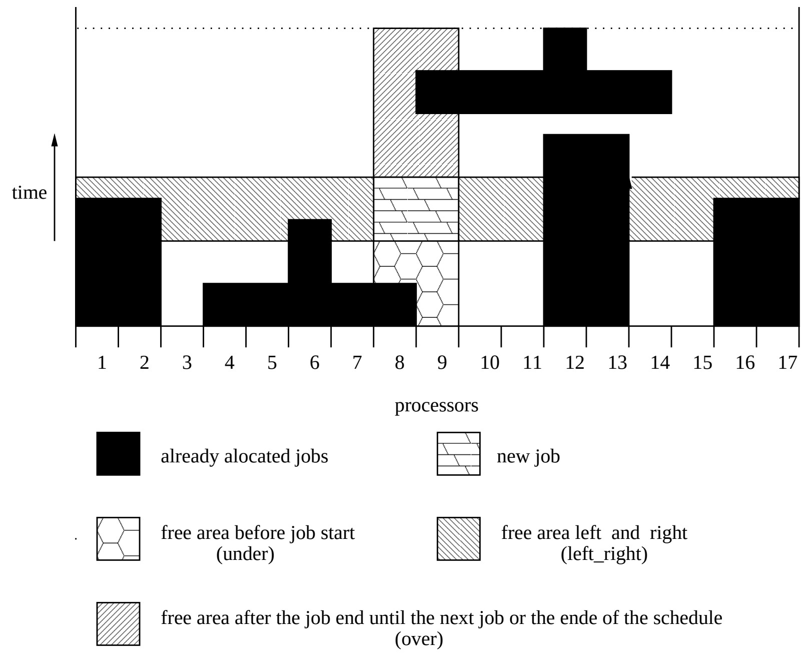
time
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
processors
already alocated jobs
new job
free area before job start (under)
free area left and right (left_right)
free area after the job end until the next job or the ende of the schedule (over)

Figure 6.7: Parameters for the calculation of the owner utility function.

Time Consumption of the Economic Scheduling Algorithm

An important issue for the practical applicability of an algorithm is its run-time behavior. The time-consumption of the algorithm depends on many parameters which define the number of steps and its complexity. Therefore, we give theoretical estimations of the time requirement for the single steps of the algorithm.

Determination of idle time intervals: The process of searching idle times on a resource requires in the worst case $(J + 1)$, while J denotes the number of jobs that have been allocated to the resource. This assumes that the corresponding lists are already ordered at the insertion of job allocations. As we have R resources that are available for a job request, the complexity for this step is maximal: $((J + 1) \cdot R)$.

Search for free intervals within the schedule During this process, we have to consider first the generation of a sorted list with start and end times: The merging of the resource information on start and end times to a list with maximal $((J+1)\cdot 2)$ elements requires at most $((J + 1) \cdot R)$ time.
Next, the algorithm for finding possible solutions as displayed in the pseudo-code on Page 105 is considered.
The loop is executed maximal $(2\cdot(J+1))$ iterations as the list can have that number of entries in the worst-case.

Calculation of potential offers: The process of sorting resources into the buckets, as described above, consists of two steps. The first sorting in respect to the available start-time on a resource requires at most R steps.
The second sorting in respect to the potential end-time on the resource requires again at most R steps.

After the bucket sort the scheduler has to find combinations from these buckets to generate offers. The maximal time-complexity is again R steps, as there are R buckets in the worst-case. In each combination the earliest start-time and latest end-time has to be calculated. The maximum run-time for this step is $\begin{pmatrix} A \\ N \end{pmatrix}$, with N denoting the number of resources requested by the job and A the number of resources in the collection bucket. This is the number of steps if all possible combinations are examined. In a real implementation this process is certainly bound by a parameter $C \ll \begin{pmatrix} A \\ N \end{pmatrix}$.

Therefore, the run-time of the whole search of potential offer intervals requires:

- $((J+1) \cdot 2 \cdot R)$ steps for the search of free intervals of each resource and list creation.
- $(2 \cdot (J+1) \cdot \{\text{runtime for offer generation}\})$ steps for the loop iterations on the generated list.
- $\left(3 \cdot R + \begin{pmatrix} A \\ N \end{pmatrix}\right)$ steps for sorting the time intervals for start and end times into buckets.

In the worst case, the number of elements in the collection bucket A equals the number of resources R. This leads to the overall run-time int the worst-case of:
$2 \cdot (J+1) \cdot \left(R + 3 \cdot R + \begin{pmatrix} R \\ N \end{pmatrix}\right)$ steps.

Again, in an implementation the term $\begin{pmatrix} R \\ N \end{pmatrix}$ will be limited by a parameter C.

Selection of an offer: We described that a heuristic is used for the actual offer selection. This heuristic for selecting resources had a time complexity of $\begin{pmatrix} R \\ N \end{pmatrix}$ steps. Note, that this strategy is an example for the selection process and other variants are possible.
Afterwards, the actual start time has to be selected for the job request as the available time frame on the resources can be longer than the requested execution time. Here, we used a heuristic that produces Z start-times.
The overall maximal run-time (without C) for the selection process is therefore:
$\left(\left(\begin{pmatrix} R \\ N \end{pmatrix} + Z + 2\right) \cdot \{\text{run-time for calculating the utility function}\}\right)$.

Calculation of the owner utility function: The calculation depends on the parameters that are included in the utility function. In our example, we used information on idle times on different machines. This step lead to a time-complexity of $(R \cdot (J + 1))$ in the worst-case.

6.3 Simulation and Evaluation

As mentioned before the economic scheduling method provides support for variable objective functions for user and owners. Therefore, the evaluation of such a economic method highly depends on these objective functions. Common criteria as the minimization of the response time or a high utilization of the resources can be part of these objective functions, but this is not required. Some users for instance may be interested in cost-reduction as long as given response times are met, other users may look for earliest response time without regarding the cost. Thus, the evaluation and comparison of the whole scheduling process by one mean, as e.g. the average weighted response time, may not be appropriate if this is not the single objective of users and owners. Especially the comparison to other scheduling methods that focus on such a objective as for example is meaningless.

However, we can examine the performance of the economic scheduling method if we assume that the overall objective is actually the reduction of the response-time. This allows the comparison to the conventional scheduling algorithms as presented in the last chapter. Note, that this is a scenario where the ability to use different objectives for certain users and owners is not utilized. According to our discussed concept for the scheduling design process, the designer has to identify and model the user workload including the objectives and analyze/optimize the economic method by evaluation. This evaluation step includes the definition of the criterion - objective function - for the overall schedule. In our example, we base our simulation on the definitions that have been used for our evaluations for the conventional algorithms. Note, that we yet do not know how to formulate objective functions for machines and users to achieve a low average weighted response time. To this end, we select several objective functions and discuss the results in respect to the average weighted response times.

In the following sections the simulation environment is described. First, the resource configurations which are used for our evaluation are described followed by an introduction of the applied job model.

6.3.1 Resource Configurations

We use the same configurations as presented in Section 5.7.1 in Table 5.1. Again, all configurations use a total of 512 resources.

Additionally, in order to apply economic scheduling methods, utility functions are required as mentioned before. For the simulations 6 different owner objective functions were used. The first one describes the most general owner utility function in our example from which all others are derived. The owner machine function MF_1 consists of several terms. The first term:

$$NumberOfProcessors \cdot RunTime$$

calculates the area that the job is using within the schedule. The second term calculates the free areas before and after the job as well as the parallel idle time for the other resources within the local schedule (see Figure 6.7.):

$$over + under + left\_right.$$

The last term of the formula is:

$$1 - left\_right\_rel,$$

where $left\_right\_rel$ describes the relation between the free areas to the left and right of the job within the schedule (left_right) and the area actual used by the job. A small factor describes that the free areas on both sides are small in comparison to the job area. This leads to the following objective function MF_1 and its derivations MF_2 - MF_6:

$$\begin{aligned}
MF_1 &= (NumberOfProcessors \cdot RunTime \\
&+ over + under + left\_right) \cdot (1 - left\_right\_rel) \\
MF_2 &= (NumberOfProcessors \cdot RunTime \\
&+ over + under + left\_right), \\
MF_3 &= (NumberOfProcessors \cdot RunTime \\
&+ over + under) \cdot (1 - left\_right\_rel), \\
MF_4 &= (NumberOfProcessors \cdot RunTime \\
&+ left\_right) \cdot (1 - left\_right\_rel), \\
MF_5 &= (NumberOfProcessors \cdot RunTime \\
&+ over + left\_right) \cdot (1 - left\_right\_rel), \\
MF_6 &= (NumberOfProcessors \cdot RunTime \\
&+ under + left\_right) \cdot (1 - left\_right\_rel).
\end{aligned}$$

However, note that these are only examples that are used in this evaluation. Other utility functions are possible.

6.3.2 Job Configurations

For the evaluation we again use the CTC workload. This allows the desired comparison of economic systems in this work to the non-economic scheduling systems. Again, the jobs from the real traces have been assigned in a round-robin fashion to the different sites to model local job submissions. 4 different sets have been examined in the simulations. These are the same workload sets that we used in Chapter 5 as shown in Table 5.2 with 10000 jobs each.

Additionally, a utility function for each job is necessary in economic scheduling to represent the preferences of the corresponding user. To this end, the following 5 user utility functions (UF) have been applied in our simulations. During a simulation, we assume that all users have the same utility function. Again, these are example utility functions to get first information on the impact on the scheduling performance to our mentioned schedule criteria in comparison to the conventional scheduling algorithms. Further work is necessary if the utility function are optimized for a real system.

The first user utility function prefers the earliest start time of the job. All processing costs are ignored.

$$UF_1 = (-StartTime).$$

The second user utility function only considers the calculation costs caused by the job.

$$UF_2 = (-JobCost).$$

The last user utility functions are combinations of the first two, but with different weights.

$$\begin{aligned} UF_3 &= (-(StartTime + JobCost)) \\ UF_4 &= (-(StartTime + 2 \cdot JobCost)) \\ UF_5 &= (-(2 \cdot StartTime + JobCost)). \end{aligned}$$

6.3.3 Results

Discrete event-based simulations have been performed according to the previously described architecture and settings.

Figure 6.8 shows a comparison of the average weighted response time for the economic method and for the conventional first-come-first-serve/backfilling scheduling system. For both systems the best achieved results have been selected. Note, that the used machine and utility functions differ between the economic simulations.

The results show for all used workloads and all resource configurations that the economically based scheduling system has the capability to outperform the conventional first-come-first-serve/backfilling strategy.

Note, it is possible to outperform backfilling as the economic scheduling system is not restricted in the job execution order. Within this system a job, that was submitted after another already scheduled job, can still be started earlier, if corresponding resources can be found. The conventional backfilling strategy used with the first-come-first-serve algorithm [51] can only start jobs earlier if all jobs that

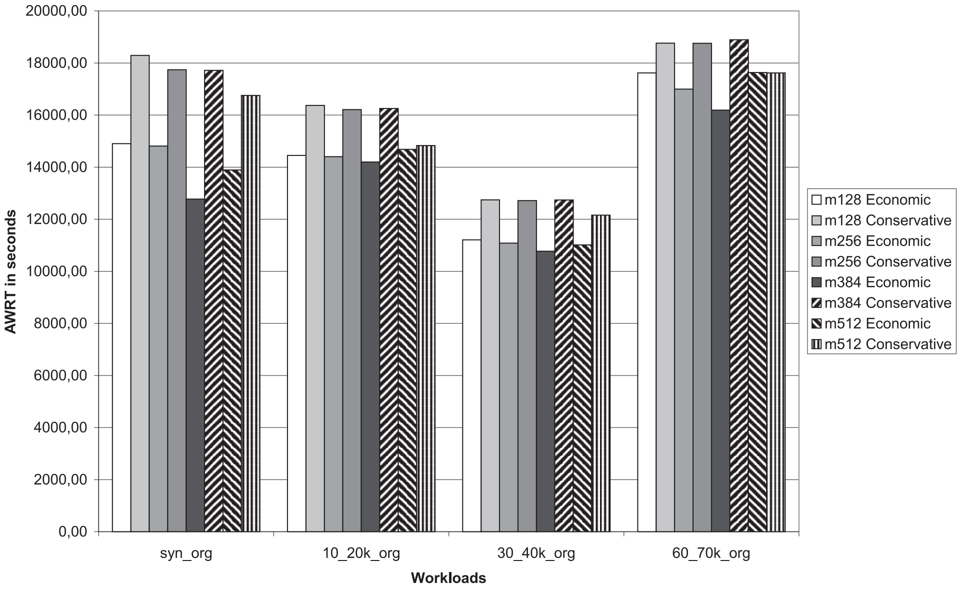

Figure 6.8: Comparison between Economic and Conventional Scheduling.

were transmitted before are not additionally delayed. The EASY backfilling lowers this restriction to not delay the first job in the queue only [29]. However, this does not result in a better performance.

Figure 6.8 shows the best results for the economic scheduling system. Now, in Figure 6.9 a comparison between the economic and the conventional scheduling system for only one machine/utility function combination is presented.

The used combination of MF_1 and UF_1 leads to scheduling results that can outperform the conventional system for all used workloads and configurations m128 and m512. Note, that the benefit of the economic method was achieved by applying a single machine/utility function combination for all workloads. This indicates that certain combinations of machine and user utility functions can provide good results for different workloads, such as the combination MF_1 and UF_1 in our example.

Figure 6.10 presents the average weighted response time (AWRT) as the sum of the corresponding run and wait times weighted by the resource consumption. In all cases the same resource configuration as well as the same machine/utility function combination are used. The time differences between the simulations for both resource configurations are small. This shows that the algorithm for multi-site scheduling (for resource configuration m128), although it is more complex, does not result in a much worse response time in comparison to a single machine. Note, that multi-site execution is not penalized by an overhead in our evaluation. Therefore, the optimal benefit of job splitting is examined and only the capability of supporting multi-site in an economic environment over remote sites is regarded. Here, effects of splitting jobs may even improve the scheduling results. Therefore, we can only compare these results to the equivalent conventional scheduling method without

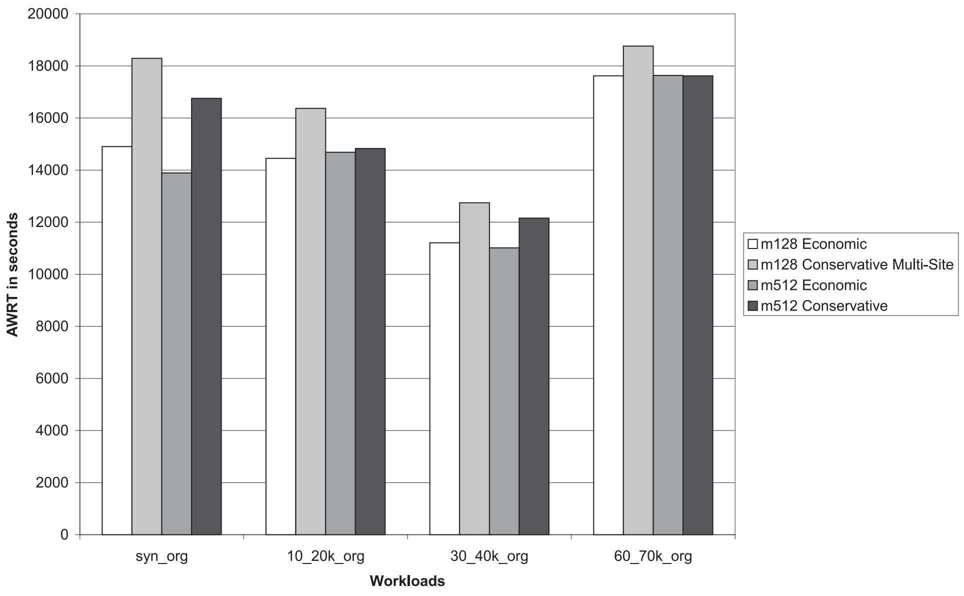

Figure 6.9: Comparison between Economic and Conventional Scheduling for the Resource Configurations m128 and m512 using MF1 - UF1.

multi-site overhead. Note, that the potential impact of using multi-site job execution has been examined in the previous chapter. The actual impact by using overhead depends only on the number of jobs that are executed in multi-site. It can be assumed that the process of splitting jobs can be designed accordingly in different scheduling strategies that the usage of multi-site leads to similar effects.

Figure 6.11 demonstrates that the average weighted response as well as the average weighted wait time do not differ significantly between the different resource configurations. In this case, the machine configurations have limited impact on the effect on multi-site scheduling. Here, the overall number of processors is of higher significance in our economic algorithm. Configurations with bigger machines have smaller average weighted response times than configurations with a collection of smaller machines.

The influence of using different machine/utility function combinations for a resource set is shown in Figure 6.12. Here, the amount of work (the sum of the products of the run time and the number of processors) is given for different resource configurations. The variant m128 is balanced in the sense of having equal sized machines. The desired optimal behavior is usually an equal balanced workload distribution on all machines.

The combination of (MF_1, UF_1) leads to a workload distribution where the decrease of the local squashed area is nearly constant between the machines, ordered by their number as shown in Figure 6.12. The maximum difference between the squashed areas is about 18%.

In the second case, the combination (MF_1, UF_2) presents a better outcome in terms of a nearly equally distributed workload.

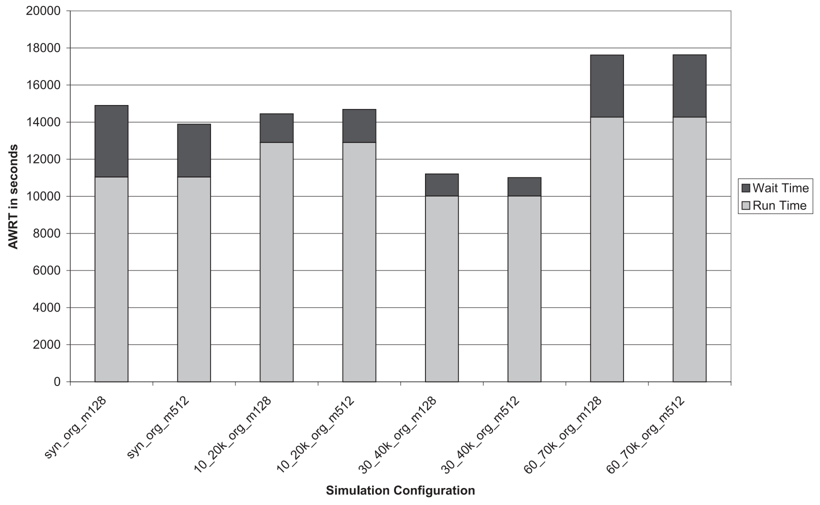

Figure 6.10: AWRT and AWWT for m128 and m512 using several workloads, machine function MF1 and utility function UF1.

The third function combination (MF_2, UF_2) leads to an unbalanced result. Two of the machines execute about 67% of the overall workload and the two remaining machines the rest.

Additional simulation results are shown for keeping the same machine/utility function combinations in Figure 6.13. The combination of (MF_1, UF_2) does not perform very well in terms of the utilization as all machines achieve less than 29%. This indicates in combination with Figure 6.12 that a well distributed workload corresponds with a lower utilization. The combination of (MF_1, UF_1) leads to a utilization between 61% and 77% on all machines. The third examined combination (MF_2, UF_2) shows a very good utilization of two machines (over 85%) and a very low utilization on the others (under 45%). In this case the distributed workloads correlates with the utilization of the machines.

After the presentation of the distributed workload and the corresponding utilization the AWWT and AWRT in Figure 6.14 clearly indicates that only the function combination (MF_1, UF_1) leads to acceptable scheduling results. Figures 6.12, 6.13 and 6.14 demonstrate that different machine/utility function combinations may result in completely different scheduling behaviors. Therefore an appropriate selection of these functions is important for an economic scheduling system.

In the following, we compare different machine/utility functions which are shown for the resource configuration m128. In Figure 6.15 the average weighted response time is given for all different machine function in combination with utility function UF_3. The average weighted response time for the machine function MF_2 performs significantly better than all other machine functions. Here, the factor $1 - left\_right\_rel$, which is used in all other machine functions, does not work well for this machine configuration. Instead it seems to be beneficial to use absolute values for the areas, e.g. ($NumberOfProcessors \cdot RunTime + over + under + left\_right$). Unexpectedly,

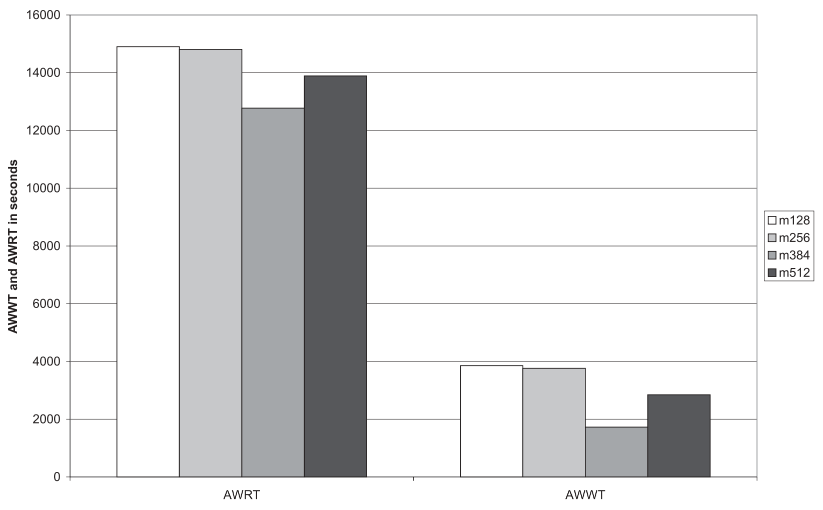

Figure 6.11: AWRT and AWWT for all Resource Configurations and the syn_org workload in combination with MF1 - UF1.

Figure 6.15 also shows that the intention to reduce the free areas within the schedule before a job starts (with attribute *under*) results in very poor average weighted response times (see the results for MF_1, MF_3, MF_6).

As machine function (MF_2) provides significantly better results, different user utility functions are compared in combination with MF_2 in Figure 6.16.

Utility function UF_1, which only takes the job start time into account, results in the best average weighted response time. In this case, no attention was paid to the resulting job cost. For our selection of the machine objective function this means that no minimization of the free areas around the job is regarded. The utility functions which include this job cost deliver inferior results in terms of the average weighted response times. The second best result originates from the usage of the utility function UF_3. In opposite to UF_1 the starting time and the job costs are equally weighted. All other utility combinations in which either only the job costs (UF_2E) or unbalanced weights for the starting time and the job costs are used, lead to higher response times.

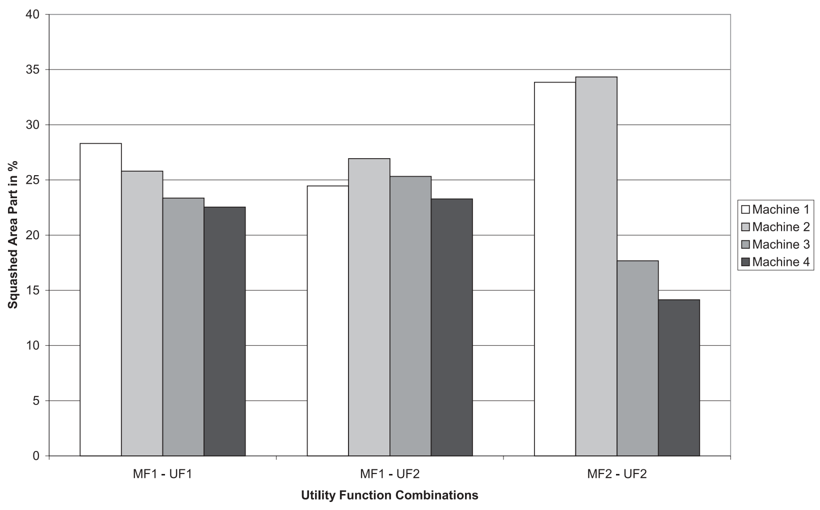

Figure 6.12: The amount of workload (processor-time-product) in simulations with m128 (4 identical machines with 128 processors) and the syn_org workload using different machine and utility functions.

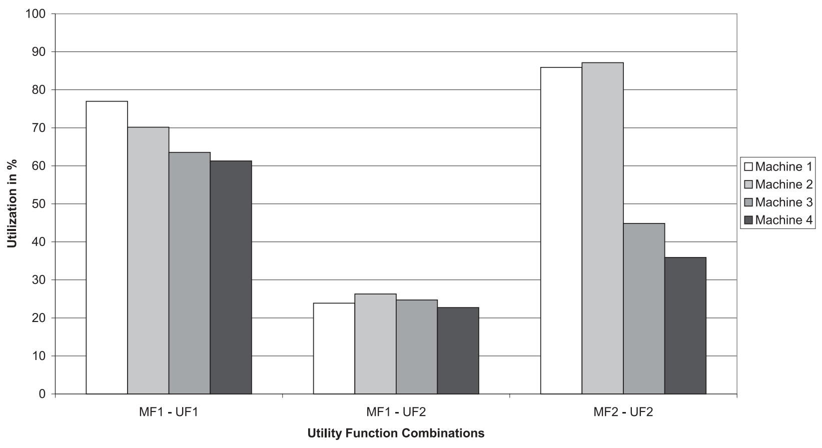

Figure 6.13: The resulting utilization of simulations with m128 (4 identical machines with 128 processors) and the syn_org workload using different machine and utility functions.

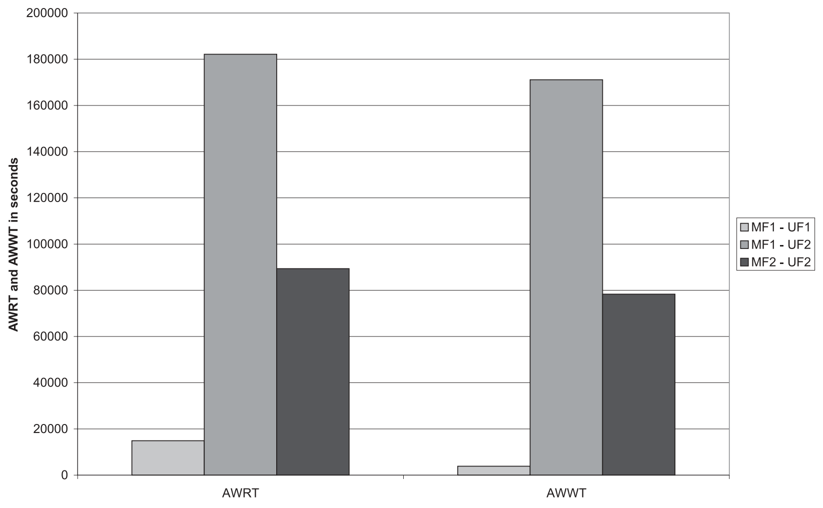

Figure 6.14: The resulting average weighted response and wait times of simulations with configuration m128 and workload syn_org using different machine and utility functions.

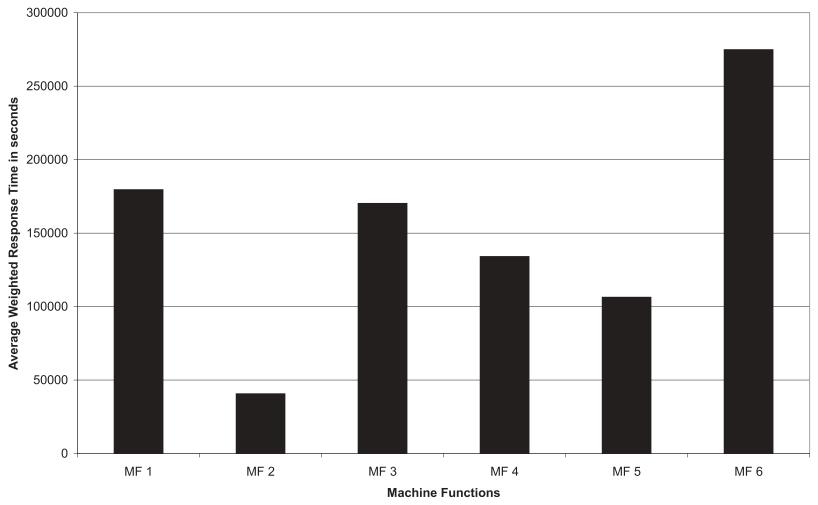

Figure 6.15: The resulting average weighted response for resource configuration m128, utility function UF 3 and several machine functions.

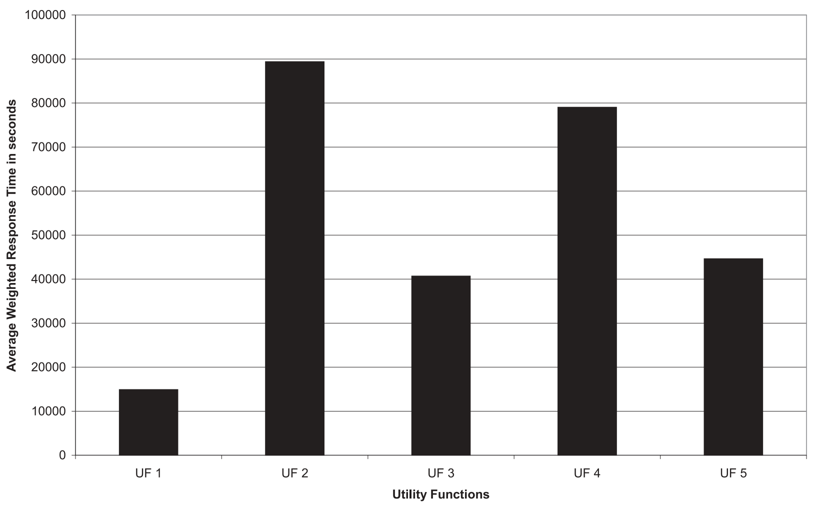

Figure 6.16: The resulting average weighted response for resource configuration m128, machine function MF 2 and several utility functions.

6.4 Summary

In this chapter we presented an economic scheduling system for grid environments. The quality of the algorithm has been examined by discrete event simulations with the same workload and machine configurations as our previous evaluations on conventional grid scheduling algorithms. Now, in this evaluation we focused on the quality in respect to minimizing the average weighted response time. To this end, several parameter settings for owner and user utility functions have been introduced.

The results demonstrate that the used economical model can achieve results in the range of conventional algorithms in terms of the average weighted response time. In comparison, the economical method leaves a much higher flexibility in defining the desired resources. Also the problems of site autonomy, heterogenous resources and individual owner policies are solved by the structure of this economic approach. Moreover, the owner and user utility function may be set individually for each job request. Additionally, features as co-allocation and multi-site scheduling over different resource domains are supported. Especially the advance reservation of resources is an advantage. In comparison to conventional scheduling systems there is instant feedback by the scheduler on the expected start time of a job already at submit time.

Note, that the examined utility functions in the simulations are first approaches and leave room for further analysis and optimization. Nevertheless, the presented results indicate that an appropriate utility function for a given resource configuration delivers steady performance on different workloads.

Further research is necessary to extend the presented model to incorporate the network as a limited resource which has to be managed and scheduled as well. In this case a network service can be designed similar to a managed computing resource which provides information on offers or guarantees for possible allocations, e.g. bandwidth or quality-of-service features.

A more extensive parameter study for comprehensive knowledge on their influence on cost and execution time is necessary. The presented architecture in general provides support for re-scheduling, that means improving the schedule by permanently exploring alternative offers for existing allocations. This feature should be examined in more detail for optimizing the schedule as well as for re-organizing the schedule in case of a system or job failure.

Besides the advantages of the economic models as described above there are additional aspects to consider in an implementation. This includes for example the actual implementation of supporting differentiated objective functions in each job requests as well as the concept of using domains that can query other domains supports an efficient infrastructure where network-locality can be taken into account. In the next chapter, we present details to a implementation of such an environment. As a proof of concept the NWIRE infrastructure of a scheduling environment is presented in which the discussed algorithms have been implemented.

Part IV

Grid Scheduling Infrastructure

Chapter 7

Grid Scheduling Implementation in *NWIRE*

After the design and evaluation of scheduling algorithms for the grid environment, there is still the question whether the algorithms can actually be implemented and how a suitable infrastructure for grid scheduling might look like. In this chapter we first want to discuss the requirements of such an infrastructure. Later on, we present our approach on a scheduling infrastructure.

It is the task of a grid or metacomputing management software to provide the user with a transparent interface to the included resources. As we already mentioned, the initial idea of grid/meta computing is the impression of a single virtual grid computer. The usage should be easy and flexible. Additionally, there is the expectation that the management software simplifies the administration and improves the efficiency of the system.

The aspect of easy and transparent access as well as the increased efficiency is closely related with the scheduling system as we have seen before. It is part of the scheduler to select suitable resources and schedule the execution without further user interaction. In the following we will focus on the scheduling aspects for the infrastructure for grid computing.

Presently, there are a number of projects that work on the realization of a metacomputing environment. These projects include Globus [32–34], Legion [40], Codine [38], Condor [52], AppLeS [1,73,87], LSF [88] and several other initiatives [2]. While many of those systems have been developed especially for a certain class of applications there are a few which use a more general approach. Most notable is the Globus project. It addresses many aspects of grid computing like for instance account management, message passing communication, and security. With regard to the Global Grid Forum, which has the task of coordinating the different grid approaches, Globus has evolved to be the most prominent reference initiative.

The scheduling in Globus is done by a multi-level hierarchy of resource brokers, resource co-allocators, and local managers. Globus also uses a special resource specification language (RSL) to define resources and generate requests.

7.1 Scheduling Considerations

In the previous chapters, we already discussed several aspects in grid scheduling in comparison to scheduling for single parallel machines. Here, we want to repeat briefly general considerations that have to be taken into account for the design of a generic grid scheduling environment, see also [10]. This includes *variable scheduling objectives*, the existence of additional *independent schedulers*, the availability to generate *arbitrary resource requests*, the support of *resource reservation* including the ability to provide *guarantees for job execution*.

Site Autonomy: The resources are typically not owned and maintained by the same administrative instance. Different owners have different scheduling policies that have to be taken into account.

Independent Schedulers: Following from the previous aspect, the resources are not exclusively dedicated to the grid. Hence, the scheduler in a grid computer has no exclusive control over all resources. This leads to the problem that not all jobs on a local machine may be submitted via the grid computing software. This makes grid scheduling more complex as the resource utilization may be changed by the local management system.

Heterogeneous Substrate: The resources typically have its own local management software with different features. Hence, we have to cope with the limitations of the local management. For instance, some scheduling systems are non-deterministic in terms that they cannot provide any information about the expected completion time of a job. Unfortunately, this kind of information is important in distributed metasystems for planning future allocations. That means the grid scheduling system should utilize such features if supported but also cooperate with systems that do not. Nevertheless, the efficiency of a schedule system highly depends on the features of the lower-level scheduler. If a resource does not provide the requested features like a guaranteed completion time, it may not be suitable for some job requests.

Flexible Resource Description: As job requirements and resources in a metasystem may vary according to type and application, there is need for ability to describe complex job requests. This request may be very specific if necessary or very broad. As an example, a user may not provide a very detailed request as he wants to get as many offers for resources as possible. More restrictive requirements would only reduce the possible resource sets for the job. Another user is looking for very specific resources. He may have access to an alternative set of local resources for the execution of his job and is therefore only interested in a better resource allocation. Consequently, he formulates very detailed requirements and preferences. The grid scheduler should support both approaches. The individual user should be able to influence the resource selection and the scheduling to get best results.

Variable Scheduling Objectives: We already discussed the necessity to support variable scheduling objectives. A conventional scheduling system for a parallel computer is optimized for a single scheduling objective or performance metric which is fixed for all jobs. Here, we already noted the minimization of the average response or turnaround time [28]. This objective is typically determined by the local management system or by the administrator. The situation is very different in grid computing. First, the distributed system is not controlled by a single instance. Therefore, the objective may vary for each available resource according to the administrator's policy. While the scheduling target for some machines may, for instance, be the maximization of the throughput, others have the objective to minimize the response time. Second, in addition to the owner objective we must also take the needs of the user into account. Some user may favor the availability of specific resource properties while other may have additional constraints about the execution of a job. A typical example would be the already mentioned deadline for a job that must be met without regard to when it is executed. For this user the minimization of the response time would not reflect his demands. Another user may only be interested in resources that fits his needs better than resources that are already locally available. Therefore, the scheduling must be adaptable to generate the most appropriate result and support individual user objectives.

Co-allocation: This addresses the aspect that some applications need several resources from different sites at the same time. The grid scheduling system should be capable of coordinating these resources. This may be difficult as we have local management systems.

Resource Reservation: Several advanced applications need the support of resource reservation. For instance, a demonstration may require the reservation of a resource allocation for a dedicated time span. It is also advantageous for the scheduler to consider system downtime or restricted access that is known in advance. Reservations are further needed for multi-site applications. It should be possible for a scheduler to reserve resources for a specific time span in order to guarantee the concurrent availability of resources at different locations.

Online Problem: The grid scheduling takes place in an online environment. Jobs may be submitted at any time. Additionally, information on the current state of resources may be difficult to obtain and to keep up to date.

Scalability and Reliability: The scheduler for a single parallel machine has a limited number of resources to control. In comparison, the grid is intended to span over a very large number of systems. Therefore, there are additional requirements in scalability and reliability. If many systems are connected to the grid, the continuous availability and work must be guaranteed. The grid should be usable even if some components are impacted by a network or resource failure. A central scheduling instance would lead to a single point of failure and may also become a performance bottleneck.

7.2 Architecture of NWIRE

Based on the previous considerations we developed the grid scheduling architecture *NWIRE* (Net-Wide-Resources). In this project we focused on the scheduling aspect. NWIRE does not provide all components for a full grid management system. Moreover, it is a proof of concept for a scheduling infrastructure which can be applied to other management systems. It supports the presented scheduling algorithms in this work and has been used for the evaluations.

7.2.1 Description of the Architecture

Most management structures use a similar concept to include different resources. Local resource agents are used to interface between the local system and the management layer. Similarly, in NWIRE resources are abstracted by so-called Resource-Managers. In our implementation these components provide remote access to resources, which are represented by CORBA objects. Other implementation may use different communication standards.

To address the autonomy problem, NWIRE uses independent domains, that are constituted by a set of local resources and local management instances. Each so called MetaDomain is controlled by a MetaManager, as shown in Figure 7.1.

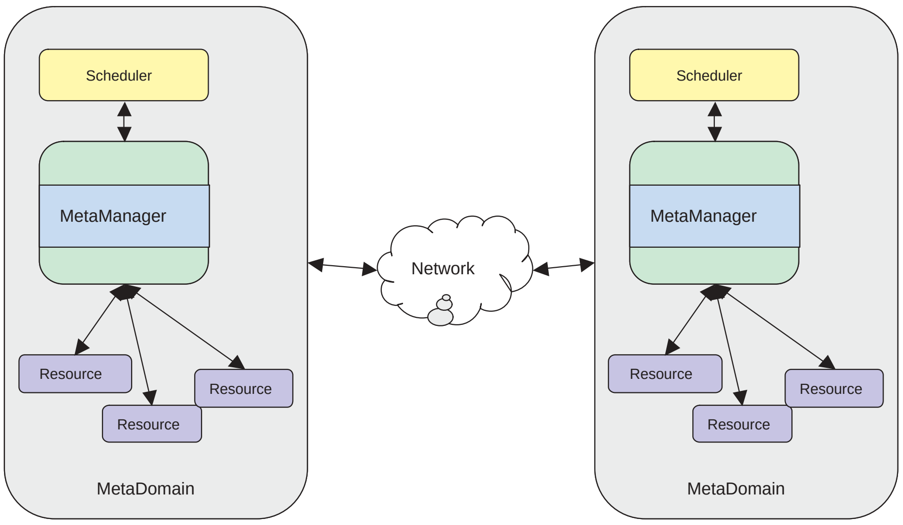

Figure 7.1: Structure of NWIRE

This MetaManager administers the local resources and also answers to local job requests. Additionally, this MetaManager consists of a local scheduler and acts as a broker/trader to other remote MetaDomains, respectively their MetaManagers. That is, the local MetaManager can offer local resources to other domains and tries to find suitable resource allocations for local requests.

The MetaManager can discover other domains by using directory services or exploring the neighborhood similar to *peer-to-peer* network strategies. Requests to another MetaManager can be further forwarded to other domains if necessary. Additional parameters are used to control strategy and depth of this search. Information on the location of specific resource types can be cached for later requests. An overview in all scheduling steps is given in Figure 7.3.

This concept provides several advantages e.g. an increased reliability and fail-safety as the domains act independently. A failure at one site has only local impact as the overall network is still intact.

Another feature is the ability to allow different implementations of the scheduling and the offer generation. Thus, according to the policy at an institution, the owner can setup an implementation that suits his needs best. Note, the policy on how offers for remote job requests are created does not have to be revealed to the public. Instead, job request for resources are analyzed locally by a MetaManager and its offers are returned. The process of this offer generation is a black-box from the point of view of a requesting clients or of MetaManagers from other domains.

Also, the scheduling-infrastructure in a MetaManager provides the ability to implement different strategies for the scheduler. This includes the ability to use conventional methods like e.g. the previously mentioned backfilling as well as the proposed economic scheduling strategy. Within the *NWIRE* system, this is achieved by using so called *requests* for the information exchange between the user and the components involved in the scheduling. The *request* is a flexible description of the conditions of a resource that are necessary for a *job*.

Note that we did not specify the term resource. In *NWIRE* it is simply required that

- certain methods can be applied to a resource and
- a resource provides status information to the attached *ResourceManager*.

For instance, in a grid environment the interconnection network between two multiprocessor systems can be considered a resource. A typical example would be the setup of quality-of-service features in switches and routers or the reservation of guaranteed bandwidth (e.g. with ATM channels).

From a functional point of view a *MetaManager* permanently collects information about all resources associated with it. Further, it handles all requests inside its *MetaDomain* and works as a resource broker to the outside world - to other *MetaDomains*.

7.2.2 Properties of the Architecture

The architecture of NWIRE especially addresses the autonomy problem by using independent domains. The administrator of resources can build a local domain and keep full control over his resources. The management and scheduling is performed locally. Hence, the scheduling policy and objective can be introduced in the design of the scheduling part of the local MetaManager. The owner or administrator of such

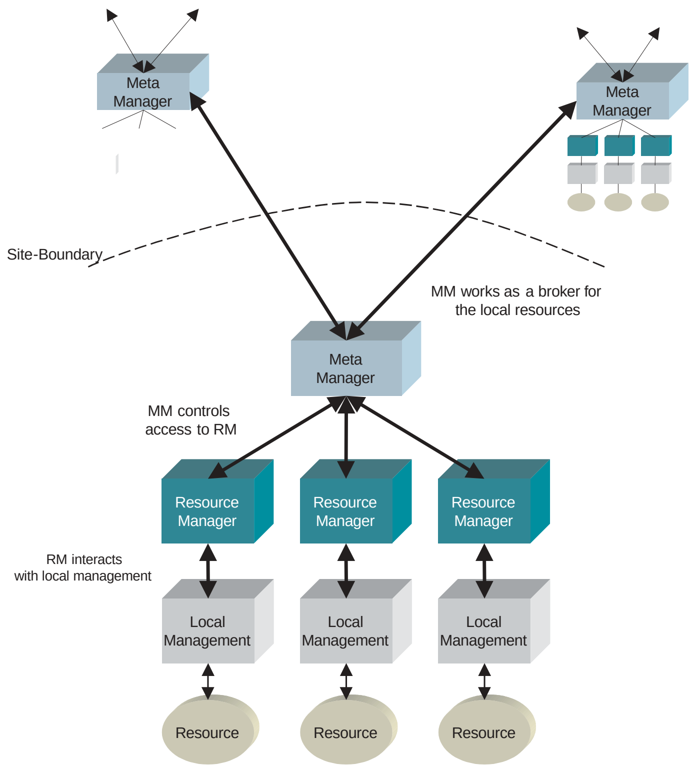

Figure 7.2: Distributed Architecture of NWIRE

a machine may decide to temporarily exclude his resource from the grid or use it for additional local requests that are not received through the grid environment.

Furthermore, the infrastructure supports arbitrary resource types. As we mentioned a resource may be more than a computational resource. Nevertheless, the distinct description and features can be included in the system and easily incorporated into the scheduling process.

In the next section, we will illustrate the scheduling in regards to implementation aspects.

7.3 Scheduling in NWIRE

NWIRE relies on a mechanism of trading resources between domains via the Meta-Manager. The schedule quality in the trading approach relies on a market mechanism. Here, the description of the objectives is a key element for the resource allocation. Chapter 6 showed our model for a economic scheduling approach.

We assume that a metacomputing system will only be accepted, if it meets most or all of the requirements defined in Section 7.1. Especially the ability to supply a scheduling objective for every resource is important as many owners of computing

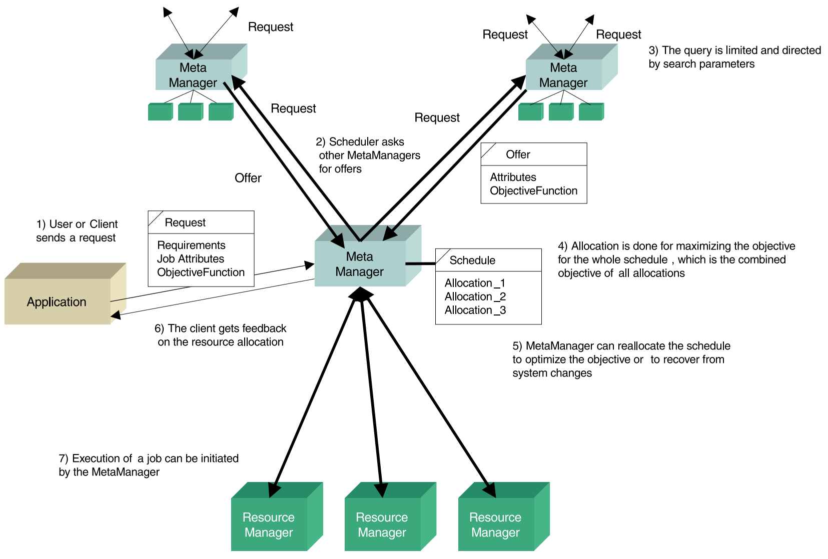
Request
Meta Manager
Request
Request
Meta Manager
Request
3) The query is limited and directed by search parameters
Request
Request
2) Scheduler asks other MetaManagers for offers
Offer
Offer
Attributes
ObjectiveFunction
1) User or Client sends a request
Request
Requirements
Job Attributes
ObjectiveFunction
Meta Manager
Schedule
Allocation_1
Allocation_2
Allocation_3
4) Allocation is done for maximizing the objective for the whole schedule , which is the combined objective of all allocations
Application
6) The client gets feedback on the resource allocation
5) MetaManager can reallocate the schedule to optimize the objective or to recover from system changes
7) Execution of a job can be initiated by the MetaManager
Resource Manager
Resource Manager
Resource Manager

Figure 7.3: Scheduling Steps in NWIRE

resources are only willing to share their resources with someone else under certain conditions. This includes political scheduling [49] as well as strategies on the overall machine utilization. Currently, this seems only to be feasible with a trading system. In addition, this approach provides the highest degree of flexibility and is usually rather fault-tolerant. If a single trader fails only some jobs may be affected but not the whole metasystem.

7.3.1 Resource Request

According to the architecture of *NWIRE*, as presented in Section 7.2, a *request* for resources is directed to the *MetaManager* of a *MetaDomain*. This request is formulated in a resource description language which allows arbitrary combinations of requirements and attributes about the job and the user. An objective function can be part of this request. This function is then used to calculate a value for the utility of a resource allocation, see also [77]. The request is parsed to determine whether a resource is suitable. While it is not necessary that the scheduler recognizes all attributes in a request, some other properties are known by the management system. This method allows arbitrary resource definitions. But resources of the same class must have the same set of obligatory and optional attributes. Otherwise if the interfaces differ, users and programs would be unable to specify or even access those resources. However, this definition must not be known by the scheduler and can therefore be modified later and standardized to meet the requirements of a metasystem.

7.3.2 Request and Offer Description

As a proof of the concept a description language has been developed which is used to specify requests, resources and offers. Our economic scheduling model from Chapter 6 has been implemented with this language.

As we mentioned before, it is essential that requests for offers are very flexible. Our description language allows arbitrary attributes which are specified as key value pairs. Note, that the implementation can deal with new resource types with specific keys that have not been defined yet. Only a few keys are specific for the management and scheduling environment. Resources can be specified with additional keys, as well as requests can form requirements for this keys. All these keys are checked for offer matching. We already discussed, that it is necessary to standardize the description and the parameter keys for known resource types.

The description language is used for requests, as well as for offer and resource description. The syntax is very similar in all cases. We allow complex statements in the formulation of the values. This includes expressions and conditions to allow parameterized attribute specification that can be evaluated at run-time. Examples are the utility function or the job length, which may be derived from other attributes as the job cost or the available processor number/speed.

Next, we want to describe several selected example attributes that are used for our scheduling examination of compute resources. The real set of attributes is larger and specific scheduling methods may utilize more of them. The scheduling for network resources would certainly require additional parameters and an extended scheduling techniques. We first show the sample attributes for the requests; an overview on the resource and offer parameterized are given next. The syntax of the description language is given afterwards with an example to further illustrate the usage.

Request Attributes:

The following is a list of parameters that are available for the requesting formulation:

Hops: This attribute limits the query depth to remote domains. A MetaManager only forwards a request to other domains if the number of hops is not exceeded.

RequestID: a unique number that denotes the current request.

MaxOfferNumber: This is the number of offers a user wants to receive for his request. The value 0 specifies that the MetaManager should automatically select and allocate the best offer according to the UtilityValue.

MinMulti-Site: This specifies a minimum number of available resources before a job is split up during multi-site execution. This prevents too small jobs from being started in multi-site mode.

MaxMulti-Site: This specifies a maximum number of available resources that is required in a domain before a job is split up during multi-site execution. This value can lead to the consequence that a job must be executed in multi-site mode.

OfferBudget: This specifies the budget that is available for offer generation. Note, that this is not the budget for the actual job execution, but for the offer generation process which may require additional costs as well.

ReservationTime: This is the time until which the available resources should be reserved.

StartTime: A job must not be started before this date.

EndTime: A job must not end after this date.

OperationSystemEqual: This parameter specifies that all resources must have the same operating system (=1) if executed in multi-site mode. If the value is 0, the different partitions of a multi-site job can consist of different operating systems which requires that the job supports different architectures.

SearchTime: The scheduling system can search for offers until this absolut time instance.

JobBudget: This parameter specifies the maximum execution cost.

ReservationBudget: This parameter specifies the maximum reservation cost.

RunTime: This parameter specifies the execution time of a job. This definition can be done relative to the speed index of resources.

UserName: This parameter specifies uniquely the submitting user.

Memory: This is the memory requirement per processor (in kBytes).

OperatingSystem: This specifies the requested operating sytem type.

NumberOfProcessors: This is the number of requested resources.

UtilityValue: This value denotes the marginal gain from the user's point of view.

Resource Attributes:

Resources are defined in a similar way as requests and offers. The following is a list of parameters that are available for the resource description:

ResourceID: This is a unique identifier for a resource.

StartTime: This is the date after which the resource is available for grid usage.

EndTime: The resource is withdrawn from grid usage at this date.

NumberofProcessors: The specifies the available nodes of the resource.

Memory: This specifies the memory available in each node (in kBytes).

OperatingSystem: This is the OS type of the available resource.

Objective: This formula denotes the owner's utility function which can be evaluated for a resource allocation.

OfferCost: This is the cost for generating the offer.

ReservationCost: This is the cost for reserving the offer.

Offer Attributes

The MetaManager receives the requests and has information on the resources above. Consequently, it generates suitable offers for the request. Therefore, the request and the resource information is combined into the offer. Following is a list of parameters that are specific the offer description:

OfferID: This is a unique specifier for the offer.

RequestID: This is a specifier for the corresponding request.

JobCost: This is the cost for the job execution.

OfferCost: This specifies the cost for the offer generation.

ReservationCost: This specifies the cost for the offer reservation.

ReservationTime: This parameter is the date until which the offer is reserved.

StartTime: This is the start time of the job.

EndTime: This is the end time of the job.

UtilityValue: This is the calculated value of the user utility function.

Description Language Syntax

Key element of the request and offer description is the ability to formulate flexible statements on the required resource as well as variable objective functions. The language allows arbitrary key value pairs. As mentioned before this is not a flat list of attributes. Moreover, complex statements are allowed for expressions and conditions.

The formal syntax of the language is given in Appendix A. It is presented in BNF-format which can be used, for instance, in implementing the parser with the common Yacc utility.

In the following section an example is given to illustrate the syntax. The main syntax element is the assignment from a value to a key. An *expression* is the simplest element of the syntax. It can represent a number as well as a string. As a string it can be a combination with another expression which is later evaluated to determine the actual value. Here, several operators are allowed like e.g. addition, substraction, multiplication, division and several other more complex operations like modulo, square-root, exponent or logarithmic functions. The expression is used for mathematical calculations like e.g. for the objective function.

A *condition* in combination with a *simplecondition* is used for conditional statements with logical operators. The common AND, OR, NOT operators are available. Additionally, there are the condition statements ISDEF and ISNDEF which check for the existence of attributes. As not all attributes are defined for each resource and we also allow arbitrary resource types, these operators allow the check if a specific key exists. This can be used for example if a completely different utility function should apply if a specific key exists. Furthermore, conditions can be combined to more complex statements.

The other syntax rules are used for simple values, lists and concatenation of other definitions.

Example

To better illustrate the description language, we give a brief example for a request formulation:

```
REQUEST "Req001" {
  KEY "Hops" {VALUE "HOPS" {2}}
  KEY "MaxOfferNumber" {VALUE "MaxOfferNumber" {0}}
  KEY "MinMulti-Site" {VALUE "MinMulti-Site" {2}}
  KEY "MaxMulti-Site" {VALUE "MaxMulti-Site" {5}}
  KEY "OfferBudget" {VALUE "OfferBudget" {100.00}}
  KEY "ReservationTime" {VALUE "ReservationTime" {900008}}
  KEY "StartTime" {VALUE "StartTime" {900000}}
  KEY "EndTime" {VALUE "EndTime" {900028}}
  KEY "OperationSystemEqual" {VALUE "OperationSystemEqual" {0}}
  KEY "SearchTime" {VALUE "SearchTime" {899956}}
  KEY "JobBudget" {VALUE "JobBudget" {900.89}}
  KEY "ReservationBudget" {VALUE "ReservationBudget" {7.56}}
  KEY "Utility" {
     ELEMENT 1 {
       CONDITION{
         ((OperationSystem EQ "Linux") || (OperationSystem EQ "UNIX"))
         && ((NumberOfProcessors >= 8) && (NumberOfProcessor <= 32))
                }
       VALUE "UtilityValue" {-StartTime}
       VALUE "RunTime" {5}
       VALUE "memory" {NumberOfProcessors * 100}
               }
     ELEMENT 2 {
       CONDITION{
         ((memory >= 360) && (memory <= 580))
         && ((NumberOfProcessors >= 8) && (NumberOfProcessor <= 32))
                }
       VALUE "UtilityValue" {-JobCost}
       VALUE "RunTime" {9}
               }
        }
}
```

Within request „Req001" some simple assignments are made for the keys *Hops* to *ReservationBudget*.
Next, two possible environments for job execution are defined for the key *Utility*. The first environment requires **Linux** or **UNIX** for the operating system and needs between 8 and 32 nodes. For this condition the (*UtilityValue*) is defined to start jobs as soon as possible by the definition - *StartTime*. The job has an execution time of 5 seconds and a memory requirement per node of 100 KBytes.
A second environment requires between 360 kByte and 580 kByte as well as between 8 and 32 nodes. For this environment a different utility function is selected which minimizes the job cost. Also the run-time of the job is now 9 seconds.

7.3.3 Resource Determination

A *request* is generated by an application or user. It is submitted to the *MetaManager* in the local *MetaDomain*. Then the *MetaManager* tries to find suitable resources and makes allocations if requested. The search for a matching resource set always starts in the local *MetaDomain*. Giving priority to local resources typically leads to shorter response time, less network load and may result in shorter startup times. The *MetaManager* may split up requests into more detailed requests which are then used to determine if a specific resource fits the requirements. Moreover, the scheduler connects to known remote *MetaManagers* and forwards requests to them. This is a peer-to-peer communication pattern, where central instances are not necessary. Nevertheless, additional directory and information services for faster retrieval of resources can easily be introduced and utilized.

The reply to a request is a resource offer. Note, that there may be several possible allocations that fit a request. Therefore several offers may be created. Even a single resource can provide several offers which differ in their attributes. Note that every *MetaManager* can ask any other *MetaManager* for offers. To prevent network flooding we therefore limit the scope of requests by special attributes: maximum number of hops to search, maximum number of collected offers etc. The whole peer-to-peer search mechanism is also similar to the trading service as defined in the CORBAservices [55]. The performance impact for searching is usually limited; the *MetaManager* does not query the whole network. Optimization with caching and lookup-strategies further minimizes the number of involved *MetaManagers*. Note that a suitable configuration of links between *MetaManagers* in this trading scheme allows to utilize network locality. For instance, *MetaManagers* should be linked if the corresponding counterparts are connected by a network with high bandwidth and low latency.

Each trading step can generate a set of resource offers. Depending on the request attributes these offers may only be valid for a specific time. Further, obtaining and reserving an offer may already result in costs. The *MetaManager* is responsible to guarantee the validity of its offers. In order to find possible allocations for a request a scheduler may combine several offers. Finally, it is up to the scheduler or the requesting user to accept an allocation.

7.3.4 Scheduling and Allocation

The generation of offers depends on the local scheduling policy. Administrators can use different strategies for different resources. Overall, our infrastructure supports the implementation of conventional algorithms as well as economic models. Our scheduling models as the economic methods presented in Chapter 6 have been implemented in NWIRE. The mechanism by using requests and offers provides a flexible interface that allows different scheduling policies to cooperate without exposing too much information to remote sites and delegating control to remote schedulers.

In our economic approach the selection of the best suited allocation is based on a comparison of the provided objective functions as we have shown in Chapter 6. The objective function of a request is applied to an allocation in order to generate a value for the utility from the user's point of view. Similarly, the offers also provide an objective function or a value for its utility to represent the resource's point of view. This also represents the actual cost for the allocation. The responsible *MetaManager* evaluates the user utility function and determines an allocation that maximizes the objective, this can also be done with respect to the overall objective of its full schedule. It is also possible to provide the user with a front-end that allows interactive selection of allocations. Such a front-end can also be used to obtain status information about the metasystem with the help of the request mechanism. This information may help a user to generate a request which results in the best suited set of resources depending on the current condition in the metasystem.

Our schedulers select allocations that optimize the utility value for the particular scheduling time instance. However, the selected allocation must not necessarily be executed. The *MetaManager* can maintain a schedule with all current allocations in its domain. Its scheduler is free to modify the current schedule at any time. Future implementations of scheduling strategies in the MetaManager can therefore try to improve the current schedule. However, changes in the current scheduling would only be allowed as long as they do not violate any guarantees that have been given for current jobs. Requested guarantees are additional constraints that limit future requests for rescheduling. This procedure of 'rescheduling' can be used to improve the current schedule or to cope with resource failures or cancellations of jobs. The rescheduling requires new requests for offers if other allocations are not active anymore. Note that a valid schedule still exists at every moment. Also, there is a tentative schedule available for all jobs after a scheduling step. A user can monitor the estimated job schedule. This is a feature that only a few job scheduling systems provide, an example is the CCS scheduling system [56].

Any improvements to the schedule can be measured in regard to the utility value. To this end, the scheduler attempts to maximize the overall objective value of the schedule. As an objective function is provided for every request, there is a combined objective function or value for each allocation. The objective functions of all allocations together define the optimization problem for the scheduler. An improvement can be achieved for instance by moving existing allocations while all constraints are observed. Alternatively, the scheduler can look for new allocations. The NWIRE scheduling concept furthermore supports multi-site scheduling and co-allocation as we have shown in our economic model.

There is also the foundation to include network resources as just another manageable resource to provide guaranteed communication bandwidth between participating resources. While this scheduling strategy does not guarantee an optimal schedule in general, it meets all requirements of Section 7.1 as separate objectives are allowed for each resource in the metasystem.

The following example illustrates the scheduling architecture. Assume a user who submits at noon a job with the following specific requests to a metasystem managed by NWIRE: 16 nodes of type Power3 with operating system AIX 4.3.1. and at least 8 GByte of total main memory. The execution time of the job will not exceed 1 hour and the user would like to obtain the results as soon as possible. Unfortunately, there are no AIX nodes on the local system. Therefore, the local *MetaManager* forwards the request to other known *MetaManagers*. If the computing system attached to such a *MetaManager* is able to provide the required resources and also permits access of the requesting user, then its *Meta Manager* generates an offer, temporarily reserves the resources and sends the offer to the first *MetaManager*. In addition, *MetaManagers* may further forward the request. The validity of those offers will typically expire after some time. Assume in our example that there are two offers: The first offer provides 8 GByte memory and guarantees completion of the job by 3 pm. The other offer does not include any information about the completion time but allows the use of 16 GByte. Based on the criterion of the user the first *MetaManager* selects the first offer, informs the user, and notifies the *MetaManagers* that generated the offers. If a scheduling criterion is not detailed enough or cannot be completely met, the *MetaManager* may also leave the final decision up to the user. After acceptance of its offer a local *ResourceManager* can rearrange the schedule as long as the conditions of the original offer are observed. For instance, in our case the job must be started no later than 2 pm to obey the given guarantee of a completion by 3 pm. Note that the process of data and code submission is not addressed in this simple example.

7.4 Summary

We presented a scheduling concept for a metacomputing environment. This method is based on a brokerage and trading approach and provides a high degree of flexibility. For instance, administrators of resources in the metasystem are still able to apply arbitrary scheduling and allocation strategies for their resource. The system also supports guarantees, reservation and multi-site applications. Similarly, a dedicated objective can be defined for each request by a user. An additional feature is the instant feedback to a request. This allows the user to adapt his job requirements and to explore the metasystem with informal requests. He can also monitor the planned allocation and execution time of his job at any time.

This implementation has been used for our grid scheduling models as the presented economic approach in Chapter 6. Different scheduling policies and algorithms can easily be implemented. Nevertheless, different schedulers can interoperate in our infrastructure with respect to the presented request and offer mechanisms. The communication is done by a description language for requests and offers and may use different communication interfaces in comparison to our example implementation that used CORBA. This may include for example RMI or SOAP web services.

Note, that NWIRE has been designed as a proof of concept and tool for analyzing distributed scheduling. This is not an implementation of a full grid management software, which requires more additional services. Nevertheless, the presented concept provide several features that are currently not available in any of the current initiatives. The infrastructure can be easily adapted to be incorporated into other grid projects. Suitable scheduling models have been presented in the previous chapters. Especially our new economic scheduling method can be implemented in this infrastructure as we have proven. Also the scheduling method provides the discussed features, such as for example the automatic co-allocation of resources, support for variable objective functions, support for guarantees etc., which are currently not available in the mentioned grid initiatives, as e.g. Globus.

Chapter 8

Conclusion

High Performance Computing is an important tool for research and development in many different areas. Originally, supercomputers were mainly used to address problems in physics. Today many research fields require access to suitable computing resources. High performance computer equipment is essential for e.g. the design of new drugs, an accurate weather forecast or the creation of new movies. Other new applications especially in the field of education are currently under development.

Grid computing is intended to offer an easy and seamless access to such resources. The grid environment should be capable of managing the access to available resources. User jobs have to be assigned automatically to suitable resources. Ideally, the user needs no further knowledge on the current state and configuration of grid resources. The submission of jobs should be as easy as accessing a local resource. Overall, it is expected that in terms of quantity more resources are available in the grid. There is also the expectation that a larger number of different resource types are available in the grid. This allows to assign more suitable resources for a computational problem than locally accessible by the user. These resources may be more efficient for the user's application. This requires a flexible and efficient scheduling strategies. The purpose of this work was the design and evaluation of such a scheduling system for grid environments. There exists a lot of research on algorithms for single autonomous parallel machines. We therefore started our examination with these well-known algorithms. We presented a concept for the design process of a scheduling system. We were especially interested in strategies that produce 'good' results in real implementations. Therefore, we think that the design of a scheduling system can only be done if the scheduling policy of the machine owner as well as the actual user's workload profile on that machine configuration is taken into account. Unfortunately, most scheduling problems are computationally NP-complete and proved to be very hard. This makes it almost impossible to achieve optimal scheduling results. Therefore, the algorithm designer has to settle for less. Consequently, the evaluation of scheduling algorithms becomes a very important task.

The evaluation of scheduling algorithms can be done theoretically and by experiments. We presented a new on-line parallel job scheduling method which adheres to a notion of *fairness* in regards of user satisfaction. We theoretically examined the algorithm by competitive analysis that leads to bounds for the worst-case scenario. However, we also discussed that the theoretical approach does not consider the workload and the actual user demand. The worst-case scenario is usually of limited relevance for a real implementation as it may occur very seldom. Additionally, it is often difficult or maybe impossible to find theoretical bounds for complex algorithms. This led to the use of experimental analysis. As real experiments are often to expensive and not applicable, simulations are used for evaluation. In addition to the theoretical analysis, we showed that the presented preemptive scheduling algorithm (PFCFS) is also efficient in terms of makespan and average response time in comparison to other scheduling algorithms. In some of the examined scenarios the algorithm could outperform common algorithms such as backfilling.

Based on these results, we presented scheduling algorithms for the grid environment. These methods based on modifications of the examined conventional strategies for single parallel machines. The results showed that: a) it is beneficial for the average response time of local system users if the machines of different sites participate in a computational grid; b) the multi-site computing paradigm improves the result even if it results in some overhead on the execution time (50%).

The requirements for grid computing differs significantly from single parallel machine computing. We discussed especially the fact that the machines belong to different owners or administrative instances and that the machines are usually not solely dedicated to grid usage. As a result this leads to the requirement that variable scheduling policies should be taken into account – for the owner as well as the user of the environment. Therefore, the application of economic models has been considered as they provide support for cost models and utility functions. To this end, we presented an economic scheduling model suitable for grid computing. The economic approach is an example for a scheduling strategy where theoretic analysis is especially difficult to perform. We used the experimental analysis by simulating workload models in the same way as we did with the other scheduling strategies. We assumed that the scheduling objective is the minimization of the response time. Although this need not necessarily be the case for the economic method, this allowed the comparison to our conventional scheduling strategy. In our study we examined how the economic method performs in comparison to the conventional scheduling methods under the assumption which requires that the response time is the overall objective in this scenario. To this end utility functions for user and owners had to be selected for this overall objective. The results with first selections of this utility functions are very promising for the economic scheduling model. Although we do not have theoretical bounds on the scheduling quality, the simulation showed that we are able to get results in the same range or even better than the conventional strategies. However, as we mentioned before the economic models has several advantages as it supports different scheduling policies, cost models etc.

A proof of concept has been given by the NWIRE architecture, that economic scheduling strategies can actually be implemented for grid usage. Here, we showed a scheduling infrastructure that supports the presented economic model. It supports a scheduling infrastructure that is flexible enough to scale with larger grid environments. In a peer-to-peer fashion the infrastructure does not need central services but still can take advantage of them. Additionally, the scheduling interfaces support different scheduling policies at each participating site. More features have also been discussed in this work.

It has to be noted, that the quality of a scheduling system closely depend on the actual workload on that system. The evaluation can only be done for a certain scenario. We therefore used an actual workload and some derivations to our simulation studies. The traces were taken from the IBM RS/6000 SP from the Cornell Theory Center. Our machine model was selected to resemble such an IBM RS/6000 SP. Before applying the results to other configurations it is necessary to verify the results by additional analysis and adapt the strategy.

Overall, our scheduling infrastructure is not a full implementation of a full grid management software, which requires many additional services. Nevertheless, the presented concept provides several features that are currently not available in any of the current initiatives like e.g. Globus. The infrastructure can be easily adapted to be incorporated into other grid projects. We presented several suitable scheduling models and gave evaluation results in this work. Especially our new economic scheduling method can be implemented for this infrastructure as shown. The scheduling method provides the discussed features which are currently not available in the scheduling components in the mentioned initiatives for grid environments.

Future Work

Future work should include a thorough parameter study to optimize the utility and machine functions of the economic scheduling model. In our work we used first example functions that seemed suitable. We believe that the results can be improved further by selecting more sophisticated functions.

The presented scheduling model allows the inclusion of network or data resources as manageable components. Future work is necessary to support these resource types and include them accordingly into the scheduling process.

In this work, we discussed the importance of experimental evaluation. Simulation with workload derived from a real traces has been used for our evaluation. For the design process for a scheduling system it is important that the workload model used for evaluation resembles the user group on the real implementation. We therefore used real traces from the CTC as the base of our workload model. Nevertheless, we always loose validity of this model as it is typically assumed, that the actual scheduling process has no impact on the job submission. This is obviously not true, as people usually see the current utilization of a machine or the quality of service they get from previous job submissions. Additionally, the response time of previously submitted jobs will influence the submission of future jobs. Therefore, there is always some deviation from real system if workloads are replayed on a different system. This deviation may be small and neglectable if the scheduling policy and method lead to vastly similar results. Nevertheless, future work should include some kind of user modelling that considers specific user profiles and submission processes. Such a model can be used to better adapt the workload to different machine configuration, e.g. for different machine configurations.

The proposed models can be applied to current grid computing environments. Momentarily, the Grid Global Forum proposes a draft for the Open Grid Service Architecture (OGSA) for extending current compatible grid environments. Work is currently underway to support OGSA by the Globus project. In future work our models can be adapted and implemented to support the OGSA standard. This is a promising way to contribute to current initiative and introduce the proposed models to broad practical usage.

While the application for grid computing allows the practical study and research on core topics like scheduling, it can be easily extended to other applications. Our results are especially usable in scenarios in which distributed resources which are owned and controlled by independent individuals must cooperate to solve a problem. The automatic allocation of resources by the presented scheduling strategies and the proposed peer-to-peer environment provides a reliable architecture and efficient strategies. Therefore, the results are not limited to grid computing as the same concepts apply to other application fields. Logistics, telecommunications or mobile devices come into mind.

Bibliography

[1] F. Berman, R. Wolski, S. Figueria, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing*, 1998.

[2] C. Bitten, J. Gehring, U. Schwiegelshohn, and R. Yahyapour. The nrw-metacomputer building blocks for a worldwide computational grid. In *Proceedings of the IPDPS 2000*, May 2000.

[3] N. Bogan. Economic allocation of computation time with computation markets. In *Master Thesis.* Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1994.

[4] M. Brune, J. Gehring, A. Keller, and A. Reinefeld. Managing clusters of geographically distributed high-performance computers. *Concurrency - Practice and Experience*, 11(15):887–911, 1999.

[5] J. Bruno, E.G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17:382–387, 1974.

[6] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in grid computing. *Special Issue on Grid Computing Environments, The Journal of Concurrency and Computation: Practice and Experience (CCPE)*, May 2002(accepted for publication).

[7] R. Buyya, J. Giddy, and D. Abramson. An evaluation of economy-based resource trading and scheduling on computational power grids for parameter sweep applications. In *The Second Workshop on Active Middleware Services (AMS 2000), In conjuction with Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC 2000)*, Pittsburgh, USA, August 2000. Kluwer Academic Press.

[8] S. Chakrabarti, C. Phillips, A.S. Schulz, D.B. Shmoys, C. Stein, and J. Wein. Improved approximation algorithms for minsum criteria. In F. Meyer auf der Heide and B. Monien, editors, *Proceedings of the 1996 International Colloquium on Automata, Languages and Programming*, pages 646–657. Springer–Verlag Lecture Notes in Computer Science LNCS 1099, 1996.

[9] S.-H. Chiang, R.K. Masharamani, and M.K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of ACM SIGMETRICS Conference on Measurement of Computer Systems*, pages 33–44, 1994.

[10] K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Job Scheduling Strategies for Prallel Processing*, volume 1459 of *Lecutre Notes in Computer Science*, pages 62–68. Springer Verlag, 1998.

[11] X. Deng, N. Gu, T. Brecht, and K. Lu. Preemptive scheduling of parallel jobs on multiprocessors. In *Proceedings of the 7^{th} SIAM Symposium on Discrete Algorithms*, pages 159–167, January 1996.

[12] R. Diekmann, B. Monien, and R. Preis. Load balancing strategies for distributed memory machines. *Multi-Scale Phenomena and Their Simulation*, pages 255–266, 1997.

[13] J. Du and J. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, November 1989.

[14] European grid forum, http://www.egrid.org, October 2001.

[15] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. On Advantages of Grid Computing for Parallel Job Scheduling. In *Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CC-GRID 2002), Berlin*, pages 39–46, 2002.

[16] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Enhanced Algorithms for Multi-Site Scheduling. In *Proceedings of the 3rd International Workshop on Grid Computing, Baltimore*. Springer–Verlag, Lecture Notes in Computer Science LNCS, 2002 - to appear.

[17] C. Ernemann, V. Hamscher, U. Schwiegelshohn, and R. Yahyapour. Economic scheduling in grid computing. In D.G. Feitelson and L. Rudolph, editors, *HPDC 2002 Workshop: Job Scheduling Strategies for Parallel Processing*. Springer–Verlag, Lecture Notes in Computer Science LNCS, 2002 - to appear.

[18] C. Ernemann, V. Hamscher, A. Streit, and R. Yahyapour. On effects of machine configurations on parallel job scheduling in computational grids. In *International Conference on Architecture of Computing Systems, ARCS*, pages 169–179, 2002.

[19] D.G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Research report rc 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY, February 1995.

[20] D.G. Feitelson. Packing schemes for gang scheduling. In D.G. Feitelson and L. Rudolph, editors, *IPPS'96 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 89–110. Springer–Verlag, Lecture Notes in Computer Science LNCS 1162, 1996.

[21] D.G. Feitelson. Online Parallel Workloads Archive. Web-Archive, 1998. http://www.cs.huji.ac.il/labs/parallel/workload/.

[22] D.G. Feitelson and M.A. Jette. Improved utilization and responsiveness with gang scheduling. In D.G. Feitelson and L. Rudolph, editors, *IPPS'97 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 238–261. Springer–Verlag, Lecture Notes in Computer Science LNCS 1291, 1997.

[23] D.G. Feitelson and B. Nitzberg. Job characteristics of a production parallel scientific workload on the nasa ames ipsc/860. In D.G. Feitelson and L. Rudolph, editors, *IPPS'95 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 337–360. Springer–Verlag, Lecture Notes in Computer Science LNCS 949, 1995.

[24] D.G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In *IPPS'95 Workshop: Job Scheduling Strategies for Parallel Processing, pages 1-18. Springer Verlag, Lecture Notes in Computer Science LNCS 949, 1995.*, 1995.

[25] D.G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In D.G. Feitelson and L. Rudolph, editors, *IPPS'95 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 1–18. Springer–Verlag, Lecture Notes in Computer Science LNCS 949, 1995.

[26] D.G. Feitelson and L. Rudolph. Towards convergence in job schedulers for parallel supercomputers. In D.G. Feitelson and L. Rudolph, editors, *IPPS'96 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer–Verlag, Lecture Notes in Computer Science LNCS 1162, 1996.

[27] D.G. Feitelson and L. Rudolph. Metrics and benchmarking for parallel job scheduling. In D.G. Feitelson and L. Rudolph, editors, *IPPS'98 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 1–24. Springer–Verlag, Lecture Notes in Computer Science LNCS 1459, 1998.

[28] D.G. Feitelson, L. Rudolph, U. Schwiegelshohn, K.C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In D.G. Feitelson and L. Rudolph, editors, *IPPS'97 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer–Verlag, Lecture Notes in Computer Science LNCS 1291, 1997.

[29] D.G. Feitelson and A.M. Weil. Utilization and Predictability in Scheduling the IBM SP2 with Backfilling. In *Procedings of IPPS/SPDP 1998*, pages 542–546. IEEE Computer Society, 1998.

[30] A. Feldmann, J. Sgall, and S.-H. Teng. Dynamic scheduling on parallel machines. *Theoretical Computer Science*, 130:49–72, 1994.

[31] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40(1):35–48, jan 1997.

[32] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the I-WAY high performance distributed computing experiment. In *Proceedings $5^t h$ IEEE Symposium on High Performance Distributed Computing*, pages 562–571, 1997.

[33] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Application*, 11(2):115–128, 1997.

[34] I. Foster and C. Kesselman. The globus project: A status report. In *IPPS/SPDP'98 Workshop: Heterogenous Computing Workshop*, volume 11, pages 4–18, 1998.

[35] I. Foster and C. Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 1998.

[36] M. Garey and R.L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, June 1975.

[37] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[38] GENIAS Software GmbH. *CODINE 4.2*, 1997. http://www.genias.de/.

[39] The Grid Forum, http://www.gridforum.org, October 2001.

[40] A. Grimshaw, A. Wulf, J. French, A Weaver, and P. Reynolds. Legion: The next logical step toward a nationwide virtual supercomputer. Technical Report CS-94-21, University of Virginia, Computer Sciences Department, 1994.

[41] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Evaluation of job-scheduling strategies for grid computing. *Lecture Notes in Computer Science*, 1971:191–202, 2000.

[42] J.L. Hennessy and D.A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, San Francisco, second edition, 1996.

[43] J. Heuer. Evaluierung und Implementierung von marktorientierten Verteilungsverfahren fr Metacomputing. In *Diploma Thesis at CEI.* University of Dortmund, Germany, 2000.

[44] S. Hotovy. Workload evolution on the Cornell Theory Center IBM SP2. In D.G. Feitelson and L. Rudolph, editors, *IPPS'96 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 27–40. Springer–Verlag, Lecture Notes in Computer Science LNCS 1162, 1996.

[45] Jack Dongarra and Hans Meuer and Erich Strohmaier. Top 500 Report. WWW Page, 2001. http://www.netlib.org/benchmark/top500/top500.list.html.

[46] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riordan. Modeling of workload in mpps. In D.G. Feitelson and L. Rudolph, editors, *IPPS'97 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 94–116. Springer–Verlag, Lecture Notes in Computer Science LNCS 1291, 1997.

[47] T. Kawaguchi and S. Kyan. Worst case bound of an LRF schedule for the mean weighted flow-time problem. *SIAM Journal on Computing*, 15(4):1119–1129, November 1986.

[48] J. Krallmann, U. Schwiegelshohn, and R. Yahyapour. On the Design and Evaluation of Job Scheduling Systems. In D.G. Feitelson and L. Rudolph, editors, *IPPS/SPDP'99 Workshop: Job Scheduling Strategies for Parallel Processing.* Springer–Verlag, Lecture Notes in Computer Science, 1999.

[49] R.N. Lagerstrom and S.K. Gipp. PscheD: Political Scheduling on the CRAY T3E. In D.G. Feitelson and L. Rudolph, editors, *IPPS'97 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 117–138. Springer–Verlag, Lecture Notes in Computer Science LNCS 1291, 1997.

[50] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the 29th ACM Symposium on the Theory of Computing*, pages 110–119, May 1997.

[51] D.A. Lifka. The ANL/IBM SP Scheduling System. In D.G. Feitelson and L. Rudolph, editors, *IPPS'95 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer–Verlag, Lecture Notes in Computer Science LNCS 949, 1995.

[52] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th Intl Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[53] M. Livny and R. Raman. High-Throughput Resource Management. In I. Foster and C. Kesselman, editors, *The Grid - Blueprint for a New Computing Infrastructure*, pages 311–337. Morgan Kaufmann, 1999.

[54] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12, October 1959.

[55] Object Management Group Document. *CORBAservices: Common Object Services Specification*, 1998. 98-07-06.

[56] F. Ramme. Building a virtual machine–room - a focal point in metacomputing. *Future Generation Computer Systems, Special Issue on HPCN*, 11:477–489, aug 1995.

[57] M.E. Rosenkrantz, D.J. Schneider, R. Leibensperger, M. Shore, and J. Zollweg. Requirements of the Cornell Theory Center for Resource Management and Process Scheduling. In D.G. Feitelson and L. Rudolph, editors, *IPPS'95 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 304–318. Springer–Verlag, Lecture Notes in Computer Science LNCS 949, 1995.

[58] W. Saphir, L.A. Tanner, and B. Traversat. Job Management Requirements for NAS Parallel Systems and Clusters. In D.G. Feitelson and L. Rudolph, editors, *IPPS'95 Workshop: Job Scheduling Strategies for Parallel Processing*,

pages 319–337. Springer–Verlag, Lecture Notes in Computer Science LNCS 949, 1995.

[59] U. Schwiegelshohn. Preemptive weighted completion time scheduling of parallel jobs. In *Proceedings of the 4th Annual European Symposium on Algorithms (ESA96)*, pages 39–51. Springer–Verlag Lecture Notes in Computer Science LNCS 1136, September 1996.

[60] U. Schwiegelshohn, W. Ludwig, J.L. Wolf, J.J. Turek, and P. Yu. Smart SMART bounds for weighted response time scheduling. *SIAM Journal on Computing*, 28(1):237–253, January 1999.

[61] U. Schwiegelshohn and R. Yahyapour. Analysis of First-Come-First-Serve Parallel Job Scheduling. In *Proceedings of the 9th SIAM Symposium on Discrete Algorithms*, pages 629–638, January 1998.

[62] U. Schwiegelshohn and R. Yahyapour. Improving first-come-first-serve job scheduling by gang scheduling. In D.G. Feitelson and L. Rudolph, editors, *IPPS'98 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 180–198. Springer–Verlag, Lecture Notes in Computer Science LNCS 1459, 1998.

[63] U. Schwiegelshohn and R. Yahyapour. The NRW Metacomputing Initiative. In G. Cooperman, E. jessen, and G. Michler, editors, *Workshop on Wide Area Networks and High Performance Computing*, pages 269–282. Springer–Verlag, Lecture Notes in Control and Information Science LNCIS 249, 1998.

[64] U. Schwiegelshohn and R. Yahyapour. Resource allocation and scheduling in metasystems. In *Distributed Computing and Metacomputing*, volume 1593 of *Lecture Notes in Computer Science*, pages 851–860. Springer, april 1999.

[65] U. Schwiegelshohn and R. Yahyapour. Fairness in parallel job scheduling. *Journal of Scheduling, 3(5):297-320. John Wiley*, 2000.

[66] D. Shmoys, J. Wein, and D. Williamson. Scheduling parallel machines on-line. *SIAM Journal on Computing*, 24(6):1313–1331, December 1995.

[67] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY — LoadLeveler API Project. *Lecture Notes in Computer Science*, 1162:41–47, 1996.

[68] L. Smarr and C.E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, June 1992.

[69] W. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[70] C. Stein and J. Wein. On the existence of schedules that are near-optimal for both makespan and total weighted completion time. *Preprint*, 1996. To appear in *Operations Research Letters*.

[71] R.E. Steuer. *Multiple Criteria Optimization, Theory, Computation and Application.* Wiley, New York, 1986.

[72] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1):48–63, 1996.

[73] A. Su, F. Berman, R. Wolski, and M. Strout. Using apples to schedule a distributed visualization tool on the computational grid. In *Proceedings of the 1998 Cluster Computing Workshop*, Blackberry Farm, Tennessee, 1998.

[74] Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload/, October 2001.

[75] K. R. Grant T. W. Malone, R. E. Fikes and M. T. Howard. Enterprise: A market-like task scheduler for distributed computing environments. In *The Ecology of Computation*, volume 2 of *Studies in Computer Science and Artifical Intelligence*, pages 177–255, 1988.

[76] P. Tucker. Market mechanisms in a programmed system. Department of Computer Science and Engineering, University of California, 1998.

[77] P. Tucker and F. Berman. On market mechanisms as a software technique. Technical Report CS96-513, University of California – San Diego, Department of Computer Science and Engineering, December 1996.

[78] J.J. Turek, W. Ludwig, J.L. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schwiegelshohn, and P. Yu. Scheduling parallelizable tasks to minimize average response time. In *Proceedings of the 6th Annual Symposium on Parallel Algorithms and Architectures, Cape May, NJ*, pages 200–209, June 1994.

[79] J.J. Turek, U. Schwiegelshohn, J.L. Wolf, and P. Yu. Scheduling parallel tasks to minimize average response time. In *Proceedings of the 5^{th} SIAM Symposium on Discrete Algorithms*, pages 112–121, January 1994.

[80] J.J. Turek, J.L. Wolf, and P. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures, San Diego, CA*, pages 323–332, June 1992.

[81] C.A. Waldspurger, T. Hogg, B. Huberman, J.O. Kephart, , and W.S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103– 117, 1992.

[82] W. Walsh, M. Wellman, P. Wurman, and J. MacKieMason. Some economics of market-based distributed scheduling. In *In Eighteenth International Conference on Distributed Computing Systems*, pages 612–621, 1998.

[83] F. Wang, H. Franke, M. Papaefthymiou, P. Pattnaik, L. Rudolph, and M.S. Squillante. A gang scheduling design for multiprogrammed parallel computing environments. In D.G. Feitelson and L. Rudolph, editors, *IPPS'96 Workshop:*

Job Scheduling Strategies for Parallel Processing, pages 111–125. Springer–Verlag, Lecture Notes in Computer Science LNCS 1162, 1996.

[84] F. Wang, M. Papaefthymiou, and M.S. Squillante. Performance evaluation of gang scheduling for parallel and distributed multiprogramming. In D.G. Feitelson and L. Rudolph, editors, *IPPS'97 Workshop: Job Scheduling Strategies for Parallel Processing*, pages 277–298. Springer–Verlag, Lecture Notes in Computer Science LNCS 1291, 1997.

[85] L.M. Wein and P.B. Chevelier. A broader view of the job-shop scheduling problem. *Management science*, 38:1018–1033, 1992.

[86] F. Ygge. *Market-Oriented Programming and its Application to Power Load Management.* PhD thesis, Department of Computer Science, Lund University, 1998.

[87] D. Zagordono, F. Berman, and R. Wolski. Application scheduling on the information power grid. *International Journal of High-Performance Computing*, 00, March 1999.

[88] S. Zhou. LSF:load sharing in large-scale heterogeneous distributed systems. In *Proceedings Workshop on Cluster Computing*, 1992.

Appendix A

Description Language Syntax in NWIRE

Following is the formal syntax of the request, offer and resource description language as implemented in NWIRE. The syntax is presented in the BNF-format.

Additional information are given in Section 7.3.2.

expression: `a_float_number (e.g. 2.34+E20)`
| `a_string (e.g. memory)`
| `a_long_integer (e.g. 123456789)`
| '−' **expression**
| **expression** '+' **expression**
| **expression** '−' **expression**
| **expression** '∗' **expression**
| **expression** '/' **expression**
| **expression** '%' **expression**
| *SQRT* '(' **expression** ')'
| *EXP* '(' **expression** ')'
| *LOG* '(' **expression** ')'
| '(' **expression** ')'
;

value: *VALUE* '"'`key_string`'"' '{' **expression** '}'
| *VALUE* '"'`key_string`'"' '{' '"'`arbitrary_string`'"' '}'
;

valuelist:**value**
| **valuelist value**
;

condition: *CONDITION* '{' **singlecondition** '}'
;

singlecondition: ’(’ `key_string` ’<’ **expression** ’)’
| ’(’ `key_string` ’>’ **expression** ’)’
| ’(’ `key_string` ’<=’ **expression** ’)’
| ’(’ `key_string` ’>=’ **expression** ’)’
| ’(’ `key_string` ’==’ **expression** ’)’
| ’(’ `key_string` ’! =’ **expression** ’)’
| ’(’ `key_string` ’EQ’ ’”’`arbitrary_String`’”’ ’)’
| ’(’ `key_string` ’NE’ ’”’`arbitrary_String`’”’ ’)’
| ’(’ **singlecondition** ’)’
| **singlecondition** ’&&’ **singlecondition**
| **singlecondition** ’||’ **singlecondition**
| ’(’ ’NOT’ **singlecondition** ’)’
| ’(’ ’ISDEF’ `key_string` ’)’
| ’(’ ’ISNDEF’ `key_string` ’)’
;

elements: **element**
| **elements element**
;
element: *ELEMENT* `long_integer` ’{’ **condition valuelist** ’}’
| *ELEMENT* `long_integer` ’{’ **valuelist** ’}’
;

keys: **key**
| **keys key**
;
key: *KEY* ’”’`key_string`’”’ ’{’ **elements** ’}’
| *KEY* ’”’`key_string`’”’ ’{’ **valuelist** ’}’
;

object: *OFFER* ’”’`object_ID`’”’ ’{’ **keys** ’}’
| *REQUEST* ’”’`object_ID`’”’ ’{’ **keys** ’}’
| *MASCHINE* ’”’`object_ID`’”’ ’{’ **keys** ’}’
;

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text, and a list of references is given.

Ramin Yahyapour

About the Author

Ramin Yahyapour was born April 16, 1972 in Dortmund, Germany. After his A-level graduation (Abitur) at the Stadtgymnasium Dortmund he studied Electrical Engineering at the University Dortmund in 1991. After receiving his diploma in 1996, he joined the Computer Engineering Institute (Lehrstuhl für Datenverarbeitungssysteme) at the University Dortmund as a research assistant of Prof. Dr.-Ing. Uwe Schwiegelshohn.

His research interest lies in parallel job scheduling and especially in its application in grid computing. He worked in the NRW metacomputing project for the scheduling task. A selection of his contributions to conferences, workshops or journals can be found in the bibliography of this document.