On-Line Scheduling on Parallel Machines

Jiří Sgall

May 1994
CMU-CS-94-144

Carnegie Mellon

On-Line Scheduling on Parallel Machines

Jiří Sgall

May 1994
CMU-CS-94-144

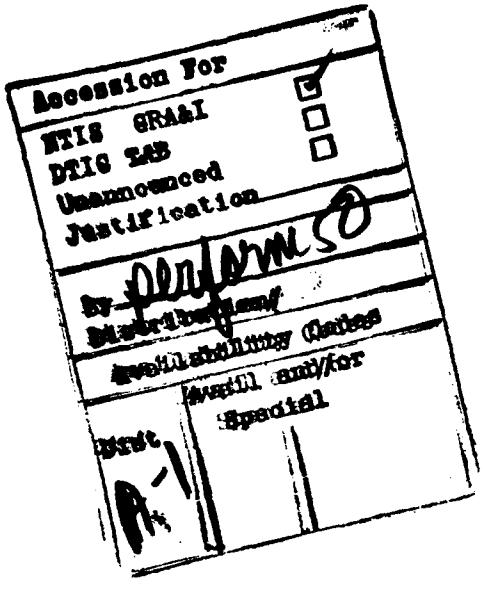
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Comittee:
Steven Rudich, Chair
Daniel D. Sleator
Avrim Blum
Russell Impagliazzo, UC San Diego

This research was sponsored by the National Science Foundation under grant number CCR-9119319.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NSF or of the U.S. Government.



Keywords: competitive analysis, dependencies, network topology, online algorithms, parallel jobs, parallel machines, randomization, scheduling, virtualization.

School of Computer Science

DOCTORAL THESIS
in the field of
Pure and Applied Logic

On-Line Scheduling on Parallel Machines

Jiří Sgall

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

THESIS COMMITTEE CHAIR	5/9/94 DATE
DEPARTMENT HEAD	5/26/94 DATE

APPROVED:

DEAN	5/27/94 DATE

Abstract

Given a parallel machine with processors arranged in some particular network topology (e.g., on a mesh machine the processors are arranged in a rectangular grid), we have to execute different parallel jobs. Each job requires some part of the machine (e.g., a mesh of a smaller size), and can be executed on any subset of processors with that network topology. Each job will run for some fixed time, regardless of when we execute it. But we do not know the running times in advance, the only way to determine the running time of a job is to execute it. Scheduling may also be constrained by dependencies between jobs; it may be the case that a job cannot be started until some other jobs have finished. Our task is to schedule a given set of jobs so that all constraints are satisfied and the total time is as small as possible.

We claim that in this model efficient on-line scheduling is possible on a variety of different parallel machines, including PRAMs, hypercubes and mesh machines. However, the efficiency depends on various factors, including the presence of dependencies, the combinatorial complexity of the network topology, randomization, the use of virtualization, and the maximal size of jobs.

We show that without dependencies, randomized algorithms can achieve a significantly better performance than deterministic ones; on the other hand with dependencies randomization does not help.

The complexity of the network topology has a big influence both with and without dependencies. For more complex networks the optimal performance is significantly worse. Without dependencies, it is to some extent possible to avoid this loss of performance by using more sophisticated algorithms for more complex networks; we show that the greedy method, which is a natural method for on-line scheduling, works very well for simple cases but it is not efficient for more complex machines.

With dependencies, we show that to achieve a good overall performance, it is sometimes essential to use virtualization, i.e., to schedule some jobs on a smaller number of processors than they request, even though it means that their running times increase proportionally. Also limiting the maximal size of jobs improves the performance with dependencies. On the other hand,

without dependencies virtualization and the maximal size of jobs are not important factors.

We also study another model in which we only have sequential jobs (jobs requiring only one processor). As opposed to the other model, they arrive and have to be scheduled one by one in a predetermined order. The running time of a job is known as soon as it arrives, but we have to schedule the job immediately without any knowledge of future jobs. The goal is again to schedule all jobs so that the total time is as small as possible. In this model, we significantly improve the previously known lower bounds on the performance of randomized algorithms.

Acknowledgements

After my arrival at CMU, Steven Rudich invited me to play a game of go against a computer program that "nobody in the department can beat". Once I had lost, he revealed to me that my opponent was not a computer but one of the strongest human players I ever faced; due to his perfect setup I had no chance to discover the trick. Shortly after that we started to talk about a serious research problem at some party. Not having any paper, Steven outlined the problem on my paper plate. Since then, Steven has always been like that—full of enthusiasm and unconventional ideas, extremely open and understanding, and fun to work with. He advised me in professional matters such as research, my presentations and my career, and introduced me to many colleagues; he also helped me in everything else from improving my English to selecting good restaurants.

I am grateful to the coauthors of the papers on scheduling, Anja Feldmann, Shang-Hua Teng and Ming-Yang Kao for their permission to include our joint work in my thesis. Shang-Hua introduced the model of on-line scheduling of parallel jobs, and his enthusiasm was the driving force of this research even after he graduated from CMU. Anja was always willing to listen to the first sketches of my proofs and read the first drafts of them, often hardly comprehensible. Her comments significantly improved the presentation of this thesis.

I would also like to thank to the rest of my committee, Danny Sleator, Avrim Blum and Russell Impagliazzo, for many valuable discussions and encouragement in my work. Discussions with Howard Karloff about on-line scheduling of sequential jobs had a major influence on that part of my thesis. I have also benefited from conversations with Guy Blelloch, Alan Frieze, Garth Gibson, Thomas Gross, Jonathan Hardwick, John Reif, Mahadev Satyanarayanan, Jay Sipelstein, and Jaspal Subhlok. CMU was an exceptional and friendly place to work, and in particular the theory community was very inspiring.

I have learned a lot from Jeff Edmonds, Russell Impagliazzo, Pavel Pudlák, Vojtěch Rödl and Avi Wigderson during our joint work on problems not related to this thesis.

My colleagues from Prague, Jan Krajíček, Pavel Pudlák, Antonín Sochor, and my co-advisor during my first year at CMU, Daniel Leivant, encouraged me to come to CMU when I was not sure that it was the right decision. Mathematical Institute of the Academy of Sciences of the Czech Republic allowed me an extended leave of absence to go to CMU.

From the many other people who made my life here more pleasant I would like to name Alex Ignjatovič, Anja Feldmann, Tom Stricker, Sven Koenig, Peter Jansen, Rachel Rue, Dennis Dancanet and Jeff Shuffelt.

I am grateful to my parents for everything that they taught me, for their guidance and understanding.

But most of all, I thank my wonderful wife Irena for her patience and support during my four years here that have been so long for her.

To my wife and my parents

Contents

List of Tables

List of Figures

1 Introduction

This thesis is a theoretical study of scheduling jobs on parallel machines when only partial information about them is available in advance. We study how the length of the schedule changes under the influence of various factors, such as the use of randomization, the presence of dependencies between jobs, the choice of network topology, the use of virtualization, and the maximal size of jobs. We prove tight or very close upper and lower bounds on the best possible performance of on-line scheduling algorithms for many combinations of these factors, thus giving an essentially complete picture of how they interact and together influence the performance and methods for scheduling.

Our on-line scenario reflects the real life situation where we rarely have full information about the jobs that we wish to schedule; similarly, the factors and network topologies we study are derived from practical concerns. We believe that with the development of larger parallel machines and increase of the amount and variety of applications performed on them, the results of this thesis will become relevant for the choice of the scheduling strategies used in practice.

1.1 On-line scheduling of parallel jobs

Imagine that we have a massively parallel machine with processors arranged in a rectangular grid. This machine is used by a number of users who submit different parallel jobs to be executed on the parallel machine. Each job will run for some fixed time, called its running time, regardless of when we execute it. But we do not know the running times in advance, the only way to determine the running time of a job is to execute it—this is what we call an on-line problem. Our task is to schedule a given set of jobs so that the total time is as small as possible.

Some parallel jobs may make use of all available processors. Other jobs may be able to use only a limited number of processors efficiently; they request a rectangular grid of some smaller size, and can be executed on any subset of processors that form such a grid. We assume that the grid structure has to be preserved because efficient parallel programs are written

for a particular network topology and in general it is impossible to execute them efficiently on a set of processors with a different network topology. We can schedule more than one job, as long as we can satisfy the requirements of all scheduled jobs simultaneously.

Scheduling may also be constrained by dependencies between different jobs. This means that it may be the case that a job cannot be started before some other jobs are finished. In fact, in the on-line problem we might not even know about the existence of a job until all jobs on which it depends are finished. Our schedule has to respect these constraints.

Once a job is started, it has to be completed on the same processors without stopping, it cannot be moved to other processors or stopped and restarted later on the same or different processors.

This problem can be viewed in a very geometric way. We have an empty square representing the machine with processors arranged in a square grid, and a set of rectangles that represent the jobs that have to be scheduled. At the beginning we want to tile the square by the rectangles so that a large fraction of the space is used: already this is a nontrivial problem. But in the on-line setting, and especially in the presence of dependencies, the situation gets much worse. After some jobs finish, the available processors may occupy a region that is much more complex and difficult to tile than the original square; in fact it might be impossible to tile it by the remaining rectangles. It is this interaction of the geometric structure of the machine with the on-line situation that makes the problem difficult.

We study this problem not only for two-dimensional meshes as introduced above, but also for a variety of other network topologies, PRAMs, hypercubes, one- and higher-dimensional meshes. We claim that efficient on-line scheduling is possible for all these topologies, both with and without dependencies. However, the performance depends significantly on various factors.

Not surprisingly, the presence of dependencies has a major influence. Not only is the performance without dependencies significantly better, but also the influence of other factors, discussed below, changes dramatically.

The network topology of the machine is very important both with and

without dependencies. For more complex networks it is necessary to use more sophisticated algorithms and the optimal performance is worse, especially in the presence of dependencies. We show that the greedy method, which is a natural method for on-line scheduling, works very well for simple machines but is not efficient for more complex ones.

Another important factor is whether the algorithm is allowed to make use of randomness. Without dependencies, we give a randomized algorithm for meshes which has much better performance than an optimal deterministic one. We know of only one other result that shows for some model of on-line scheduling that randomization provably improves the performance—namely the result of [BFKV92] on randomized scheduling of sequential jobs on two processors, which we discuss in Section 17. However, in that result the improvement is only by a small constant factor, whereas in our case the improvement is much larger; also our technical tools are different and much more involved.

Surprisingly, in contrast to the randomized algorithm above, we demonstrate that with dependencies randomization cannot improve the performance significantly.

We show that to achieve overall efficiency of scheduling with dependencies, it is essential to schedule some jobs on a smaller number of processors than they request, even though it means that their running times increase proportionally. This technique of running a job on a smaller number of processors than it could use is called virtualization and it is a standard tool in the design of parallel algorithms. Our work shows its importance in another area by demonstrating that it is a necessary and useful technique for efficient on-line scheduling of parallel jobs. If we bound the maximal size of jobs (the requested number of processors), it improves the optimal performance in the presence of dependencies. Without dependencies, neither bounding the size of jobs nor using virtualization has a significant influence.

We are interested in theoretical aspects of this problem, abstracting from some questions that may be important for practical systems. However, the additional costs that we neglect are small, if the jobs do not have extremely short running times, which is usually the case. We assume that the startup

costs are included in the running time of a job, which means that we neglect the fact that these costs may depend on the particular placement of the jobs. We also do not consider the costs of communication between dependent jobs, and between the parallel machine and the users. We do not consider the overhead of the scheduling algorithm; however, all our algorithms are simple and easy to implement even in a distributed environment, hence this overhead is very small.

1.2 Randomized on-line scheduling of sequential jobs

For sequential jobs (jobs that require only one processor) we consider another model, which was studied before in [Gra66, GW93, BFKV92, KPT94].

This model is essentially a modified version of the game of Tetris. We have some fixed number of columns. Rectangles arrive one by one, each of them is one column wide and extends over one or more rows. We have to put each rectangle in one of the columns. The goal is to minimize the total number of rows that are at least partially used by the rectangles.

In this scenario the columns represent the processors, rows represent the time steps and the rectangles represent the jobs. The jobs are sequential, which is represented by the requirement that every rectangle is only one column wide. The running time is represented by the height of a rectangle. The running time is known when a job arrives, unlike in our model for parallel jobs. The jobs arrive one by one, as soon as a job arrives, the scheduler has to assign it to one of the processors.

In this model, the problem is completely solved only for two processors, in which case an optimal randomized algorithm is known and it is better than the optimal deterministic algorithm. For more processors, the best known algorithm is deterministic, which means that we do not know how to make use of randomization at all.

We prove a lower bound on the performance of randomized algorithms in this model, which improves significantly on the previously known bounds for more than two processors. It also gives significant insight about how a randomized algorithm matching this bound has to work. However, we are not able to give such an algorithm at this time.

1.3 Outline

In the first three parts of the thesis we study scheduling of parallel jobs. In Part I we introduce our model for scheduling of parallel jobs, summarize our results and present some basic techniques. In Part II we prove the results about scheduling without dependencies, and in Part III we prove the results about scheduling with dependencies.

The result on scheduling of sequential jobs is presented in Part IV, which includes the definitions and overview of the results and previous work. This part is completely self-contained and independent of previous parts.

Part I
A model and results for scheduling parallel jobs

We first define our model and discuss the practical motivation Sections 2 and 3. Then in Sections 4 and 5 we present and discuss our results. We discuss some possible alternatives to our model and related work in Section 6. Finally, in Section 7 we introduce some basic technical tools and notation used in Parts II and III.

2 The model

2.1 Parallel machines and network topologies

For our purpose, a *parallel machine* with a specific network topology is an undirected graph where the nodes represent processors and the edges represent the communication links between the processors. A set of *job types* is a collection of subgraphs of the machine that can be requested by a job. Each job requests a particular job type, which means that it can be scheduled on any subgraph of the machine isomorphic to its job type. We assume that on any machine a job may request the whole machine (if it uses full parallelism) or a single processor (if it is an inherently sequential job). In our representation it means that the set of job types always contains at least the graph representing the whole machine and the graph with a single node.

Let us define the network topologies that we consider in this thesis. The simplest theoretical model of a parallel machine is a *PRAM*, which supports direct communication between any two processors. We represent a PRAM machine with N processors as a complete graph with N nodes, which reflects the fact that direct communication between any two processors is supported. Available job types are all complete graphs with up to N nodes. This represents the fact that a job requesting p processors can be executed on any

subset of p processors.

Machines whose underlying topology is a *d-dimensional mesh* are represented by grid graphs of dimension d. Job types are all grids with dimensions smaller than or equal to the dimensions of the machine. The *one-dimensional mesh* machine consists of N processors arranged on a line, each of them is connected to its neighbors. Any job must be executed on a connected segment of the machine. Thus the job types are all contiguous segments with up to N processors. A *two-dimensional* $n_1 \times n_2$ mesh machine is represented by a rectangular grid with width n_1 and height n_2. The set of job types is the set of all smaller rectangular grids. In contrast to PRAMs and one-dimensional meshes, in the case of two-dimensional meshes the job types are not determined by the number of processors, since we distinguish between a 10×10 rectangle and a 5×20 rectangle.

A *d-dimensional hypercube* has $N = 2^d$ processors indexed by vectors of d bits. Two processors are connected if their indices differ in exactly one bit. Available job types are all d'-dimensional subcubes for $d' \leq d$.

2.2 Parallel jobs and virtualization

A *parallel job* is characterized by the job type G and the running time t on a set of processors of that job type. We assume that all processors run at the same speed, thus if a job is executed on two isomorphic subgraphs, the running times are equal. The *work* of a job is $t|G|$, where $|G|$ is the *size* of the job, i.e., the number of processors requested by the job. A *sequential job* is a job requesting one processor.

An important fact is that any parallel job can be scheduled on fewer processors than it requests. In the extreme case we can run it on a single processor. Then all the work is done by that processor, which increases the running time to the product of the requested number of processors and the original running time of the job. In general, a job is executed on a smaller set of processors, each of them simulating several processors requested by the job, and the running time is proportionally larger. This technique is called *virtualization* and can be implemented by the operating system without any knowledge of the algorithm executed by the parallel job [HB84, Sar89, Ble90,

SOG+94]. Virtualization yields good results if the mapping of the requested processors on the smaller set preserves the network topology. This can be achieved for machines with regular topology, which includes all machines considered in this work, PRAMs, meshes and hypercubes.

A job requesting a machine G with running time t can run on a machine G' in time $\alpha(G, G')t$, where $\alpha(G, G')$ is the *simulation factor* [BCH+88, KA86]. Neither the running time nor the work can be decreased by virtualization, i.e., $\alpha(G, G') \geq \max(1, |G|/|G'|)$. If the network topology of G can be efficiently mapped on G', the work does not increase. We assume that the simulation assumptions always preserve the work. This is a reasonable assumption, since efficient mappings exist for all network topologies we consider.

Under this assumption, a job which requests p processors on a PRAM (resp. one-dimensional mesh) can be simulated efficiently on a PRAM (resp. one-dimensional mesh) of p' processors for $p' < p$ and the running time increases time by a simulation factor of p/p'. A job requesting an $a \times b$ mesh can be simulated on an $a' \times b'$ mesh for $a' \leq a$ and $b' \leq b$ and the running time increases by by a simulation factor of $ab/a'b'$. A job requesting a d-dimensional hypercube can be run on a d'-dimensional hypercube for $d' < d$ and the running time increases by $2^{d-d'}$.

2.3 Parallel job systems and schedules

A *parallel job system* (without dependencies) is a collection of parallel jobs. A *parallel job system with dependencies* is a collection of jobs with the dependencies given by a directed acyclic graph called the *dependency graph*. The nodes of the graph correspond to the jobs and edges correspond to dependencies between the jobs. A job can be scheduled only when all its predecessors in the dependency graph are finished.

A *schedule* is an assignment of a set of processors and a time interval to each job such that all requirements given by the job types, running times and dependencies are satisfied. That means that (1) the processors assigned to a job correspond to its job type or can simulate it using virtualization, (2) the length of the time interval assigned to a job is its running time multiplied by the simulation factor, if virtualization is used, and (3) if there is a dependency

between two jobs, then the time interval assigned to the first job ends before the interval assigned to the second job starts. A processor can be assigned to at most one job at any time.

Once a job is started on some processor or a set of processors, it has to run on them until completion. We do not allow a job to be preempted, i.e., to move it to a different set of processors where it would continue to run. We also do not allow a job to be stopped and restarted later on the same or different processors.

2.4 The scheduling problem and scheduling algorithms

A *scheduling problem* specifies a network topology together with a set of available job types and simulation factors, whether dependencies are allowed and whether virtualization can be used. As a network topology usually determines the job types and simulation factors in a natural way, we omit this part of the specification most of the time. An *instance* of the scheduling problem is a parallel job system (with dependencies, if they are allowed) and a machine with the given topology. The output is a schedule for the given job system on the given machine.

A *scheduling algorithm* (for a given scheduling problem) produces a schedule for any instance of the problem. A scheduling algorithm is *off-line* if it receives the complete information as its input, i.e., all jobs, their dependencies and running times. In the on-line problem, the running times are not given as a part of the input, but can only be determined by executing the jobs. For on-line algorithms, we distinguish two notions, depending on whether the jobs and their resource requirements are given in advance or only when the jobs become available. We say that an algorithm is *on-line* if the running times are only determined by scheduling the jobs and completing them, but the dependency graph and the resource requirements may be known in advance. An algorithm is *fully on-line* if it is on-line and at any given moment it only knows the resource requirements of the jobs currently available but has no information about the future jobs and their dependencies.

2.5 The performance measure

We measure the performance by the *length of a schedule*, which is the time when the last job finishes (the makespan). An *optimal* schedule for a given job system is a schedule with the minimal length. An optimal schedule can be computed off-line, with knowledge of all running times and dependencies, and with unlimited computational power.

An on-line algorithm is evaluated by the *competitive ratio*, introduced by Sleator and Tarjan in [ST85], which compares the performance achieved by the on-line algorithm to the optimal schedule. In the case of a scheduling algorithm, the competitive ratio for a given input is the ratio of the length of a schedule produced by the on-line algorithm to the length of an optimal schedule. The competitive ratio of an on-line algorithm is the maximal competitive ratio over all inputs. Equivalently, we say that a scheduling algorithm is σ-*competitive* if for every input it generates a schedule which is at most σ times longer than the optimal schedule. A randomized scheduling algorithm is σ-competitive if for every the expected length of the schedule S generated by the algorithm is at most σ times longer than the optimal schedule for each instance; the expectation is taken over the random bits of the scheduling algorithm.

3 Practical motivation

Our model is motivated by scheduling on massively parallel machines in two different scenarios.

In the first scenario, the massively parallel machine is running in a batch mode for some period of time. All jobs from different users are submitted in advance and their degree of parallelism is known, but the running times might be unknown. In practice, supercomputer centers typically run their computers in a similar scenario for at least some part of the time. Typically this is used for computationally intensive jobs with limited input and output. for which the costs that we neglect are relatively small. Thus this scenario matches our model of scheduling without dependencies very well. (We discuss

the variation of this scenario when jobs are not given in advance but arrive at some fixed times in Section 6.7.)

In practice, the scheduling problems are solved by very simple heuristics. The parallel machine is typically partitioned into a few partitions (e.g, a CM5 machine with 512 processors can be divided into 5 partitions of 256, 128, 64, 32 and 32 processors). Jobs can require only one of these few sizes, and a separate queue is maintained for every partition. If there are no jobs for one partition, its processors are left unused. Jobs requiring the whole machine are run in special batches. In addition, to aid in scheduling, users are often required to give an estimate of the running time of their job. While such simple heuristics might be sufficient for the size and load of the parallel computers today, it is clear that this is not a solution that can satisfy growing demands in the future.

In the second scenario, a parallel job system with dependencies can be a result of a decomposition of a large task into subtasks that might differ in the degree of parallelism that can be used efficiently. Such a decomposition can be provided by a programmer, or it could be obtained automatically by a sophisticated compiler. Typically in such a decomposition the amount of communication between the separate subtasks is small. This matches our model of scheduling with dependencies.

Current research and methods for design of parallel compilers are focused on partitioning the processors and assigning them to the subtasks during the compile time; this assignment is then fixed and is independent of the input data [Sar89, Sub93. SOG+94]. This approach is suitable if the dependency of the running times on the input is small, since then the necessary estimates of the running times of the subtasks can be based on the previous performance. However, for many applications the running times depend significantly on the input. To achieve a satisfactory performance for such tasks, it is necessary to address the on-line scheduling problem.

4 Our results

For most network topologies we prove tight or very close lower and upper bounds on the competitive ratio, both with and without dependencies.

The technically most difficult results are the bounds for randomized scheduling, specifically our randomized constant-competitive algorithm for meshes without dependencies, and our lower bound of $\Omega(\frac{\log N}{\log\log N})$ for randomized scheduling on one-dimensional meshes with dependencies. Of the results for deterministic scheduling, the most difficult are the lower bound of $\Omega(\sqrt{\log\log N})$ for scheduling on meshes without dependencies and the tradeoff for scheduling on PRAMs with dependencies using virtualization.

In all the results, N denotes the number of processors of the machine.

4.1 Scheduling without dependencies

For deterministic on-line scheduling without dependencies we obtain optimal or almost optimal algorithms for all basic network topologies that we study, PRAMs, hypercubes, one- and two-dimensional meshes, and higher-dimensional meshes if the dimension is constant. The bounds on the competitive ratio for deterministic scheduling are summarized by Table 1. The lower bound of $2 - \frac{1}{N}$ uses only sequential jobs and is from [SWW91]; we present it in Section 7.

Topology	Upper bound	Lower bound
two-dim. mesh	$O(\sqrt{\log\log N})$	$\Omega(\sqrt{\log\log N})$
PRAM	$2 - \frac{1}{N}$	$2 - \frac{1}{N}$
hypercube	$2 - \frac{1}{N}$	$2 - \frac{1}{N}$
one-dim. mesh	2.5	$2 - \frac{1}{N}$
d-dim. mesh	$O(d \log d \sqrt{\log\log N} + (2d \log d)^d)$	$\Omega(\sqrt{\log\log N})$

Table 1: *The bounds on the competitive ratio for deterministic on-line scheduling without dependencies.*

If randomization is allowed, we obtain an on-line algorithm for scheduling on d-dimensional meshes for which the competitive ratio depends only on d. Moreover, under some restrictions we obtain a much stronger result, an algorithm with competitive ratio that does not even depend on the dimension of the mesh. The restrictions are that the dimensions of the jobs and of the mesh are powers of two, and all the dimensions of each job are strictly smaller than the corresponding dimensions of the mesh (note that this implies that each dimension of a job is at most a half of the corresponding dimension of the mesh, since all dimensions are powers of two). See Table 2 for the summary of these results.

Topology	Restrictions	Upper bound
two-dim. mesh	none	$O(1)$
d-dim. mesh	dimensions of the mesh are powers of two; dimensions of jobs are powers of two and smaller than the dimensions of the mesh	$O(1)$
d-dim. mesh	none	$O(4^d)$

Table 2: *The bounds on the competitive ratio for randomized on-line scheduling without dependencies.*

4.2 Scheduling with dependencies

For scheduling with dependencies we study separately the cases with and without virtualization, since they are quite different.

For deterministic scheduling we also study the dependence of the competitive ratio on the size of the largest job. We assume that N is the number of processors and that no job requests more than λN processors, where $0 < \lambda \leq 1$ is a constant. For scheduling on PRAMs we obtain tight tradeoffs between λ and the optimal competitive ratio in both cases, with and without virtualization.

With virtualization, we obtain an optimal algorithm for one-dimensional meshes and efficient algorithms for hypercubes and higher-dimensional meshes. Table 3 summarizes the results on deterministic scheduling with virtualization.

Topology	Restrictions	Upper bound	Lower bound
PRAM	none ($\lambda = 1$)	$2 + \phi$	$2 + \phi$
PRAM	$0 < \lambda \leq 1$	$2 + \frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$	$2 + \frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$
one-dim. mesh	none	$O(\frac{\log N}{\log\log N})$	$\Omega(\frac{\log N}{\log\log N})$
d-dim. mesh	none	$(O(\frac{\log N}{\log\log N}))^d$	$\Omega(\frac{\log N}{\log\log N})$
hypercube	none	$O(\frac{\log N}{\log\log N})$	

Table 3: *The bounds on the competitive ratio for deterministic on-line scheduling with dependencies with virtualization;* $\phi = (\sqrt{5} - 1)/2 \approx 0.618$ *is the golden ratio.*

Without virtualization, in addition to the tradeoff for PRAMs mentioned above, we prove that no efficient scheduling is possible if the size of the jobs is not restricted. See Table 4 for the results.

Topology	Restrictions	Upper bound	Lower bound
arbitrary	none ($\lambda = 1$)	N	N
arbitrary	$0 < \lambda < 1$		$1 + \frac{1}{1-\lambda}$
PRAM	$0 < \lambda < 1$	$1 + \frac{1}{1-\lambda}$	$1 + \frac{1}{1-\lambda}$

Table 4: *The bounds on the competitive ratio for deterministic on-line scheduling with dependencies without virtualization.*

Even if randomization is allowed, we cannot use it efficiently in the presence of dependencies. We prove a lower bound showing that no randomized algorithm for scheduling on one-dimensional meshes with virtualization has a better competitive ratio than $\Theta(\frac{\log N}{\log\log N})$. Thus our deterministic algorithm

is within a constant factor of the optimal competitive ratio. We also prove that with randomization still no efficient scheduling without virtualization is possible if the size of jobs is not restricted; we get a lower bound of $N/2$ in this case. See Table 5 for these results.

Topology	Virtualization	Lower bound
one-dim. mesh	allowed	$\Omega(\frac{\log N}{\log \log N})$
arbitrary	not allowed	$N/2$

Table 5: *The bounds on the competitive ratio for randomized on-line scheduling with dependencies.*

4.3 Structure of dependencies

In addition to the bounds on the competitive ratio we prove that for on-line scheduling with dependencies the structure of the dependency graph is not so important. The following theorem which says that any on-line algorithm for scheduling job systems whose dependency graphs are trees can be converted to an algorithm for scheduling general dependency graphs with the same competitive ratio.

Theorem 4.1 *Let an on-line scheduling problem with dependencies be given (i.e., a specific architecture and simulation factors). Then the optimal competitive ratio for this problem is equal to the optimal competitive ratio for a restricted problem in which we allow only job systems whose dependency graphs are trees as inputs.*

This theorem is easy to prove for fully on-line algorithms, since then the algorithm does not know the dependency graph in advance. The same theorem holds even for general on-line algorithms, where the algorithm knows the dependency graph and all job types before the scheduling starts, but a more sophisticated argument is necessary.

4.4 Technical assumptions

All our lower bound results assume that the running time of a job may be zero. This is only a convenient technical assumption which can be removed easily. As all our proofs are constructive, we simply replace all zero times by unit times and scale all other running times to be sufficiently large. This only decreases the lower bounds by arbitrarily small additive constants.

The lower bounds for PRAMs for scheduling with dependencies give the best constant competitive ratios that can be achieved for all N if λ is fixed. There is a small additional term that goes to 0 as N grows.

4.5 History of the problem

The model of deterministic on-line scheduling of parallel jobs without dependencies was first introduced and studied in a joint paper with Anja Feldmann and Shang-Hua Teng [FST91]. That paper contains the results of Theorem 10.1 from Section 10, Sections 11.1 to 11.3 and Section 12.1 of this thesis. The journal version [FST94] of the paper [FST91] contains all the results above together with the results of Sections 8, 9 and the rest of the Section 10.

The results on randomized scheduling without dependencies from Sections 11.4, 12.2 and 12.3 has not been published before. Sections 11.1, 11.2 and 12.1 are substantially revised versions of the material from the paper [FST91].

Deterministic on-line scheduling with dependencies was introduced and studied in joint papers with Anja Feldmann, Ming-Yang Kao and Shang-Hua Teng [FKST92, FKST93]. Those papers contains the results of Sections 13, 14.1, 15 and 16.

The results on randomized scheduling without dependencies from Section 14.2 and part (iii) of Theorem 15.1 in Section 15 has not been published.

The claim in [FKST92, FKST93] that the $O(\frac{\log N}{\log \log N})$-competitive algorithm for scheduling on hypercubes is optimal is wrong; we have a matching lower bound at the present time.

5 Discussion of the results

Our results show, not surprisingly, that scheduling with dependencies is significantly harder than scheduling without dependencies, which is what we expected. What is somewhat surprising is how much harder it is in some cases; for example without dependencies we have a 2.5-competitive algorithm for one-dimensional meshes, while with dependencies no algorithm for one-dimensional meshes can achieve a better competitive ratio than $\Theta(\frac{\log N}{\log \log N})$, even if both randomization and virtualization are allowed.

It is very interesting to compare the influence of various factors on the performance of scheduling with and without dependencies, see Table 6. We examine these factors one by one in the rest of this section.

Factor	Influence without dependencies	Influence with dependencies
randomization	yes	no
network topology	small	yes
virtualization	no	yes
size of the jobs	no	yes

Table 6: *Factors influencing the performance of on-line scheduling algorithms.*

The performance of scheduling with dependencies depends significantly on virtualization, and the maximal size of a job, while for scheduling without dependencies these factors are not very important. On the other hand, randomization does not help much for scheduling with dependencies, while it significantly improves performance of scheduling without dependencies on meshes. Network topology has a big influence on the performance of scheduling with dependencies: without dependencies the changes in performance are smaller, but for more complex topologies the algorithms are significantly more complex.

5.1 Randomization

For scheduling with dependencies we can prove that the randomization does not help, and the optimal competitive ratio for the one-dimensional mesh is still $\Theta(\frac{\log N}{\log \log N})$, which is achieved by a deterministic algorithm.

On the other hand, without dependencies, we can use randomization to significantly improve the competitive ratio for scheduling on two- and higher-dimensional meshes. If the dimension of the mesh is constant, the optimal competitive ratio for deterministic algorithm is $\Theta(\sqrt{\log \log N})$, while our randomized algorithm achieves a constant competitive ratio. If the dimensions of the machine and of the jobs are powers of two, and there are no large jobs, the competitive ratio does not even depend on the dimension of the mesh. If there are no restrictions, the competitive ratio is $O(4^d)$, which is still significantly better than the deterministic algorithm, where the dependency on d is $O((2d \log d)^d)$. Note that in practice d is very small, typically a constant, as arbitrarily large meshes can be built without changing d. Hence the competitive ratio is not that large even if the dependency on d is exponential.

To achieve such a strong result, we estimate for each job size the total work of all jobs of that size based on a small random sample. However, it is not clear how this can lead to a constant competitive ratio, since the number of different job sizes depends on the number of processors, and in particular it is exponential in d. We use Chernoff-Hoeffding bounds in a powerful way to prove that with some constant probability all of the many different instances of sampling give a good approximation of the work.

These results are particularly interesting in the view of the fact that the lower bound for deterministic algorithms for scheduling on one-dimensional meshes with dependencies in [FKST93] and the lower bound for deterministic algorithms for scheduling on two-dimensional meshes without dependencies in Section 11.3 both use a very similar technique. Yet the first lower bound generalizes to randomized algorithms and the other one does not.

It is also interesting that randomization is used in our algorithm only to randomly permute the jobs at the beginning for the purpose of sampling. If we assume that the usage pattern of a parallel machine does not change very fast, we could estimate the work of jobs of different sizes based on the

previous usage of the machine, and then schedule them very efficiently even without randomization. This is actually used in practice, since scheduling is sometimes done manually based on the previous experience and data. Our result explains to some extent why it might be very useful to consider the estimates based on previous experience. If these estimates are good, it saves us the sampling which is a relatively expensive part of our algorithm.

5.2 Network topology and greedy algorithms

Our results show that the complexity of the network topology has a big influence on on-line scheduling.

On the more complex machines, not only is the optimal performance lower, but also the algorithms are more involved. The simplest algorithms use the greedy method, which means that they schedule any available job as soon as its resource requirements can be satisfied. This is optimal for PRAMs, both with and without dependencies, but as the network topology gets more complex, the performance of greedy methods decreases because they tend to scatter the available processors and hence make them unusable for larger jobs.

This is particularly clear without dependencies. Already for hypercubes and one-dimensional meshes we need to modify the greedy approach. In our algorithms we schedule the largest job whenever processors are available, as in a greedy algorithm, but if it is impossible to schedule the largest job, all smaller jobs are postponed as well, even if they could be scheduled immediately. For the two-dimensional mesh we have to abandon the greedy approach completely. The optimal algorithm begins by using only a small fraction of the mesh without even attempting to use the whole mesh, and continues with a larger fraction only when the available processors in the current fraction become unusable. The proof of the lower bound actually shows that this is not an arbitrary choice—no greedy-like algorithm can achieve a substantially better competitive ratio than $\Omega(\log \log N)$. This shows that the general heuristic of using greedy algorithms for scheduling can be wrong despite the fact that it works well in many previous scheduling algorithms, see for example [Gra66, LST90, SWW91].

In the case without dependencies we can still keep the competitive ratio constant by using more complex algorithms and randomization for all architectures including the higher-dimensional meshes. However, there is still some difference in the performance since the constants are higher for more complex topologies.

On the other hand, with dependencies the performance decreases sharply with the increasing complexity of the machine. Intuitively, the reason is that algorithms for scheduling with dependencies have to be able to deal with the jobs that become available later during the schedule, and hence we do not have the option of abandoning the greedy approach completely.

5.3 Virtualization

Without dependencies, virtualization does not help us to improve the performance of our algorithms. In fact, all our algorithms for scheduling without dependencies do not use virtualization but are competitive even against schedules that use virtualization. All our lower bounds for scheduling without dependencies are valid even for algorithms that use virtualization.

This is no longer true in the presence of dependencies. The main reason for this distinction is that without dependencies we can process all large jobs first. But with dependencies, jobs requiring the whole machine can be dependent on other jobs. When they become available, other jobs may be running and we have to wait until all or most of them finish to be able to satisfy the requirements of the large jobs. This causes an inefficiency which can be avoided only by the use of virtualization.

This is clearly demonstrated by the results on scheduling with dependencies when virtualization is prohibited. If we allow jobs requiring the whole machine, no efficient scheduling is possible on any machine, even using randomization. If we restrict the number of processors that a job can request to some constant fraction of the machine, the situation improves somewhat, but the competitive ratio is still significantly larger than with analogous restriction on the size of jobs and virtualization allowed.

All our algorithms use virtualization only for large jobs, or can be modified to do that. This support the intuition that the large jobs are the main

problem, which makes scheduling with dependencies impossible without virtualization.

5.4 The size of the jobs

With virtualization, efficient scheduling is possible even with no restriction on the size of jobs. However, even then the competitive ratio depends on the maximal allowed size of jobs.

Our tight tradeoffs between the maximal size of a job and the optimal competitive ratio for deterministic scheduling on PRAMs with dependencies both with and without virtualization are illustrated on Figures 1 and 2.

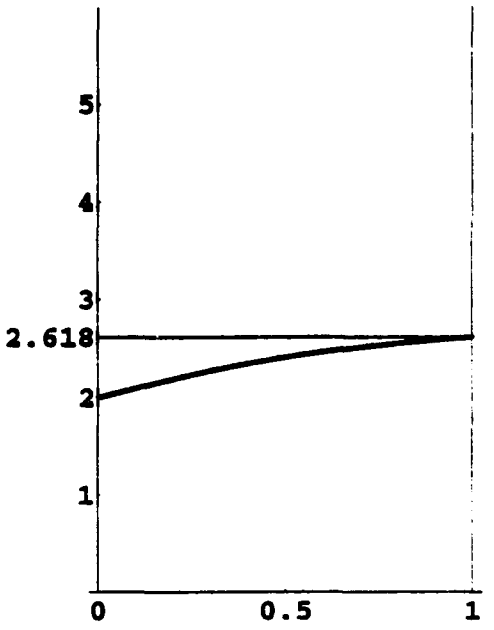


Figure 1: *The relation between λ and the competitive ratio for PRAMs, using virtualization*

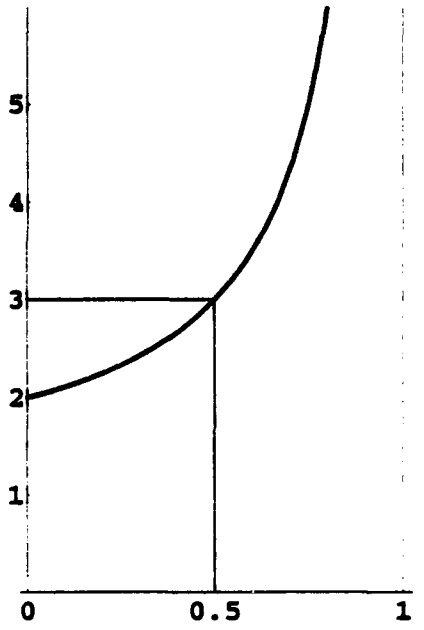


Figure 2: *The relation between λ and the competitive ratio for PRAMs, without using virtualization.*

If the size of all jobs is a small fraction of the total number of processors, the competitive ratio is in both cases close to 2, which is the optimal

competitive ratio even if we allow only sequential jobs and no dependencies. If the maximal size of jobs increases, the competitive ratio increases to $2 + \phi \approx 2.618$ if virtualization is allowed, but with no virtualization it is unbounded.

6 Previous models and results

6.1 Off-line scheduling and approximation algorithms

First, let us point out that the exact solution of off-line scheduling problems is NP-hard even for very simple special cases involving only sequential jobs, see [GJ79]. For parallel jobs, Blażewicz, Drabowski and Węglarz [BDW86] proved that if the number of processors is a part of the input, optimal scheduling is NP-complete. Later Du and Leung [DL81] improved this result and showed that optimal scheduling is NP-hard for scheduling on two processors with dependencies and five processors without dependencies.

Since optimal scheduling is hard, there have been a significant interest in approximation algorithms. However, it is NP-complete even to decide if a set of sequential jobs with unit running times and with dependencies has a schedule of length three on a given number of processors [LRK78]. This result implies that it is NP-hard to approximate the length of an optimal schedule for sequential jobs within a factor better than 4/3. For parallel jobs, it is easy to see that even deciding if a set of jobs with unit running times without dependencies has a schedule of length two is as hard as NP-complete problem PARTITION, and hence it is NP-hard to approximate the length of an optimal schedule within a factor better than 3/2

Approximation algorithms are related to this thesis, since every on-line algorithm is also an approximation algorithm, but the converse is not necessarily true. Our on-line algorithm have similar or even better performance than the previously known approximation algorithms. Most previous results on scheduling parallel jobs do not consider the network topology and thus in our model they are valid only for scheduling on PRAMs.

The results of Graham [Gra66] on list scheduling give an approximation

algorithm with an approximation ratio of $(2 - \frac{1}{N})$ for scheduling of sequential jobs on N processors without dependencies. This algorithm is in fact on-line and can be easily generalized for parallel jobs, which gives our result in Section 8. A similar result appears also in [TWY92].

Wang and Cheng [WC92] give an approximation algorithm for scheduling on PRAMs with dependencies which achieves approximation ratio of 3. Their algorithm is not on-line. We improve this result in Section 13 by presenting an on-line algorithm which achieves a competitive ratio of approximately 2.618, which is optimal for on-line algorithms.

There have been some results on approximation algorithms for scheduling of independent jobs on hypercubes and one-dimensional meshes. Chen and Lai [CL88] give an algorithm for hypercubes that achieves the approximation ratio of $2 - \frac{2}{N}$. Their algorithm is not on-line, however, it is very similar to our on-line algorithm from Section 9 which achieves the optimal competitive ratio of $2 - \frac{1}{N}$.

Scheduling on one-dimensional meshes without dependencies is equivalent to packing two-dimensional rectangles into a two-dimensional bin so that the total height is as small as possible. For a long time the best known algorithm was by Sleator [Sle80] which achieves an approximation ratio of 2.5. This algorithm is not on-line, and our 2.5-competitive algorithm in Section 10 is very different. Recently Steinberg [Ste93] obtained an algorithm with an approximation ratio of 2. This algorithm uses the information about running times in a crucial way and hence it is not on-line.

6.2 Computational complexity of on-line algorithms

The competitive ratio does not measure the amount of computation done by the on-line algorithms in any explicit way. However, typically the algorithms that are good from the viewpoint of competitive analysis are also simple and do not require a large amount of computation.

This is true in our case. The most complex operation done by our scheduling algorithms is to sort the jobs according to their size. This can be done very fast in parallel, since the number of different sizes is at most proportional to the number of processors. Once the jobs are sorted, they are either

processed in a predetermined order, or the different sizes are treated independently, and no significant amount of computation is needed.

The algorithms can also be executed in a distributed environment, since it is possible to implement them with a very limited amount of communication.

6.3 Emphasis on network topology

The new feature of our model is that we study on-line scheduling on concrete network topologies. Only a small fraction of previous work on scheduling of parallel jobs takes into account the network topology of the machine [Sle80, CL88, TWY92]; in all cases the focus is on approximation algorithms of independent jobs. The rest of previous literature on scheduling is concerned only with the number of processors required by a job, not with the constraints of the concrete network topology; thus from our point of view, these results apply only to the simplest network topology, PRAMs.

Various network topologies have been studied extensively, but with a different emphasis. A typical questions in this area is whether it is possible to simulate one network by another network (either with a different topology, or with a different size), and how efficient the simulation can be [BCH+88, KA86, Ran87]. Such results are closely related to the technique of virtualization. Virtualization is possible only if the simulation of the given network by a smaller one can be efficient. Therefore the most important aspect of these results for us is that they justify the technique of virtualization for simple cases and study its limits for more complex network topologies.

6.4 Virtualization

Virtualization is possible on practical systems with a regular topology, possibly with some additional costs that we neglected. These costs are small if the parallel job uses parallelism to reduce its running time, and other resources are not critical. In some cases a job might need some amount of parallelism to satisfy other resource requirements. For example, it might be memory-intensive job, and if there is a limited amount of memory available

to each processor (which is usually the case), we have to schedule it on a sufficient number of processors to satisfy its memory requirements. In such a case virtualization is not available at all or only to a limited degree. Since our algorithms use virtualization only for large jobs, our results may still apply or be modified for the particular application.

We assume that virtualization preserves the work. In other words, this means that up to a certain number of processors a parallel job scales perfectly and the running time decreases proportionally, and after that increasing the number of processors does not improve the running time. Examples given in [SOG+94] show that this approximation closely matches the behavior of many tasks encountered in practice.

For approximation algorithms some research has been done under the assumption that virtualization does not preserve the work [TWY92]. In the off-line setting, an approximation algorithm can use the full information about the running time of a job on any number of processors. In particular, it can find the number of processors such that the work of the job is minimal, which is essential for efficient scheduling. In the on-line case, we assume no knowledge of the running times. If we do not even know how the work of the job depends on the number of processors, then no efficient scheduling is possible.

6.5 Speed of processors

In our model we require that all processors run with the same speed. Our motivation of massively parallel computation assumes one parallel machine for processing parallel jobs with possibly high amount of communication between the processors and such parallel machines are designed with processors of the same speed. It is difficult to imagine a parallel job that can run on processors of possibly different speeds, since in such cases it is usually possible and more efficient to break up the job into sequential jobs with dependencies.

In the case of sequential jobs, processors of different speeds are motivated by the environment in which many sequential machines (possibly different) are connected by a communication network. This model is outside the scope of this work. It leads to interesting problems studied for example in [LST90,

SWW91, ST93].

6.6 Preemption

In our model, it often happens that many processors are available but not in the requested configuration. If we are allowed to preempt a job, i.e., to move a running job to different processors, or at least to restart a job on different processors, we can reorganize the running jobs periodically so that the unused processors can be used more efficiently.

But on massively parallel machines, the cost of context switching, which has to be used for preemptions and restarts is very expensive. The main reason is that moving the data and the current state of the memory requires high amount of communication, which can interfere with execution of other parallel jobs. If we allow preemptions, these costs can be incurred repeatedly, and thus cannot be included in the running time of the job. If the preemptions occur often, the additional costs are very high. The additional costs of restarting a job are similar, and in addition, some jobs might not allow restarting at all. For these reasons we consider our choice of the model without preemption and restarts more reasonable.

For scheduling sequential jobs, preemptive on-line scheduling is studied for example in [MPT93, SWW91].

6.7 Fixed release times

We assume that a job can be executed immediately unless it is dependent on other jobs. However in practice a job may become available at a fixed time, independent of other jobs. This is a motivation for a model with release times, considered for example in [GJ79, SWW91]. In this model each job has a fixed release time, and it cannot be executed earlier. This might appear to be a natural and powerful extension of our model of scheduling without dependencies.

We have two reasons showing that we do not need to incorporate the release times explicitly. First, the effect of the fixed release times can be easily

simulated in the model with dependencies. Second, a very general observation of [SWW91] shows that even in the model without dependencies, an on-line problem with release times is not much more difficult than the corresponding problem without release times. The intuition is that before the last job is released, nothing important can happen, because the optimal schedule has to contain that time interval as well. Therefore if we start scheduling only after the last job has arrived, we lose only a small constant factor. Of course, we do not know which job is the last one, but this is a minor technical problem which can be solved by scheduling the jobs in several batches as follows. We first schedule all jobs available at the beginning, disregarding the newly arrived jobs. After all of them are finished, we schedule all jobs available at that point, and so on. Using this approach, the competitive ratio is only twice as much as the competitive ratio for the algorithm used to schedule the individual batches.

6.8 Performance measures

Intuitively, the competitive ratio can be interpreted as a value that measures the cost of the information about the future, in our case the information about the running times. We pay for not having the information about the running times of jobs in advance by decreasing our efficiency by a factor which is at most the competitive ratio. Thus, given an algorithm with a small competitive ratio, we know that any schedule which it produces is close to the optimal one.

In both our models, with and without dependencies, the length of the schedule and the competitive ratio seem to be good and realistic performance measures.

In the model with fixed release times mentioned above, using the total length of the schedule as a performance measure is somewhat questionable. The competitive ratio (with respect to the length of the schedule) is still relevant to get a rough picture. As we will see later, to achieve a competitive ratio of σ, the algorithm typically has to maintain at least $1/\sigma$ of the processors busy most of the time. Hence if σ is the optimal competitive ratio, and the total work of the jobs is $1/\sigma$ fraction of the work that can be done

by the machine, it is possible to schedule all jobs; otherwise the backlog of unscheduled jobs and the response time necessarily increase with time.

However, since the model of fixed release times is motivated mostly by an interactive environment in which users can submit their jobs at any time, to get more accurate and insightful results, it would be essential to consider different measures of performance, such as the response time. Average response time is considered for several variants of on-line scheduling of sequential jobs in [MPT93]. The work of [TSWY94] considers approximate off-line scheduling of parallel jobs with respect to average response time. On-line scheduling of parallel jobs with average response time as the performance measure is an interesting area for further research outside the scope of this work.

7 Notation and basic techniques

For a given job system $\mathcal{J}$ with or without dependencies, we denote the length of an optimal schedule by $T_{\text{opt}}(\mathcal{J})$. By $T(S)$ we denote the length of the schedule S, i.e., the time when the last job finishes (its makespan). A deterministic scheduling algorithm is σ-competitive if for every job system $\mathcal{J}$ the schedule S generated by the algorithm satisfies $T(S) \leq \sigma T_{\text{opt}}(\mathcal{J})$. A randomized scheduling algorithm is σ-competitive if for every job system $\mathcal{J}$ the expected length of the schedule S generated by the algorithm satisfies $\mathrm{E}[T(S)] \leq \sigma T_{\text{opt}}(\mathcal{J})$, where the expectation is taken over the random bits of the scheduling algorithm.

For proving lower bounds on the competitive ratio it is useful to interpret the problem as a game between the scheduling algorithm and the *adversary*. In the deterministic case the adversary chooses the number of jobs, their job types and dependencies in advance. Then the on-line scheduling algorithm starts scheduling them, while the adversary has complete control over the running times of the jobs; he can stop any job at any time. To prove the lower bound, we simulate this game, and then use the job system with running times as specified by the adversary during the game as an input for the scheduling algorithm. The actions of the scheduling algorithm are the same, as it is deterministic and on-line.

For randomized algorithms we can define different models depending on the strength of the adversary, see [BDBK+90]. Our model corresponds to an *oblivious adversary*, which means that the adversary has no access to the random bits of the scheduling algorithm. This makes the adversary significantly less powerful than in the deterministic case.

To analyze the competitive ratio of our algorithms, we need to bound the length of the optimal schedule. We use the following two lower bounds for the optimal schedule.

First, the optimal schedule has to schedule the jobs that are dependent on each other sequentially. For a given job system $\mathcal{J}$, $T_{\max}(\mathcal{J})$ denotes the maximal sum of running times of jobs along any path in the dependency graph; for a job system without dependences this is just the maximal running time of a job. Clearly $T_{\max}(\mathcal{J}) \leq T_{\text{opt}}(\mathcal{J})$ because an optimal algorithm has to schedule these jobs sequentially; even if we use virtualization, their running time cannot get shorter.

Second, the optimal schedule has to perform all the work of all jobs. For a given job $J \in \mathcal{J}$, the *work* of J, denoted work(J), is the product of the number of processors it requests and its running time. We define $T_{\text{eff}}(\mathcal{J})$ to be the time required to perform the work of all jobs on N processors, where N is the number of processors of the given machine. $T_{\text{eff}}(\mathcal{J}) = \sum_{J \in \mathcal{J}} \text{work}(J)/N$. Clearly $T_{\text{eff}}(\mathcal{J}) \leq T_{\text{opt}}(\mathcal{J})$. Using virtualization does not influence this fact, because we assume that virtualization preserves the work, i.e., if a job is scheduled on a smaller number of processors (with a given network topology, see Section 2.2), its running time is proportionally larger.

We omit the argument $\mathcal{J}$ of $T_{\text{opt}}(\mathcal{J})$, $T_{\max}(\mathcal{J})$ or $T_{\text{eff}}(\mathcal{J})$ if it is clear from the context which job system we are referring to.

To analyze scheduling algorithms, the concept of efficiency is very important. Lemma 7.1 shows that if a schedule has high efficiency except for a short period of time, then the schedule cannot be much longer than the optimal one.

The *efficiency* of a set $\mathcal{C}$ of currently running jobs, denoted by eff($\mathcal{C}$), is defined as the total number of processors assigned to a job in $\mathcal{C}$ divided by N. The *efficiency of a schedule* at time t is the efficiency of the set of all jobs

running at time t, or equivalently the number of processors busy at time t divided by N. The efficiency with respect to any subgraph of the machine is defined similarly. Efficiency of an algorithm refers to the efficiency of the schedule generated by that algorithm (for a particular job system); if this algorithm is used only to schedule jobs on a part of the machine, it refers to the efficiency with respect to that part of the machine.

For a schedule S and an $\alpha \leq 1$, $T_{<\alpha}(S)$ denotes the total time during which the efficiency of S is less than α. Our symbols are summarized in Table 7.

Symbol	Explanation
$T_{\mathbf{opt}}(\mathcal{J})$	The length of an optimal schedule for a job system $\mathcal{J}$
$T_{\mathbf{max}}(\mathcal{J})$	The maximal sum of running times along any path in the dependency graph of a job system $\mathcal{J}$
$T_{\mathbf{eff}}(\mathcal{J})$	Time required to perform the total work of all jobs in a job system $\mathcal{J}$
$T(S)$	The length of a schedule S
$T_{<\alpha}(S)$	The total time during a schedule S when the efficiency is less than α

Table 7: *Table of symbols.*

Lemma 7.1 *Let S be a schedule for a job system $\mathcal{J}$ (with or without dependencies) such that the work of each job is preserved.*

(i) Let $\alpha \leq 1$, $\beta \geq 0$. Suppose that $T_{<\alpha}(S) \leq \beta T_{\mathbf{opt}}(\mathcal{J})$. Then $T(S) \leq (\beta + \frac{1}{\alpha})T_{\mathbf{opt}}(\mathcal{J})$.

(ii) Let $0 \leq \alpha_1 \leq \alpha_2 \leq 1$, $\beta \geq 0$. Suppose that the efficiency of the schedule S is at least α_1 at all times and $T_{<\alpha_2}(S) \leq \beta T_{\mathbf{opt}}(\mathcal{J})$. Then $T(S) \leq (\beta + \frac{1-\alpha_1\beta}{\alpha_2})T_{\mathbf{opt}}(\mathcal{J})$.

Proof. **(i)** is a trivial consequence of (ii) for $\alpha_1 = 0$. **(ii)** The optimal algorithm has to do at least the same amount of work as S, because in S the

work of each job is preserved. Hence

$$T_{\text{eff}} \geq \alpha_2 T(S) - (\alpha_2 - \alpha_1) T_{<\alpha_2}(S).$$

Therefore

$$\begin{aligned} T(S) &\leq \frac{1}{\alpha_2}(T_{\text{eff}} + (\alpha_2 - \alpha_1) T_{<\alpha_2}(S)) \\ &\leq \frac{1}{\alpha_2}(1 + (\alpha_2 - \alpha_1)\beta) T_{\text{opt}} = \left(\beta + \frac{1 - \alpha_1 \beta}{\alpha_2}\right) T_{\text{opt}}. \end{aligned}$$

□

Another lemma, which is useful for analysis of scheduling of job systems with dependencies, is from [Gra66]. We use it to bound the time when the efficiency is low. Such a bound makes it possible to apply Lemma 7.1.

Lemma 7.2 (Graham, 1966) *Let S be a schedule for a job system $\mathcal{J}$ with dependencies. Then there exists a path of jobs in the dependency graph such that whenever there is no job available to be scheduled, some job on that path is running.*

Proof. Let J_0 be the job that finishes last. Let t_1 be the last time before J_0 is started at which no job is available. Then there is a job J_1 running at the time t_1 that is an ancestor of J_0 in the dependency graph, as otherwise J_0 would already be available at t_0. By the same method construct t_2, J_2, t_3, J_3, ..., t_k, J_k, until there is no time with no job available before J_k is started. Because of the way we selected the jobs, $J_k, J_{k-1}, \ldots, J_0$ is a path in the dependency graph and one of the jobs $J_k, J_{k-1}, \ldots, J_0$ is running at any time when no job is available. □

Now we present the example of Shmoys, Wein and Williamson [SWW91] which proves a lower bound of $2 - \frac{1}{N}$ on a competitive ratio for scheduling on any machine of N processors. This lower bound uses only sequential jobs, and hence it is valid for any network topology. Take a job system of $N(N-1)+1$ sequential jobs. The adversary assigns running time 1 to all jobs except to the job that is started last by the on-line algorithm; to the last

job he assigns time N. The length of the schedule generated by the on-line algorithm is at least $2N-1$. The optimal schedule takes time N, and hence the competitive ratio is at least $2-\frac{1}{N}$.

For a particular architecture it is usually possible to specify the graph requested by a job by a few parameters. We represent jobs on PRAMs as (p,t), where p is the requested number of processors and t is the running time on p processors. Jobs on hypercubes are represented as (d,t), where d is the dimension of the requested hypercube and t is the running time. Jobs on d-dimensional meshes are represented as $(a_1,\ldots,a_d,t)$, meaning that the requested graph is a mesh of size $a_1\times\cdots\times a_d$ and t is the running time; we always assume without loss of generality that $a_1\geq\ldots\geq a_d$. Of course, the on-line algorithms do not know the running times.

We write our algorithms in an easy to understand pseudocode. The instruction "**wait**" means that the algorithm waits until all currently running jobs are scheduled. We say that a *processor is available*, if it is currently not assigned to any job. A *job is available*, if all its predecessors in the dependency graph are finished, and it was not scheduled yet. Note that without dependencies any unscheduled job is available. On the other hand, with dependencies it might happen that no jobs are available, but at some later time there will be available jobs, namely those that are dependent on the currently running jobs. Thus we can be sure that all jobs have been scheduled only when no jobs are available and no jobs are running.

In some algorithms we do not specify exactly which job should be scheduled next. In that case any available job (satisfying given constraints, if there are any) can be scheduled, and our bounds are true for any such implementation of the algorithm. Sometimes we require the machine to be partitioned into several subgraphs. It is understood that these subgraphs have to be disjoint; possibly they do not cover the whole machine (usually due to rounding).

By "size" and "large" we refer to the number of processors requested by a job, while "length" and "long" refers to its time. This might be especially confusing for one-dimensional mesh machines; we use "segment" for a connected part of the machine or of the real line, while "interval" is reserved for

time intervals.

In reference to processor requirements of jobs, we use "require" if virtualization is not allowed, otherwise we use "request" to indicate that a job can be scheduled on a smaller number of processors.

The formula a/bc always means the same as $a/(bc)$. All logarithms are in base 2, unless specified otherwise.

Part II
Scheduling parallel jobs with no dependencies

8 PRAMs

In this section we present an optimal $(2 - \frac{1}{N})$-competitive algorithm for on-line scheduling on PRAMs without dependencies. This is tight due to the general lower bound presented in Section 7.

This result is essentially a generalization of Graham's results on list scheduling of sequential jobs [Gra66] for parallel jobs.

This algorithm uses the natural greedy approach. It schedules an arbitrary job if sufficiently many processors are available. It does not matter which job we choose, as long as we always schedule some job as soon as possible. As we will see later, this is not true for more complex architectures.

Algorithm GREEDY

while there is an unscheduled job J **do**

  if some job J requires p processors and p processors are available,

    then schedule J on the p processors;

wait.

Theorem 8.1 *The algorithm* GREEDY *is* $(2 - \frac{1}{N})$*-competitive for a PRAM with* N *processors.*

Proof. Suppose that the algorithm generates a schedule of length T for a job system $\mathcal{J}$. Let p be the minimal number of busy processors during the entire schedule. Consider the last time τ when only p processors were busy. Let J be some job running at that time. Before J is scheduled, there could not have been p processors available, since at that point our algorithm would schedule some job. After J is finished, there also cannot be p processors

available: at any time after τ there has to be some job J' running that was scheduled after τ, as the efficiency is no longer minimal; and if there were p processors available, J' would only require $N - p$ processors and it would already have been scheduled before τ, a contradiction.

The efficiency is at least $\alpha_1 = \frac{p}{N}$ during the entire schedule, and it less than $\alpha_2 = \frac{N-p+1}{N}$ only when J is running. The time when J is running is bounded by $T_{\max} \leq T_{\text{opt}}$, hence by Lemma 7.1 we get $T \leq (1 + \frac{N-p}{N-p+1})T_{\text{opt}} = (2 - \frac{1}{N-p+1})T_{\text{opt}} \leq (2 - \frac{1}{N})T_{\text{opt}}$. □

9 Hypercubes

In this section we present an optimal $(2-\frac{1}{N})$-competitive algorithm HYPERCUBE for on-line scheduling on hypercubes. The algorithm is still greedy in the sense that if there are sufficiently many processors available to schedule a job, some job is always scheduled. However, in contrast to the algorithm for PRAM, we now require that the largest job is scheduled; this is always possible due to the nice structure of the hypercube.

A similar algorithm appears in [CL88]. Their variation of the algorithm is not on-line; by using the information about running times for scheduling they achieve a slightly better approximation ratio of $2 - \frac{2}{N}$. However, for on-line algorithms it follows from the general lower bound presented in Section 7 that it is impossible to achieve a better competitive ratio than $2 - \frac{1}{N}$; hence our algorithm is optimal.

We suppose that the jobs $J_i = (d_i, t_i)$ are sorted by size, $d_1 \geq d_2 \geq \ldots \geq d_m$, where m is the number of jobs and d_i the dimension of a hypercube required by the job J_i. We say that a d-dimensional subcube is *normal* if the coordinates of all its processors are identical except possibly the last d coordinates. This implies that if two normal subcubes of any dimension intersect then one of them is a subcube of the other one. To ensure that the space is used efficiently, the jobs are only scheduled on normal subcubes.

Algorithm HYPERCUBE

for $i := 1$ to m **do**

if there is a normal d_i subcube available,

then schedule the job J_i on it;

wait.

Theorem 9.1 *The algorithm* HYPERCUBE *is* $(2 - \frac{1}{N})$*-competitive for a hypercube of* N *processors.*

Proof. From the properties of normal subcubes and the scheduling algorithm it follows that whenever there is any processor available, then there is a whole normal d-dimensional subcube available, where d is the dimension of the job scheduled last. Since the jobs are sorted in a decreasing order according to their dimensions, it follows that any available job can be scheduled, in particular the largest one. This proves that the efficiency is 1 as long as there is some unscheduled job left. The remaining time is bounded by $T_{\max}$ and the efficiency is at least $1/N$. The theorem follows by Lemma 7.1. □

10 One-dimensional meshes

In this section we present two algorithms for scheduling on one-dimensional meshes of N processors.

Similarly to the algorithm HYPERCUBE, we schedule the large jobs first. However, we can assure that the efficiency is 1 only if the sizes of the jobs and of the machine are powers of 2. If this is not the case, the first algorithm, ORDERED, only achieves efficiency 1/2, and hence it is 3-competitive. The second algorithm, CLUSTERS, is more complex and is 2.5-competitive. The best lower bound we know is the general $2 - \frac{1}{N}$ one which still leaves a gap between the bounds.

A different algorithm which is not on-line and achieves an approximation ratio of 2.5 was obtained by Sleator [Sle80]. Recently this result was improved by Steinberg [Ste93], who obtained an algorithm which is not on-line but achieves an approximation ratio of 2.

For both algorithms we suppose that the jobs $J_i = (a_i, t_i)$ are sorted by their size, $a_1 \geq a_2 \geq \ldots \geq a_m$, where m is the number of jobs and a_i is the number of processors required by the job J_i.

Algorithm ORDERED schedules the jobs in the order of their size, exactly as algorithm HYPERCUBE does in the case of hypercubes. Note however that if the sizes of jobs are not powers of two, it can happen that a job is not scheduled even if there is a sufficiently long segment of available processors. For example, if we have three jobs of sizes $2N/3$, $N/2$ and $N/3$, the algorithm schedules the first job of size $2N/3$, and then waits until it finishes to schedule the job of size $N/2$. The job of size $N/3$ is not scheduled even though there are sufficiently many processors available, because we require the larger job of size $1/2$ to be scheduled first. This means that the algorithm is less greedy than the algorithms for PRAM and hypercubes.

If the machine consists of several disconnected segments and any job fits into any of these segments, the first algorithm can still be applied and the same bounds hold. This will be useful in Section 11.1, when we use it as a subprogram in the algorithms for two-dimensional meshes.

Algorithm ORDERED

for $i := 1$ to m **do**

if there is a segment of a_i processors available,

then schedule the job J_i on the leftmost such segment;

wait.

Theorem 10.1 *(i) If the sizes of all jobs and of the machine (or each segment, if the machine consists of more segments) are powers of two, the algorithm ORDERED is* $(2 - \frac{1}{N})$*-competitive and its efficiency is 1 as long as there are unscheduled jobs.*

(ii) For general jobs, the algorithm ORDERED is 3-competitive and its efficiency is larger than $1/2$ *as long as there are unscheduled jobs.*

Proof. If the sizes are powers of two, the proof is identical with the proof for hypercubes. For general jobs, as long as there is a job available, the size of any occupied segment is larger than the size of the largest waiting job which is in turn larger than the size of any available segment. Therefore the efficiency is larger than $1/2$ as long as some job is available, and the

the remaining time of the schedule is bounded by $T_{\max}$. Consequently by Lemma 7.1 the algorithm is 3-competitive. □

The second algorithm is basically a refinement of the previous one. If we could achieve that there are always two adjacent jobs between any two unused intervals, the efficiency would be 2/3 instead of previous 1/2. It is impossible to maintain this arrangement all the time, but by placing the jobs carefully we can still achieve an efficiency of at least 2/3.

This algorithm is less greedy than the previous one, since it can happen that the largest job is not scheduled even if there are sufficiently many processors available.

The algorithm divides the mesh into a number of segments, starting from one segment and dividing it into more as the jobs get smaller. We call those segments clusters. Each cluster contains up to 3 running jobs: a *left job* aligned with the left end of the segment, a *right job* aligned with the right end of the segment and a *middle job* somewhere between the left and right ones. (This is slightly different in Phase 2 of the algorithm.)

The jobs are divided into the set $\mathcal{J}'$ of all jobs requiring at most 1/3 of the processors and the set $\mathcal{J}''$ of all jobs requiring more than 1/3 of the processors, $\mathcal{J}'' = \mathcal{J} - \mathcal{J}'$. When we refer to the largest job in $\mathcal{J}'$ or $\mathcal{J}''$, we always mean the largest unscheduled job.

In some steps of the algorithm it is not evident that a job can be scheduled as required; we prove the correctness of the algorithm in Theorem 10.2.

A substantial part of the following algorithm and the proof is concerned with large jobs, namely the step (1)(a) and the entire Phase 2 of the algorithm. We feel that this is not the main issue, and recommend the reader to focus on the other parts.

Algorithm CLUSTERS

Phase 1:

while there is an unscheduled job in J' **do**

if there is a cluster I with efficiency less then 2/3, **then**

(1) **if** there is no left job in I **then**

(a) **if** I is the leftmost cluster and there is some job available in $\mathcal{J}''$, **then** schedule the largest job in $\mathcal{J}''$ as a left job in I,

(b) **else** schedule the largest job in $\mathcal{J}'$ as a left job in I;

(2) **else if** there is no right job in I, **then** schedule the largest job in $\mathcal{J}'$ as a right job in I;

(3) **else if** there is no middle job in I, **then** schedule the largest job in $\mathcal{J}'$ as a middle job in I, positioned so that either its left end is in the left third of the cluster or its right end is in the right third of the cluster;

(4) **else** schedule the largest job in $\mathcal{J}'$ adjacent to the middle job and divide I into two clusters with two jobs each.

Phase 2: Considering the whole mesh as a single cluster,

while there is an unscheduled job in J'' **do**

(5) **if** there is no left job, **then** schedule the largest job in $\mathcal{J}''$ as a left job,

(6) **else if** there is no right job, all jobs from J' are finished and the efficiency is at most 1/2, **then** schedule the largest job in $\mathcal{J}''$ as a right job;

wait.

Theorem 10.2 *The algorithm* CLUSTERS *is correct and* 2.5*-competitive.*

Proof. First let us show that all steps of the algorithm are correct, namely that it is always possible to schedule the jobs as required by the algorithm. Steps (1)(a) and (5) ensure that there is at least one job from $\mathcal{J}''$ running as long as $\mathcal{J}''$ is nonempty (notice that if a job from $\mathcal{J}''$ is finished, the efficiency drops under 2/3 and the condition in the step (1)(a) or (5) is satisfied). The jobs from $\mathcal{J}''$ are processed in decreasing order, starting with the largest job, so that the next job will always fit.

Steps (1)(b), (2), (5) and (6) are correct because of similar reasoning, since the jobs from both $\mathcal{J}'$ and $\mathcal{J}''$ are processed in decreasing order.

If there are both left and right jobs running in I but no middle job, and the efficiency is less then 2/3, then either the left or the right job has to be

smaller than 1/3 of the length of I, and also the largest unscheduled job in J' is smaller. This justifies step (3).

If there are 3 jobs running in I, the middle one had to be scheduled in step (3). Therefore we can assume that the middle job has its left end in the left third of I (the other case is symmetric). If the efficiency is less then 2/3, the left job must be smaller than the space between the middle and right jobs (otherwise the space occupied by the three jobs would be at least as large as the interval from the left end of the middle job to the right end of I, which is at least 2/3 of I by the previous assumption). The largest unscheduled job is not larger than the left job, therefore it can be scheduled between the middle and right jobs and step (4) is justified.

Now we prove that the competitive ratio is at most 2.5.

The algorithm ensures that the efficiency is at least 2/3 as long as there is some job available in $\mathcal{J}'$. So if the job which finishes last is from $\mathcal{J}'$, we get by Lemma 7.1 that the competitive ratio is at most $3/2 + 1 = 2.5$.

If the last job is from $\mathcal{J}''$, it means that during the entire schedule at least one job from $\mathcal{J}''$ is running and hence the efficiency is more than 1/3.

Before we proceed with the proof, note that if the off-line algorithm is not allowed virtualization, it can run at most two jobs from $\mathcal{J}''$ at once. Hence if the job that finishes last is from J'', the length of the on-line schedule is within a factor of 2 any schedule that does not use virtualization. The rest of the proof is needed only because we allow virtualization for the off-line algorithm.

Let T be the length of the entire schedule. We distinguish two cases depending on the efficiency at the time when the last job from $\mathcal{J}'$ finishes. If it is at most 1/2, then the only time when the efficiency can be less than 2/3 is at the beginning of Phase 2 after the last job from $\mathcal{J}'$ is scheduled and at the end of Phase 2 when only one job is running. Therefore $T_{<\frac{2}{3}} \leq 2T_{\max}$ and by Lemma 7.1, $T \leq (2 + \frac{1-2/3}{2/3})T_{\text{opt}} = 2.5T_{\text{opt}}$.

If the efficiency at the time when the last job from $\mathcal{J}'$ finishes is more than 1/2, we know that the efficiency is at least 1/2 except for the time when only the last job from $\mathcal{J}''$ is running: it is at least 2/3 during Phase 1, it is at least 1/2 before until the last job form $\mathcal{J}'$ finishes, and then the

second job from $\mathcal{J}''$ is scheduled by the step (6) as soon as the efficiency drops under 1/2, and the efficiency is again at least 2/3, as two jobs from $\mathcal{J}''$ are scheduled simultaneously. Therefore $T_{<\frac{1}{2}} \leq T_{\max}$ and by Lemma 7.1, $T \leq (1 + \frac{1-1/3}{1/2})T_{\text{opt}} \leq 2.5T_{\text{opt}}$. □

11 The two-dimensional mesh

11.1 Deterministic algorithms

Because of the more complex geometric structure of the two-dimensional mesh, the greedy approach does not work too well. A better strategy is to partition the jobs according to one dimension first, and then to schedule the jobs in the same partition using one of the algorithms for one-dimensional meshes from the previous section.

We first give some definitions and simple algorithms. Then we gradually build the optimal algorithm using the simple and less efficient algorithms as subprograms.

Throughout this section we work with an $n_1 \times n_2$ mesh of $N = n_1 n_2$ processors. A job requiring a $a_i \times b_i$ mesh with running time is represented as (a_i, b_i, t_i) (of course, on-line algorithms do not know t_i). We assume that $n_1 \geq n_2$ and that for each job $a_i \geq b_i$ without loss of generality.

We first describe a simple modification of the algorithm ORDERED. It treats the jobs as one-dimensional, and thus it is efficient only if the heights of all the jobs are within a small range. Let b denote the maximal height of a job, $b = \max\{b_i | (a_i, b_i, t_i) \in \mathcal{J}\}$.

Algorithm CLASS

Partition the mesh into $\lfloor n_2/b \rfloor$ submeshes of size $n_1 \times b$:

Apply ORDERED to schedule $\mathcal{J}$ on these submeshes, disregarding the second dimension of the jobs and the submeshes, viewing them as one-dimensional.

Lemma 11.1 *Suppose that the height of any job is more than $b/2$.*

(i) If the dimensions of all jobs and of the machine are powers of two, the algorithm CLASS is 2-competitive and its efficiency is 1 as long as there are unscheduled jobs.

(ii) For general dimensions, the algorithm CLASS is 9-competitive and its efficiency is larger than $1/8$ *as long as there are unscheduled jobs.*

Proof. If the dimensions are powers of two, the efficiency does not decrease by disregarding the second dimension, as it is the same for all jobs. For general jobs, the efficiency decreases by less than factor of two, since we assume that the second dimension of every job is more than $b/2$, and by an additional factor smaller than two because of rounding when we divide the mesh. The rest follows from Theorem 10.1. □

Let $\mathcal{J}$ be a set of two-dimensional jobs. Define a partition of $\mathcal{J}$ into *job classes* $\mathcal{J} = \mathcal{J}^{(0)} \cup \ldots \cup \mathcal{J}^{(\lfloor \log n_2 \rfloor)}$ by $\mathcal{J}^{(l)} = \{(a_i, b_i, t_i) \in \mathcal{J} | n_2/2^{l+1} < b_i \leq n_2/2^l\}$. Define the *order* of $\mathcal{J}$ to be the number of nonempty job classes, $\text{order}(\mathcal{J}) = \left|\{\mathcal{J}^{(l)} | \mathcal{J}^{(l)} \neq \emptyset\}\right|$.

The algorithm CLASS is efficient if the job system has only one job class, but not for general job systems. The simplest way to use it for general job systems is to schedule the classes one by one. This leads to the following $O(\log N)$-competitive algorithm.

Algorithm SERIAL

for $l := 0$ to $\log n_2$ **do**

Apply CLASS to schedule the jobs from $\mathcal{J}^{(l)}$.

Lemma 11.2 *(i) If the dimensions of all jobs and of the mesh are powers of two, the algorithm SERIAL is* $(\text{order}(\mathcal{J}) + 1)$*-competitive.*

(ii) For general jobs, the algorithm SERIAL is $(\text{order}(\mathcal{J}) + 8)$*-competitive.*

Proof. By Lemma 11.1 the time when the efficiency is low (less than 1 if dimensions are powers of two, less than $1/8$ for general dimensions) is at most $T_{\max}$ for every nonempty class, the total of $\text{order}(\mathcal{J})T_{\max}$. Lemma 7.1 finishes the proof. □

To achieve a lower competitive ratio, we schedule multiple job classes in parallel in different submeshes. However, then the large jobs may not fit into our submeshes. We solve this problem by handling the large jobs separately. We use the previous algorithm to schedule a small number of classes which contain the large jobs. This approach leads to the following $O(\log \log N)$-competitive algorithm.

Algorithm PARALLEL

(1) Use SERIAL to schedule $\bigcup_{l=0}^{\lfloor \log(\text{order}(\mathcal{J})) \rfloor} \mathcal{J}^{(l)}$;

let $i := 1$;

repeat

(2) **let** $\mathcal{J}_i$ be all jobs that have not been scheduled yet;

let $h_i := \text{order}(\mathcal{J}_i)$;

partition the mesh into h_i submeshes of size $n_1 \times \lfloor n_2/h_i \rfloor$;

to each nonempty job class $\mathcal{J}_i^{(l)}$ assign one of these submeshes and denote it by $G_{i,l}$;

(3) apply CLASS in parallel to each nonempty class $\mathcal{J}_i^{(l)}$ on $G_{i,l}$ until the first time when the total efficiency of the running jobs (with respect to the whole $n_1 \times n_2$ mesh) is less than $1/16$;

(4) **wait**;

let $i := i + 1$;

until all jobs are scheduled.

Theorem 11.3 *Let S be a schedule generated by* PARALLEL *for a job system $\mathcal{J}$. Then $T(S) \leq O(\log(\text{order}(\mathcal{J})))T_{\text{opt}}(\mathcal{J})$. In particular, the algorithm* PARALLEL *is $O(\log \log N)$-competitive for scheduling on a two-dimensional mesh of N processors.*

Proof. For every i and l the submesh $G_{i,l}$ is large enough to fit any job from $\mathcal{J}^{(l)}$, since $l > \log(\text{order}(\mathcal{J}))$ if $\mathcal{J}^{(l)}$ is nonempty after the step (1), and therefore the smaller dimension of any job in $\mathcal{J}^{(l)}$ is at most $n_2/\text{order}(\mathcal{J}) \leq n_2/h_i$.

By Lemma 11.2, step (1) takes time at most $(\log(\text{order}(\mathcal{J})) + 8)T_{\text{opt}}$ by

To bound the time spent in the loop of steps (2) to (4), we first prove that $h_{i+1} \leq h_i/2$ for all i during the execution of PARALLEL. If there are unscheduled jobs in $\mathcal{J}_i^{(l)}$ at the end of step (3), then by Lemma 11.1 the efficiency of the schedule for the jobs in $\mathcal{J}_i^{(l)}$ (with respect $G_{i,l}$) is greater than 1/8. Therefore, the total efficiency at the end of step (3) is at least $h_{i+1}/8h_i$. On the other hand, by the condition in (3), the efficiency is less than 1/16. Hence $h_{i+1} \leq h_i/2$.

It follows that the number of passes through steps (2) to (4) is at most $\log(\text{order}(\mathcal{J})) + 1$. Since the time spent in each pass through step (4) is bounded by $T_{\max}$ and the efficiency during the step (3) is at least 1/16, the total time spent in steps (2) to (4) is by Lemma 7.1 bounded by $(\log(\text{order}(\mathcal{J})) + 17)T_{\text{opt}}$.

Therefore the length of the schedule is at most $O(\log(\text{order}(\mathcal{J})))T_{\text{opt}} = O(\log\log N)T_{\text{opt}}$, which finishes the proof. □

Now we construct an $O(\sqrt{\log\log N})$-competitive on-line scheduling algorithm for the two-dimensional mesh. We use the previous algorithm PARALLEL to schedule the large jobs.

The improvement of this algorithm over PARALLEL is in making an optimal tradeoff between the average efficiency of the schedule and the amount of time that the efficiency of the schedule is below average. To achieve this tradeoff, we schedule new jobs in a small part of the mesh, instead of using the whole machine as in PARALLEL. This ensures that when the efficiency is too low, we can use the next part immediately, instead of waiting for all running jobs to finish.

In Section 11.3 we prove a matching lower bound. The proof of the lower bound shows that this non-intuitive strategy of using only a part of the mesh is in fact necessary to achieve the optimal competitive ratio. If we try to use the whole mesh, we cannot get a much better competitive ratio that $O(\log\log N)$ of the algorithm PARALLEL.

Let $k = \left\lceil \sqrt{\log(\text{order}(\mathcal{J}))} \right\rceil = O(\sqrt{\log\log N})$. We partition the mesh into k submeshes of size $n_1 \times \lfloor n_2/k \rfloor$, denoted by G_j, $1 \leq j \leq k$ (see Figure 3).

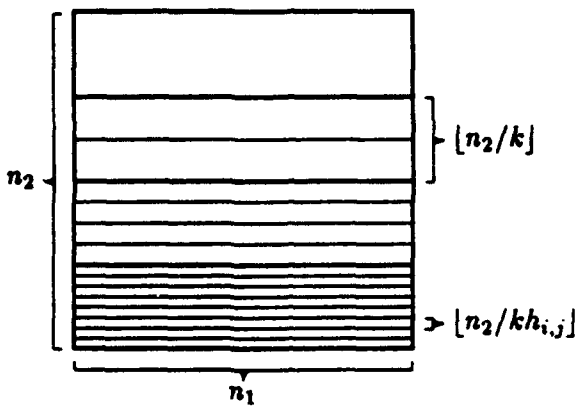


Figure 3: *The partition of the mesh used in the algorithm BALANCED PARALLEL. (The thick lines partition the mesh into submeshes G_j, the thin lines partition them further into submeshes $G_{i,j,l}$.)*

Algorithm BALANCED PARALLEL

(1) Use PARALLEL to schedule $\bigcup_{l=0}^{\lfloor 2 \log\log N \rfloor} \mathcal{J}^{(l)}$;

let $i = 1$;

repeat

for $j := 1$ to k **do begin**

(2) **let** $\mathcal{J}_{i,j}$ be all jobs that have not been scheduled yet;

let $h_{i,j} :=$ order($\mathcal{J}_{i,j}$);

partition the mesh into $h_{i,j}$ submeshes of size $n_1 \times \lfloor n_2/kh_{i,j} \rfloor$;

to each nonempty job class $\mathcal{J}_{i,j}^{(l)}$ assign one of these submeshes and denote it by $G_{i,j,l}$ (see Figure 3);

(3) apply CLASS in parallel to each nonempty class $\mathcal{J}_{i,j}^{(l)}$ on $G_{i,j,l}$ until the first time when the efficiency of all currently running jobs with respect to G_j is less than $1/16$, then interrupt all instances of CLASS (but allow the current jobs to be processed);

end;

(4) **wait**;
let $i := i + 1$;
until all jobs are scheduled.

Theorem 11.4 *The algorithm* BALANCED PARALLEL *is* $O(\sqrt{\log\log N})$*-competitive for scheduling on a two-dimensional mesh of* N *processors.*

Proof. For every i, j and l the submesh $G_{i,j,l}$ is large enough to fit any job from $\mathcal{J}^{(l)}$, since $l > 2\log\log N$ if $\mathcal{J}^{(l)}$ is nonempty after the step (1), and therefore the smaller dimension of any job in $\mathcal{J}^{(l)}$ is at most $n_2/(\log N)^2 \leq n_2/kh_i$.

By Theorem 11.3, step (1) takes time at most $O(\log(\log\log N))T_{\mathbf{opt}} \leq O(\sqrt{\log\log N})T_{\mathbf{opt}}$.

As in the proof of Theorem 11.3, $h_{i,j}$ decreases by a factor of 2 after each pass through the steps (2) to (3). That means that during each pass through the repeat-until loop it decreases by at least a factor of 2^k and therefore there can be at most $\lceil \log(\text{order}(\mathcal{J}))/k \rceil = O(\sqrt{\log\log N})$ passes through the step (4). Since the time spent in each pass through step (4) is bounded by $T_{\mathbf{max}}$ and the efficiency with respect to the whole mesh during the step (3) is at least $1/16k = \Omega(1/\sqrt{\log\log N})$, the total time spent in steps (2) to (4) is by Lemma 7.1 bounded by $O(\sqrt{\log\log N})T_{\mathbf{opt}}$. □

Note that we have actually proved that the competitive ratio is at most $O(\sqrt{\log\log n_2})$, which is smaller than $O(\sqrt{\log\log N})$ if n_2 is much smaller than n_1.

11.2 Off-line scheduling

In this section we prove that for any job system $\mathcal{J}$, $T_{\mathbf{opt}}(\mathcal{J})$ is within a constant factor of $\max(T_{\mathbf{eff}}(\mathcal{J}), T_{\mathbf{max}}(\mathcal{J}))$. Intuitively, this means that if there are no long jobs, we can schedule all the jobs so that the average efficiency is at least some constant.

We use this result twice. In Section 11.3 we prove that no on-line algorithm can guarantee that the length of a schedule S it generates is bounded by $T(S) = o(\sqrt{\log\log N})\max(T_{\mathbf{eff}}, T_{\mathbf{max}})$. To derive a lower bound on the

competitive ratio, we will use the result of this section. In Section 11.4 we use the same ideas as a part of our on-line randomized algorithm.

To make our exposition simpler, we first assume that the dimensions of all the jobs and the machine are powers of two and the machine is a square $n \times n$ mesh. Let $m_l = n/2^l$. In contrast to the on-line deterministic algorithm, we now divide the jobs into classes according to their larger dimension, $\mathcal{J}^{(l)} = \{(a_i, b_i, t_i) \in \mathcal{J} | m_{l+1} < a_i \leq m_l\}$.

The idea of our algorithm is to assign to each class an area proportional to the work in that class, and then schedule each class using the on-line algorithm ORDERED. We have to ensure that every job fits in the area assigned to it. To achieve this, we schedule the class $\mathcal{J}^{(0)}$ separately, and require the area assigned to class $\mathcal{J}^{(l)}$ to be a union of several square submeshes of size $n/2^l$. This rounding requires some additional area, which can be bounded by a third of the size of the machine.

The schedule is generated by the following off-line algorithm. W_l denotes the work of $\mathcal{J}^{(l)}$, W denotes the total work of all classes except $\mathcal{J}^{(0)}$, and z_l denotes the number of $m_l \times m_l$ meshes assigned to $\mathcal{J}^{(l)}$.

Algorithm OFFLINE

(1) schedule $\mathcal{J}^{(0)}$ using the algorithm ORDERED;

(2) for each $l > 0$, **let** $W_l := \sum_{(a_i,b_i,t_i) \in \mathcal{J}^{(l)}} a_i b_i t_i$;
let $W := \sum_{l>0} W_l$;
for each $l > 0$, **let** $z_l := \lceil \frac{2}{3} W_l / m_l^2 W \rceil$;

(3) choose disjoint square submeshes $H_{l,j}$, $l > 0$, $j = 1, \ldots, z_l$ of size $m_l \times m_l$ (see below for details).

(4) in parallel for each l schedule the class $\mathcal{J}^{(l)}$ on the collection of grids $\{H_{l,0}, \ldots, H_{l,z_l}\}$ using ORDERED;

We need to justify the step (3). Since the meshes are squares whose sizes are powers of two, we can place them greedily starting with the largest mesh, so that the coordinates of each mesh are divisible by its size. See Figure 4 for an example.

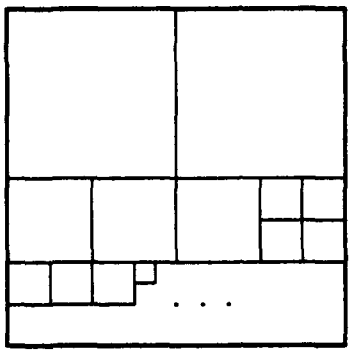


Figure 4: *An example of the partition of the mesh used in the algorithm OFFLINE.*

This process will place all the meshes as long as the number of processors is not too large, which is true in our case since

$$\sum_{l>0} z_l m_l^2 \leq \sum_{l>0} \left(\frac{2W_l}{3m_l^2 W} + 1 \right) m_l^2 \leq \sum_{l>0} \frac{2W_l}{3\sum_l W_l} + \sum_{l>0} m_l^2 \leq \frac{2}{3}n^2 + \frac{1}{3}n^2 = n^2.$$

For this algorithm we cannot prove that the time when the efficiency is low is short. Instead, we prove a similar claim about the average efficiency. More precisely, we prove that the length of the schedule is bounded by a constant multiple of the work divided by the number of processors, plus an additive term bounded by a constant multiple of the longest running time.

Theorem 11.5 *(i) If the dimensions of the jobs and of the machine are powers of two and the machine is a square mesh, the algorithm OFFLINE produces a schedule S whose length is bounded by $T(S) \leq 3.5 \max(T_{\text{eff}}, T_{\max})$.*

(ii) For general dimensions, there exists an off-line algorithm which produces a schedule S whose length is bounded by $T(S) = O(\max(T_{\text{eff}}, T_{\max}))$.

Proof. (i) We prove for each $l > 0$ that the instance of ORDERED scheduling $\mathcal{J}^{(l)}$ in step (4) finishes after the time at most $3W/2n^2 + T_{\max}$. Suppose for a contradiction that for some l this is not the case. Then for time more

than $3W/2n^2$ there are jobs available, and hence by Lemma 11.1 the efficiency relative to the area assigned to $\mathcal{J}^{(l)}$ is 1. Thus the work done by jobs in $\mathcal{J}^{(l)}$ is more than $(3W/2n^2)z_l m_l^2 \geq W_l$, a contradiction. Therefore the step (4) takes time at most $3W/2n^2 + T_{\max}$.

During the step (1), the efficiency is 1 except for time $T_{\max}$. Let W' be the total work including $\mathcal{J}^{(0)}$. Then the length of the schedule is bounded by $3W'/2n^2 + 2T_{\max} = \frac{3}{2}T_{\text{eff}} + 2T_{\max}$.

(ii) For general dimensions, we first schedule large jobs. Then we round the dimensions of jobs to the next higher power of two and the size of the mesh to the next smaller power of two, and proceed as before. This changes the efficiency by a constant factor only.

If the mesh is not a square, we partition the jobs into classes according to their smaller dimension. Instead of square meshes we then assign the area to each class in submeshes whose longer dimension is the longer dimension of the machine. This is less efficient, since we use larger area for rounding the area assigned to each class to these submeshes, but the difference is again only in the constant factor. □

11.3 A lower bound on deterministic scheduling

We prove that no on-line scheduling algorithm on an $n \times n$ mesh of N processors can achieve a competitive ratio better than $\varepsilon\sqrt{\log \log N}$ for some $\varepsilon > 0$. This proves that the algorithm in Section 11.1 is optimal up to a constant factor. (Note that it is sufficient to prove the lower bound for square meshes.)

To prove this lower bound we use an adversary as introduced in Section 7. We specify the number of jobs and their job types, and then design an adversary who assigns the running times depending on the action of the scheduler.

The adversary tries to restrict the possibilities of the scheduler so that he has to act similar to the optimal algorithm we presented in Section 11.1. The key technical point is Lemma 11.6. It shows that the adversary can restrict the actions of the scheduler substantially. More precisely, if the efficiency is high, the adversary is able to find a small subset of the running jobs which

effectively block a large portion of the mesh. This implies that new jobs have to take new space, and eventually there is no more space available and the scheduler must wait until the running jobs are stopped.

Of course, the adversary cannot go on forever. The price he pays is that after each such step the number of distinct sizes of available jobs is reduced. Nevertheless, it is possible to repeat this process sufficiently many times before all available job sizes are eliminated.

11.3.1 Notation

For the proof of the lower bound it is convenient to represent meshes as rectangles with both coordinates running through the real interval $[0, n]$. A processor then corresponds to a unit square and a $x \times y$-submesh at (X, Y) corresponds to the $x \times y$ rectangle with the lower left corner at (X, Y).

During the proofs we will also use rectangles with non-integer dimensions and coordinates. We say that a rectangle R' *intersects* a set of rectangles $\mathcal{R}$, if the area of $R' \cap R$ is not zero for some $R \in \mathcal{R}$.

A *normal* $x \times y$*-rectangle* is a rectangle with width x and height y with the lower left corner at (X, Y) such that X is an integer multiple of x and Y is an integer multiple of y. A *normal* (x, y)*-rectangle* is a normal $x \times y$- or $y \times x$-rectangle.

Observe that any two normal $x \times y$-rectangles are disjoint and that the any rectangle larger than $x \times y$ (in particular the $n \times n$ mesh in our case) can be partitioned into a set of non-intersecting normal $x \times y$-rectangles and a small leftover.

11.3.2 The scheduling problem

Now we are ready to specify the job system used for the lower bound proof.

Let $k = \lfloor \frac{1}{3}\sqrt{\log \log N} \rfloor$, $s = \lceil (\log \log N)^2 \rceil$, $t = \lfloor \frac{1}{2} \log_s n \rfloor$.

We have $t + 1$ different job classes, $\mathcal{J} = \mathcal{J}_0 \cup \ldots \cup \mathcal{J}_t$. The job class $\mathcal{J}_j$ contains nk^2 jobs of size $\frac{n}{s^j} \times s^j$ (the running times of the jobs will be determined dynamically by the adversary depending on the actions of the on-line scheduler). Note that for $i \leq j \leq t$, $\frac{n}{s^i} \geq \frac{n}{s^j} \geq s^j \geq s^i$.

Suppose we use an on-line scheduling algorithm to schedule this job system. For $I \subseteq [0..t]$, let $\mathcal{C}(I)$ denote the set of all submeshes corresponding to currently running jobs from $\mathcal{J}_j$, $j \in I$. Let $\mathcal{C} = \mathcal{C}([0..t])$ denote the set of submeshes corresponding to all currently running jobs.

11.3.3 Adversary strategy

The adversary strategy is based on the following lemma which is proved in Section 11.3.4.

Lemma 11.6 *Suppose that we are scheduling the job system from Section 11.3.2. Then at any time of the schedule and for any interval $I' = [a..b]$, $0 \leq a \leq b \leq t$, and $\overline{I'} = [0..t] - I'$, there exists a set $\mathcal{D} = \mathcal{D}(I') \subseteq \mathcal{C}(\overline{I'})$ such that $\mathrm{eff}(\mathcal{D}) \leq 1/8k$ and every normal $(\frac{n}{4ks^b}, \frac{s^a}{4k})$-rectangle intersected by $\mathcal{C}(\overline{I'})$ is also intersected by $\mathcal{D}$.*

The adversary maintains an *active interval* denoted by I. Initially $I = [0..t]$ and with time I gradually gets smaller. Let T be some fixed time. The adversary reacts to the scheduler's actions according to the following steps.
SINGLE JOB: If the scheduler schedules some job on a submesh with an area smaller than n/k, then the adversary removes all other jobs (both running and waiting ones) and runs this single job for a sufficient amount of time.
DUMMY: If the scheduler starts a job that does not belong to a job class in the active interval (a job from $\mathcal{J}_j$, $j \notin I$), then the adversary removes it immediately.
CLEAN UP: If the time since the last CLEAN UP step (or since the beginning of the schedule) is equal to T and there was no SINGLE JOB step, then the adversary removes all running jobs, i.e., assigns their running times so that they are completed at this point.
DECREASE EFFICIENCY: If $\mathrm{eff}(\mathcal{C})$ exceeds $1/k$, the adversary does the following: He takes an interval $I' \subseteq I$ such that $|I'| = \lfloor |I|/2 \rfloor$ and $\mathrm{eff}(\mathcal{C}(I')) \leq \mathrm{eff}(\mathcal{C}(I))/2$ (such I' obviously exists: either the upper or the lower half of I, whichever has lower efficiency). Then he computes $\mathcal{D}(I')$

according to Lemma 11.6 and removes all jobs except those from $\mathcal{C}(I') \cup \mathcal{D}(I')$. He then sets the active interval to I'.

11.3.4 Evaluation of the Adversary Strategy

In this section we prove that the adversary strategy from the previous section ensures that $T(S) \geq k \cdot \max(T_{\text{eff}}(\mathcal{J}), T_{\max}(\mathcal{J}))$. By Theorem 11.5 this is sufficient to prove the lower bound on the competitive ratio.

If the scheduler starts a job that does not belong to a job class in the active interval then immediately removing it by a DUMMY step essentially does not change the schedule. If the scheduler allows a SINGLE JOB step, the scheduler used a simulation factor greater than k to schedule this job. Therefore in an optimal schedule the running time of this job will be more than k times shorter. The adversary assigns a sufficiently large time to this job, and thus guarantees that the scheduler is not k-competitive.

So we assume that the scheduler always starts jobs from the active interval and the adversary only performs DECREASE EFFICIENCY and CLEAN UP steps. The CLEAN UP steps divide the schedule into *phases*.

The lower bound proof follows this outline. We first prove Lemma 11.6 which justifies the DECREASE EFFICIENCY step and then we show that each phase can have at most $6k$ of these steps. This implies that every schedule has to have at least k phases thus proving that $T(S) \geq kT \geq kT_{\max}(\mathcal{J})$. Because the efficiency is at most $1/k$ during the entire schedule, we get $T(S) \geq kT_{\text{eff}}(\mathcal{J})$.

The next claim is the key to the proof of Lemma 11.6. It states a purely geometrical fact which is true for an arbitrary set of submeshes $\mathcal{D}$, but we will only use it for $\mathcal{D}$ being a subset of running jobs.

Claim 11.7 *Let y and v be given and let $\mathcal{D}$ be a set of submeshes with height at most y/v. Then there exists a $\mathcal{D}' \subseteq \mathcal{D}$ such that $\text{eff}(\mathcal{D}') \leq 2/v$ and each normal $1 \times y$-rectangle intersected by $\mathcal{D}$ is also intersected by $\mathcal{D}'$.*

Proof. Let R be a normal $n \times y$-rectangle. We will define $\mathcal{D}_R \subseteq \mathcal{D}$ such that $\text{eff}(\mathcal{D}_R) \leq 2y/vn$ and it intersects all columns of R intersected by $\mathcal{D}$. It is

then sufficient to set $\mathcal{D}' = \bigcup_R \mathcal{D}_R$, since every normal $1 \times y$-rectangle is a column of some normal $n \times y$-rectangle. Because there are only $\lfloor \frac{n}{y} \rfloor$ normal $n \times y$-rectangles the efficiency of D' is $\text{eff}(\mathcal{D}') \leq \lfloor n/y \rfloor 2y/vn \leq 2/v$.

$\mathcal{D}_R$ is obtained by a sweep over the mesh. Let $D_i \in \mathcal{D}$ be the submesh which intersects the ith column and has the largest right coordinate of all such submeshes (D_i is undefined if no such submesh exists). Define a sequence $\mathcal{D}_0 = \emptyset, \mathcal{D}_1, \ldots, \mathcal{D}_n = \mathcal{D}_R$ by

$$\mathcal{D}_i = \begin{cases} \mathcal{D}_{i-1} \cup \{D_i\} & \text{if the } i\text{th column of } R \text{ intersects } \mathcal{D} \text{ but not } \mathcal{D}_{i-1}, \\ \mathcal{D}_{i-1} & \text{otherwise.} \end{cases}$$

From the way we choose D_i it follows that no column of R is intersected by more than two submeshes of $\mathcal{D}_R$. Since the height of every submesh of $\mathcal{D}_R$ is at most y/v, we have $\text{eff}(\mathcal{D}_R) \leq 2y/vn$. □

Proof of Lemma 11.6. Divide $\mathcal{C}(\overline{I'})$ into the following three parts:

- $\mathcal{C}_1$ contains all submeshes whose heights are at most s^{a-1},
- $\mathcal{C}_2$ contains all submeshes whose heights as well as widths are at most n/s^{b+1} and
- $\mathcal{C}_3$ contains all submeshes whose widths are at most s^{a-1}.

All submeshes from $\mathcal{C}([0..(a-1)])$ are either in $\mathcal{C}_1$ or in $\mathcal{C}_3$ depending on their orientation and all submeshes of $\mathcal{C}([(b+1)..t])$ are in $\mathcal{C}_2$, hence $\mathcal{C}(\overline{I'}) = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$.

We apply Claim 11.7 four times to obtain the following sets:

- $\mathcal{D}_1 \subseteq \mathcal{C}_1$, which intersects all normal $1 \times \frac{s^a}{4k}$-rectangles intersected by $\mathcal{C}_1$;
- $\mathcal{D}_2 \subseteq \mathcal{C}_2 \cup \mathcal{C}_3$, which intersects all normal $\frac{n}{4ks^b} \times 1$-rectangles intersected by $\mathcal{C}_2 \cup \mathcal{C}_3$;
- $\mathcal{D}_3 \subseteq \mathcal{C}_1 \cup \mathcal{C}_2$, which intersects all normal $1 \times \frac{n}{4ks^b}$-rectangles intersected by $\mathcal{C}_1 \cup \mathcal{C}_2$;
- $\mathcal{D}_4 \subseteq \mathcal{C}_3$, which intersects all normal $\frac{s^a}{4k} \times 1$-rectangles intersected by $\mathcal{C}_3$.

Hence $\mathcal{D}_1 \cup \mathcal{D}_2$ intersects all normal $\frac{n}{4ks^b} \times \frac{s^a}{4k}$-rectangles intersected by $\mathcal{C}(\overline{I'})$ and $\mathcal{D}_3 \cup \mathcal{D}_4$ intersects all normal $\frac{s^a}{4k} \times \frac{n}{4ks^b}$-rectangles intersected by $\mathcal{C}(\overline{I'})$.

In all four cases we have $v = s/4k$. So setting $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \mathcal{D}_3 \cup \mathcal{D}_4$ gives us an efficiency $\text{eff}(\mathcal{D}) \leq 32k/s \leq 1/8k$ for sufficiently large n. □

The next claim is the key to the proof of the lower bound. It shows that the adversary strategy does not allow the scheduler to reuse the space efficiently.

Claim 11.8 *Let $j \in I$ and R be a normal $(\frac{n}{4ks^j}, \frac{s^j}{4k})$-rectangle that does not intersect $\mathcal{C}$ at this step of the schedule. Then during the current phase R has never intersected $\mathcal{C}$.*

Proof. The DECREASE EFFICIENCY step is the only step removing jobs during a phase, and hence it is sufficient to prove that no previous DECREASE EFFICIENCY step removed all the jobs which intersected R.

Assume that the active interval before such a previous DECREASE EFFICIENCY step is $[a\,..\,b]$. Since the active interval never grows, we have $a \leq j \leq b$. Because the dimensions of R are integer multiples of the dimension of the normal rectangles from Lemma 11.6 as used in that DECREASE EFFICIENCY step, R can be partitioned into a set $\mathcal{R}$ of such rectangles without any leftover. According to the assumption, the rectangle R, and hence some rectangle from $\mathcal{R}$, is intersected by $\mathcal{C}$ before the DECREASE EFFICIENCY step. But then by Lemma 11.6 this rectangle, and hence also R, is also intersected by $\mathcal{C}$ after the DECREASE EFFICIENCY step. □

Claim 11.9 *Each phase can have at most $6k$ DECREASE EFFICIENCY steps.*

Proof. Suppose that the scheduler starts a job from $\mathcal{J}_j$, $j \in I$, in a submesh C. As this does not cause a SINGLE JOB step, the job is scheduled on a submesh with dimensions at least $\frac{n}{ks^j}$ and $\frac{s^j}{k}$. Thus at least half of that submesh consists of normal $(\frac{n}{4ks^j}, \frac{s^j}{4k})$-rectangles. It follows from the previous claim that at least half of the area could not have been used so far during the current phase.

The efficiency immediately before a DECREASE EFFICIENCY step is at most $1/k+1/n$ (since the threshold $1/k$ was reached by the last job started). By the construction and Lemma 11.6, the efficiency after the DECREASE EFFICIENCY step is at most $1/2k + 1/2n + 1/8k \leq 2/3k$ for sufficiently large n.

This implies that between two DECREASE EFFICIENCY steps the scheduler has to schedule jobs of the current active interval which will increase the efficiency by at least $1/3k$ using at least $1/6k$ of the area that was not yet used in this phase. This can be done at most $6k$ times. □

Theorem 11.10 *The adversary strategy forces that for every schedule S generated by the on-line scheduler*

$$T(S) \geq \frac{1}{3}\sqrt{\log\log N} \cdot \max(T_{\text{eff}}(\mathcal{J}), T_{\max}(\mathcal{J})).$$

Proof. We first prove that after k phases the active interval I is still nonempty. Each DECREASE EFFICIENCY step halves I while all other steps leave it unchanged. In k phases there are at most $6k^2 \leq \frac{2}{3}\log\log N$ DECREASE EFFICIENCY steps. At the beginning the length of I is $t+1 \geq \frac{1}{2}\log_s n = \frac{\log n}{2\log\lceil(\log\log N)^2\rceil} > 2^{\lceil\frac{2}{3}\log\log N\rceil}$ for sufficiently large n. So the active interval cannot be empty after k phases.

If j is in the active interval at the end of the kth phase then it was in the active interval during all k phases and the adversary could remove the jobs of $\mathcal{J}_j$ only in the k CLEAN UP steps. During one CLEAN UP step he could remove at most nk of such jobs (since each job has area at least n/k), hence some of them are not finished before the end of the kth phase. Hence $T(S) \geq kT \geq kT_{\max}(\mathcal{J})$.

During the entire schedule the efficiency was at most $1/k$, hence $T(S) \geq kT_{\text{eff}}(\mathcal{J})$. Therefore $T(S) \geq k \cdot \max(T_{\text{eff}}(\mathcal{J}), T_{\max}(\mathcal{J}))$. □

This together with Theorem 11.5 gives the following theorem.

Theorem 11.11 *The competitive ratio of any on-line scheduling algorithm for a mesh of N processors is at least $\Omega(\sqrt{\log\log N})$.* □

11.4 Randomized scheduling

In this section we give a constant competitive randomized algorithm for two-dimensional meshes. The competitive ratio is 28 if the dimensions of all the jobs and of the machine are powers of two; otherwise it can be bounded by 44.

The basic idea of the algorithm is the following. We partition the jobs according to their size. For each size we schedule a random sample of jobs and estimate the total work of the jobs of that size. Then we partition the mesh so that each job size is assigned an area proportional to the estimated work of jobs of that size and schedule all jobs of given size in the assigned area.

There are some issues we have to deal with. We need to have an estimate on the longest running time, as this is crucial for any sampling. To solve this problem, we assume that the longest job is at most twice as long as the longest job we have seen so far. If we see a longer job, we abort our current attempt and start from the beginning while doubling our estimate. We bound the time of the schedule by a sum of a term proportional to the work done and a term which is bounded by a constant multiple of our estimate of the longest running time. If we sum the bounds for the parts of schedule with different estimates, the sum of the first terms is still proportional to the work, while the second terms are a geometric sequence and hence it is bounded by a constant multiple of the longest running time and our doubling strategy works.

Even if we have ·ect bound on the longest running time, sampling is not trivial. We ha ·uarantee that our sample is sufficiently good while keeping the time required for sampling small. Sampling a fixed number or fixed fraction of jobs does not work—for some size we can have many jobs with small running time and a few very long jobs; in that case we are not likely to see any long job and then we have no useful information about the total work. Instead, we sample until we see jobs with total running time exceeding some bound. Intuitively, if there are only two long jobs in the first quarter of the jobs, it is likely that the number of long jobs is close to eight; if there are two long jobs among the first four, probably about a half of the

jobs are long. Even though we have a much larger sample in the first case, Hoeffding bounds guarantee that in both cases the probability of a wrong estimate is about the same. Note that while sampling in this way, we may schedule most or all jobs before the bound is reached.

In addition, it is necessary to guarantee that we get good estimates for all of different sizes of jobs at once. This is important and non-trivial, since the number of different sizes is not constant. To achieve this, we partition the mesh for sampling so that we schedule in parallel larger number of smaller jobs. Therefore we have a larger and more reliable sample for smaller jobs, and the total probability of an error can be bounded.

11.4.1 The algorithm

To make our exposition more simple and clear, we assume that the mesh is a square and its size is a power of 2 and the sizes of each job are powers of 2. Since this often true in practice, and the competitive ratio is somewhat better, this simpler case can be of independent interest. These assumptions can be eliminated in following way. We can round the sizes of jobs to the next larger power of two and use a submesh whose sizes are multiples of the modified sizes of all jobs (processing the large jobs separately); this changes the efficiency and the competitive ratio by a constant factor. It is also not difficult to modify the algorithm to handle non-square meshes.

Let $m_l = n/2^l$. We define the job classes (as in Section 11.2) by $\mathcal{J}^{(l)} = \{(a_i, b_i, t_i) \in \mathcal{J} | m_{l+1} < a_i \leq m_l\}$, and subclasses by $\mathcal{J}^{(l,l')} = \{(a_i, b_i, t_i) \in \mathcal{J}^{(l)} | m_{l+l'+1} < b_i \leq m_{l+l'}\}$. Note that under our assumptions the size of all jobs in $\mathcal{J}^{(l,l')}$ is exactly $m_l \times m_{l+l'}$. Unless we say otherwise, l and l' range over $3 \leq l \leq \log n$, $0 \leq l' \leq \log n$. Note that all the subclasses with $l' = 0$ contain jobs requiring square meshes, for $l' = 1$ meshes with $2:1$ ratio of their dimensions, etc.

In step (1) of the algorithm we schedule the large jobs (the three job classes $\mathcal{J}^{(0)}$, $\mathcal{J}^{(1)}$, and $\mathcal{J}^{(2)}$). Steps (2) and (3) implement the doubling strategy to estimate the longest running time. At any point the estimate is $2^l\tau$.

Step (4) implements the sampling. It uses a fixed partition of the mesh illustrated by Figure 5. To each subclass $\mathcal{J}^{(l,l')}$, $l \geq 3$, $l' \geq 0$, we assign a submesh $G_{l,l'}$ of size $(4m_l)\times((l'+1)m_{l'}/4)$. Each $G_{l,l'}$ is divided into $(l'+1)2^l$ submeshes $G_{l,l',j}$ of size $m_l \times m_{l+l'}$. Hence $(l'+1)2^l$ jobs of each subclass $\mathcal{J}^{(l,l')}$ can be scheduled in parallel. We need to verify that these submeshes can be placed onto an $n\times n$ mesh so that they are pairwise disjoint. For a fixed l, the sum of the heights of the grids $G_{l,l'}$ is bounded by $\sum_{l'\geq 0}(l'+1)n/2^{l'+2} = n$, therefore they all fit into a submesh of size $4m_l \times n$. The total width of these meshes for $l \geq 3$ is bounded by n, hence all the meshes fit into the $n \times n$ mesh.

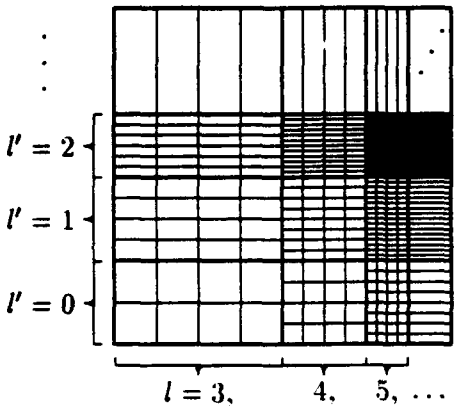


Figure 5: *The partition of the mesh used for sampling in the algorithm SAMPLE. (The thick lines partition the mesh into submeshes $G_{l,l'}$, the thin lines partition them further into submeshes $G_{l,l',j}$.)*

Step (5) computes the estimates based on previous sampling. For each l and l', $w_{l,l'}$ estimates the total running time of all jobs in $\mathcal{J}^{(l,l')}$ not scheduled before the step (4). Note that this also estimates the work (after rescaling), as the size of all jobs in $\mathcal{J}^{(l,l')}$ is the same. w_l estimates the total work of jobs in $\mathcal{J}^{(l)}$, and w estimates the total work of all unscheduled jobs. p_l is a number of processors proportional to w_l, z_l is the number of grids of size $m_l \times m_l$ actually assigned to the class $\mathcal{J}^{(l)}$ (this rounding guarantees that each job fits into the assigned mesh).

Steps (6) and (7) schedule the jobs in areas approximately proportional to the estimated work in each class, using a slight modification of the methods from Section 11.2. The condition in (7) and the step (8) guarantee that we sample again if some of the estimates turns out to be wrong.

Algorithm SAMPLE

(1) Schedule the classes $\mathcal{J}^{(0)}$, $\mathcal{J}^{(1)}$ and $\mathcal{J}^{(2)}$ using ORDERED;
while no job of nonzero time was scheduled **do**
schedule one job;
wait;
let τ be the maximal running time of the jobs scheduled so far;
let $I := 1$;

(2) During the steps (3) to (7),
if the running time of any job exceeds $2^I\tau$,
then begin let $I := I + 1$; **goto** (3) **end**;

(3) **wait**;

(4) For every l, l', **let** $r_{l,l'}$ be the number of unscheduled jobs in $\mathcal{J}^{(l,l')}$;
For time $2 \cdot 2^I\tau$, **do**
if $G_{l,l',j}$ is empty and there is an unscheduled job in $\mathcal{J}^{(l,l')}$
then schedule a random unscheduled job in $\mathcal{J}^{(l,l')}$ onto $G_{l,l',j}$;
wait;

(5) for every l, l', **if** there are unscheduled jobs in $\mathcal{J}^{(l,l')}$
then begin
let $k_{l,l'}$ be the smallest k such that the sum of the running times of the first k jobs from $\mathcal{J}^{(l,l')}$ scheduled during the preceding step (4) is at least $2(l'+1)2^l2^I\tau$;
let $w_{l,l'} := 2r_{l,l'}(l'+1)2^l2^I\tau/k_{l,l'}$;
end;
else let $w_{l,l'}$ be the total running time of jobs from $\mathcal{J}^{(l,l')}$ scheduled during the preceding step (4);

for every l, **let** $w_l := m_l \sum_{l'} w_{l,l'} m_{l+l'}$;

let $w := \sum_l w_l$;

for every l,

let $p_l := \frac{47}{48} n^2 w_l / w$;

let $z_l := \lceil p_l / m_l^2 \rceil$;

(6) Partition the machine into square submeshes $H_{l,j}$, $l > 0$, $1 \le j \le z_l$, of size $m_l \times m_l$;

(7) in parallel for each l schedule the class $\mathcal{J}^{(l)}$ on the collection of grids $\{H_{l,0}, \ldots, H_{l,z_l}\}$ using ORDERED, provided that

if ((all jobs of some class $\mathcal{J}^{(l)}$ are finished **and** the total work of the jobs $\mathcal{J}^{(l)}$ scheduled during this step is less than $w_l/4$)

or the time spent in this step is $\frac{48}{47} w/n^2$),

then interrupt all instances of ORDERED;

(8) **wait**;

if there are unscheduled jobs, **then goto** (4).

We need to justify the step (6). As in Section 11.2, it is sufficient to show that the total area of all the submeshes $H_{l,j}$ is not too large, since the meshes are squares whose sizes are powers of two. The area is bounded by

$$\sum_{l \ge 3} z_l m_l^2 \le \sum_{l \ge 3} (p_l / m_l^2 + 1) m_l^2 \le \sum_{l \ge 3} p_l + \sum_{l \ge 3} m_l^2 \le \frac{47}{48} n^2 + \frac{1}{48} n^2 = n^2.$$

11.4.2 Probability estimates

Our basic tool are Chernoff-Hoeffding bounds, which bound the probability that a sum of random variables differs significantly from its mean [Hoe63, HR90, ASE92]. Many sources state them only for a random variable $\mathbf{S}$ which is a sum of independent 0-1 variables $\mathbf{X}_i$. However, the original Hoeffding paper [Hoe63] proves that the same bounds are true even for more general variables $\mathbf{X}_i$. First, it is sufficient if the values of each $\mathbf{X}_i$ are from the real interval $[0, 1]$, not necessarily integers. Second, the variables can be produced by sampling without replacement from some universe, in which

case the variables are not independent; independent variables correspond to sampling with replacement. Intuitively, both of these changes should keep the value of $\mathbf{S}$ even closer to its mean. Note that in an extreme case of sampling without replacement we obtain all elements of the universe, in which case we know the sum exactly.

The most convenient form of the Hoeffding bounds states that for the appropriate $\mathbf{S}$ and any $0 \leq \varepsilon \leq 1$ the following holds

$$\text{Prob}[\mathbf{S} \leq (1-\varepsilon)\text{E}[\mathbf{S}]] \leq e^{-\varepsilon^2 \text{E}[\mathbf{S}]/2}$$

$$\text{Prob}[\mathbf{S} \geq (1+\varepsilon)\text{E}[\mathbf{S}]] \leq e^{-\varepsilon^2 \text{E}[\mathbf{S}]/3}$$

An elegant derivation for independent 0-1 variables can be found in [HR90]; to modify it for the more general $\mathbf{S}$ it is necessary to use convexity of the function $e^{h\mathbf{X}_i}$ and Jensen's inequality during the proof, see [Hoe63].

Sampling without replacement corresponds to the process by which we schedule the jobs within one subclass during the step (4) of the algorithm and hence the bounds apply in our case. However, we need the following variation in which we stop sampling after the sum of samples achieves a certain threshold. The threshold has to be smaller than the total sum, so that we do not run out of samples.

Lemma 11.12 *Let U be a set of r real numbers from $[0,1]$ with sum W. Let $\mathbf{X}_1, \ldots, \mathbf{X}_r$ be a sequence of random variables obtained by sampling without replacement out of the set U. Let $\mathbf{S}_i$ be the sum of the first i of them. Let $5 \leq \lambda < W$ be given. Let $\mathbf{k}$ be the unique integer such that $\mathbf{S}_{\mathbf{k}-1} < \lambda \leq \mathbf{S}_{\mathbf{k}}$ (note that $\mathbf{k}$ is a random variable). Then*

$$\text{Prob}[\neg(r\lambda/2\mathbf{k} \leq W \leq 2r\lambda/\mathbf{k})] \leq 2e^{-\lambda/6}$$

Proof. Let $\alpha = r\lambda/W$: intuitively α is the expected value of $\mathbf{k}$, i.e., the expected number of samples after which the threshold λ is reached. Note that $\text{E}[\mathbf{S}_i] = iW/r = i\lambda/\alpha$ for all $i \leq r$.

If $W < r\lambda/2\mathbf{k}$ then $\mathbf{k} < \alpha/2$ by the definition of α. From the definition of $\mathbf{k}$ it follows that for $\beta = \lfloor \alpha/2 \rfloor$, $\mathbf{S}_\beta \geq \lambda = \frac{\alpha}{\beta}\text{E}[\mathbf{S}_\beta]$. Using Hoeffding bound

we get

$$\text{Prob}[\mathbf{k} < \alpha/2] \leq \text{Prob}[\mathbf{S}_\beta \geq \frac{\alpha}{\beta}\mathrm{E}[\mathbf{S}_\beta]] \leq e^{-(1-\frac{\alpha}{\beta})^2\mathrm{E}[\mathbf{S}_\beta]/3} \leq e^{-\lambda/6}$$

since the exponent satisfies $(1-\frac{\alpha}{\beta})^2\mathrm{E}[\mathbf{S}_\beta] = (1-\frac{\alpha}{\beta})^2\frac{\beta}{\alpha}\lambda = (\frac{\alpha}{\beta}+\frac{\beta}{\alpha}-2)\lambda \geq \lambda/2$; the last inequality uses $\frac{\beta}{\alpha} \leq 1/2$ and the fact that $x + \frac{1}{x}$ decreases for $x < 1$.

If $W > 2r\lambda/\mathbf{k}$ then $\mathbf{k} > 2\alpha$. We put $\beta = \lfloor 2\alpha \rfloor$. Note that $\frac{\beta}{\alpha} \geq 2 - \frac{1}{\alpha} \geq 2 - \frac{1}{\lambda} \geq \frac{9}{5}$. By the definition of $\mathbf{k}$, $\mathbf{S}_\beta \leq \lambda = \frac{\alpha}{\beta}\mathrm{E}[\mathbf{S}_\beta]$. Using Hoeffding bound,

$$\text{Prob}[\mathbf{k} > 2\alpha] \leq \text{Prob}[\mathbf{S}_\beta \leq \frac{\alpha}{\beta}\mathrm{E}[\mathbf{S}_\beta]] \leq e^{-(\frac{\alpha}{\beta}-1)^2\mathrm{E}[\mathbf{S}_\beta]/2} \leq e^{-\lambda/6}$$

since $(\frac{\alpha}{\beta}-1)^2\mathrm{E}[\mathbf{S}_\beta] = (\frac{\alpha}{\beta}-1)^2\frac{\beta}{\alpha}\lambda = (\frac{\alpha}{\beta}+\frac{\beta}{\alpha}-2)\lambda \geq \lambda/3$; the last inequality uses $\frac{\beta}{\alpha} \geq 9/5$ and the fact that $x + \frac{1}{x}$ increases for $x > 1$. □

11.4.3 Expected time analysis

First we analyze one pass through the steps (4) to (8). Let us introduce some notation. Let $W_{l,l'}$ be the total running time of all unscheduled jobs in the subclass $\mathcal{J}^{(l,l')}$ before the step (4), $W_l = m_l \sum_{l'} m_{l+l'} W_{l,l'}$ the total work of all unscheduled jobs in the class $\mathcal{J}^{(l)}$ and $W = \sum_l W_l$ the total work of all unscheduled jobs. Let $W'_{l,l'}$, W'_l and W' be the same quantities restricted to the jobs actually scheduled during the steps (4) to (8). Note that $w_{l,l'}$, w_l and w are estimates of $W_{l,l'}$, W_l and W.

The first claim says that with large probability, after step (6) all the estimates are sufficiently good.

Claim 11.13 *If no running time is larger than $2^l\tau$, then after step (6),* $\text{Prob}[\neg((\forall l,l')w_{l,l'}/4 \leq W_{l,l'} \leq w_{l,l'})] \leq 1/6$, *and therefore* $\text{Prob}[\neg((\forall l)w_l/4 \leq W_l \leq w_l)] \leq 1/6$.

Proof. First we argue that for any given l and l',

$$\text{Prob}[\neg(w_{l,l'}/4 \leq W_{l,l'} \leq w_{l,l'})] \leq 2e^{-2(l'+1)2^l/6} \leq 2\alpha^{(l'+1)2^l/8},$$

where $\alpha = e^{-16/6} \approx 0.0695$

If $W_{l,l'} \leq 2(l'+1)2^l 2^I \tau$ then $w_{l,l'} = W_{l,l'}$ by the definition of $w_{l,l'}$.

Otherwise the total running time of jobs from $\mathcal{J}^{(l,l')}$ is at least $2(l'+1)2^l 2^I \tau$. Hence if we normalize the running times of scheduled jobs and $W = W_{l,l'}$ by dividing them by $2^I\tau$, set $r = r_{l,l'}$, $\lambda = 2(l'+1)2^l$, and $\mathbf{k} = k_{l,l'}$, we get exactly the situation described in the assumptions of Lemma 11.12. The statement follows from Lemma 11.12 by definition of $w_{l,l'} = r_{l,l'}\lambda 2^I\tau / k_{l,l'}$.

Now, we sum over all l and l',

$$\begin{aligned}\sum_{l\geq 3}\sum_{l'\geq 0} \alpha^{(l'+1)2^l/8} &\leq \sum_{l\geq 3} \frac{\alpha^{2^l/8}}{1-\alpha^{2^l/8}} \leq \frac{\alpha}{1-\alpha} + \sum_{i\geq 1}\frac{\alpha^{2^i}}{1-\alpha^{2^i}} \\ &\leq \frac{\alpha}{1-\alpha} + \frac{\sum_{i\geq 1}\alpha^{2i}}{1-\alpha^2} \leq \frac{\alpha}{1-\alpha} + \frac{\alpha^2}{(1-\alpha^2)^2} \leq \frac{1}{12}.\end{aligned}$$

The statement for w_l is a trivial consequence. □

Now we want to prove that the algorithm is efficient, and if the estimates are good, the step (7) schedules all the jobs. The step (7) is the only one that could be inefficient, as the length of all the other steps is bounded by a constant multiple of the current estimate of the longest running time. Since the mesh is divided proportionally to the estimated work, we expect that all the classes are finished at about the same time. The time bound in step (7) is chosen so that if for no class the estimate of work is too small, all jobs are finished. If the estimate is not too large, then the average efficiency in that class is at least 1/4; otherwise the condition in step (7) interrupts immediately. From this it follows that the average efficiency is sufficiently large even if we average over all job classes.

However, since different classes can finish at different times (within a factor of 4), the exact statement is somewhat tedious. One technical issue is that our estimates include the jobs that have already been scheduled during the sampling step. However, since the length of the sampling step is bounded by a multiple of the longest running time, we can "credit" this work to step (7). We solve this formally as in Section 11.2. Instead of literally proving that the time when the efficiency is low is short, we again prove that the length of the schedule is bounded by a constant multiple of the work divided by the number of processors, plus an additive term bounded by a constant multiple

of the longest running time. The next claim analyzes one pass through the steps (4) to (8) in this way and also proves that if the estimates from the sampling are good, the step (7) schedules all jobs or ends by finding a job longer than $2^l\tau$.

Claim 11.14 *(i) The total time T of a pass through steps (4) to (8) is bounded by $T \leq (4 + \frac{4}{47})W'/n^2 + 4 \cdot 2^l\tau$.*

(ii) If $w_l/4 \leq W_l \leq w_l$ for all l, then the pass through steps (4) to (8) either ends by invoking the condition in step (2), or schedules all jobs.

Proof. Since the sizes of all jobs are powers of two, the efficiency of each instance of ORDERED is 1 as long as there are jobs available in that class.

(i) Let T' be the length of the step (7). We prove that for every l,

$$W'_l \geq \frac{1}{4}p_l(T' - 2^l\tau). \tag{1}$$

If $W'_l \geq w_l/4$, then $W'_l \geq \frac{1}{4}p_lT'$, since $T' \leq w_l/p_l$ by the definition of p_l and the condition that bounds the time in step (7). The condition (1) follows.

If $W'_l < w_l/4$ then either not all jobs of $\mathcal{J}^{(l)}$ are finished at the end of step (7), or the step ended because the last job of $\mathcal{J}^{(l)}$ just finished and $W_l < w_l/4$. In both cases at time $2^l\tau$ before the end of the step there were unscheduled jobs in $\mathcal{J}^{(l)}$; otherwise the some job is running for $2^l\tau$ and the step is interrupted by the condition in step (2). Therefore for time at least $T' - 2^l\tau$ the efficiency of the corresponding instance of ORDERED is 1 and the work done is at least $p_l(T' - 2^l\tau)$, which proves the condition (1).

The condition (1) implies that $W' = \sum_l W'_l \geq \frac{1}{4}(\sum_l p_l)(T' - 2^l\tau) \geq \frac{47}{4 \cdot 48}n^2(T' - 2^l\tau)$, and therefore $T' \leq (4 + \frac{4}{47})W'/n^2 + 2^l\tau$. The bound on T now follows, since the length of steps (4) and (8) is bounded by $3 \cdot 2^l\tau$ and no time is spent in steps (5) and (6).

(ii) Suppose for a contradiction that $W_l \geq w_l/4$ for all l, the step (7) is not interrupted by the condition in (2), and does not schedule all jobs from some $\mathcal{J}^{(l)}$. In that case the step (7) takes time $\frac{48}{47}w/n^2$. As the area assigned to $\mathcal{J}^{(l)}$ is at least $\frac{47}{48}w_l/w$ and the efficiency is 1, the total work of jobs from $\mathcal{J}^{(l)}$ scheduled during (7) is at least w_l. If $w_l \geq W_l$, then this means that

all jobs from $\mathcal{J}^{(l)}$ have been scheduled (considering that some positive work was also done during (4), to break the equality); a contradiction. □

Now it is easy to prove the main theorem. We only need to take care of the fact that the sampling step can be repeated several times and of the doubling of the estimate on the longest running time. One subtle observation is that if the running time is larger than the current estimate, then all the previous claims still work until we actually see a long job. This is justified by the following mental experiment: replace all the long jobs by jobs whose length is equal to the current estimate; until the moment we see a job running longer than the estimate, the algorithm behaves exactly the same way on both instances, and hence all the bounds must be true as well.

Theorem 11.15 *The algorithm SAMPLE is 28-competitive.*

Proof. First we bound the total length T_i of steps (3) to (8) during which $I = i$. Let W_i'' denote the total work done during that part of schedule. Step (3) takes time at most $2^I\tau$. According to Claim 11.13, if no job longer than $2^I\tau$ is found, the probability that the steps (4) to (8) are repeated is less than 1/6. Therefore the expected number of passes through steps (4) to (8) is 6/5 and hence using Claim 11.14 we get

$$\begin{aligned} T_i &\leq 2^i\tau + \left(4 + \frac{4}{47}\right) W_i''/n^2 + \frac{6}{5} 4 \cdot 2^i\tau \\ &= \left(4 + \frac{4}{47}\right) W_i''/n^2 + \left(6 - \frac{1}{5}\right) 2^i\tau. \end{aligned}$$

Let W'' be the total work of all jobs and let i' be the maximal value of I during the algorithm. Obviously $2^{i'}\tau \leq 2T_{\max}$. To bound the length of step (1), we use the fact that during step (1) the efficiency is less than 1 for at most 3τ. Hence the total length of the schedule $T(\mathcal{J})$ is bounded by

$$\begin{aligned} \mathrm{E}[T(\mathcal{J})] &\leq \left(4 + \frac{4}{47}\right) W''/n^2 + 3\tau + \left(6 - \frac{1}{5}\right) \sum_{i=1}^{i'} 2^i\tau \\ &\leq \left(4 + \frac{4}{47}\right) T_{\mathrm{eff}}(\mathcal{J}) + \left(6 - \frac{1}{5}\right) \sum_{i=0}^{i'} 2^i\tau \end{aligned}$$

$$\leq \left(4 + \frac{4}{47}\right) T_{\text{eff}}(\mathcal{J}) + \left(6 - \frac{1}{5}\right) 2^{i'+1}\tau$$
$$\leq \left(4 + \frac{4}{47}\right) T_{\text{eff}}(\mathcal{J}) + 4\left(6 - \frac{1}{5}\right) T_{\max}(\mathcal{J}) \leq 28 T_{\text{opt}}(\mathcal{J}).$$

□

12 Higher-dimensional meshes

In this section we generalize our results from Section 11 to higher-dimensional meshes. Obviously, the lower bound from Section 11.3 is true also for higher dimensions, because we can use the two-dimensional proof modified so that the additional dimensions of all jobs are equal to the dimensions of the machine. Generalization of our algorithms from Sections 11.1, 11.2 and 11.4 requires some work. In all three cases the ideas behind the algorithms do not change, however, to prove that they still work requires some tedious calculations.

The main result is that the competitive ratio of the generalized randomized algorithm SAMPLE is a constant that does not depend even on the dimension d, if the dimensions of all jobs and of the machine are are powers of two, and there are no large jobs (i.e., no dimension of any job is more than half of the corresponding dimension of the mesh). If the sizes are power of two, but large jobs are allowed, the competitive ratio is $O(d)$. For general jobs, the competitive ratio is $O(4^d)$.

The competitive ratio of the generalized deterministic algorithm is $O(\sqrt{\log \log N})$ for a constant dimension d, which is optimal up to a constant factor. However, the competitive ratio depends on d.

12.1 Deterministic scheduling

Our algorithm for deterministic scheduling on d-dimensional meshes is a essentially generalization of the algorithm BALANCED PARALLEL for two-dimensional meshes from Section 11.1. The main problem is that the number

of job classes and in particular the number of classes of large jobs is exponential in d.

We assume without loss of generality that the dimensions of each job $(a_1, \ldots, a_d)$ satisfy $a_1 \geq a_2 \geq \ldots \geq a_d$. For simplicity we suppose first that the dimensions of all jobs and of the machine are powers of two and all the dimensions of the machine are the same, n. We say that two jobs are in the same job class, if all the coordinates except for the last one are identical. The number of different classes is $(\log n)^{d-1}$.

Now it is possible to generalize algorithms from the previous section in a straightforward way. CLASS schedules all jobs in a single class using the one-dimensional algorithm ORDERED disregarding all but the last dimensions. SERIAL schedules all classes sequentially. PARALLEL and BALANCED PARALLEL divide the mesh into submeshes with the last coordinate smaller, and then proceed similarly to the two-dimensional case.

The d-dimensional algorithm runs in three phases. In the first phase a d-dimensional version of BALANCED PARALLEL is used to schedule jobs whose last dimension is small. The second phase repeatedly uses a d-dimensional version of PARALLEL to schedule jobs whose last dimensions get larger and larger. Finally, the third phase uses a d-dimensional version of SERIAL, when PARALLEL stops to make progress.

We use BALANCED PARALLEL with $k = \sqrt{\log \log n}$ for all jobs with the last dimension at most $a_1 \leq n/(k \cdot \text{order}(\mathcal{J})) \leq n/(\log n)^d$. This achieves competitive ratio of $\log(\text{order}(\mathcal{J}))/k = d\sqrt{\log \log n}$.

The number of remaining job classes is bounded $(\log((\log n)^d))^d = (d \log \log n)^d$. Hence just applying applying PARALLEL once and then SERIAL leads to a $f(n)^d$ term in the competitive ratio for some non-constant function $f(n)$, which would mean that the algorithm is not optimal even for constant d. To avoid that, we apply PARALLEL recursively. In every step of PARALLEL the number of job classes is reduced by an application of logarithm. This implies that after i iterations of PARALLEL the number of remaining job classes is bounded by $(d(2 \log d + \log^{(i+2)} n))^d$ and after $\log^* n$ iterations it is bounded by $(2d \log d)^d$. After this we finish the schedule using SERIAL for these job classes. This results in a competi-

tive ratio of $O(d\sqrt{\log\log n} + d\log d\log^* n + d\sum_{i>2}\log^{(i)} n + (2d\log d)^d) = O(d\sqrt{\log\log n} + d\log d\log^* n + (2d\log d)^d)$.

If the dimensions are not powers of two, the efficiency decreases by a factor of 2^d. However, efficiency is an additive term in the competitive ratio, and hence it is absorbed in the last term in our case. We have proved the following theorem.

Theorem 12.1 *There exists an on-line algorithm for scheduling on d-dimensional meshes of N processors which is $O(d\sqrt{\log\log N} + d\log d\log^* N + (d\log d)^d)$-competitive. If d is a constant, the algorithm is optimal up to a constant factor.* □

12.2 Off-line scheduling

In this section we bound T_{opt} by a multiple of $\max(T_{eff}, T_{opt})$, which is a generalization of the off-line algorithm for two-dimensional meshes from Section 11.2. At the same time this generalizes a part of the randomized algorithm SAMPLE from Section 11.4, namely the steps (5) and (7) in which we schedule the jobs based on the previous estimates of the work of each job class. This is the main application of this section: as opposed to the two-dimensional case, we do not need this result for the deterministic lower bound.

We can obtain a constant factor independent of d if the dimensions of all jobs and of the machine are powers of two, and there are no large jobs, i.e., each dimension of a job is smaller than the corresponding dimension of the mesh.

If the machine is a cube, i.e., all the dimensions of the mesh are equal, we can process the large jobs with a loss of only a factor d as follows. We partition them according to the number i of dimensions that are same as the corresponding dimension of the mesh. For each i we process them by as $d-i$ dimensional jobs, disregarding the first i dimensions. If the machine is not a cube, the same process loses a factor up to 2^d. If the dimensions of the jobs and of the mesh are not powers of two, the efficiency decreases by a factor of 4^d.

In the rest of this section we demonstrate how to produce a schedule with a length within a constant factor of $\max(T_{\text{eff}}, T_{\max})$, assuming that the machine is a cube, the dimensions of the jobs and of the machine are powers of two and there are no large jobs. We make the assumption that the machine is a cube only for clarity, it is easy to see that it is not necessary.

Two jobs are defined to be in the same job class if they differ only in the last (smallest) dimension. As before, we assign to each class a volume proportional to its work. This time we partition the whole volume, and prove that even after all rounding it can fit into a constant number of copies of the machine. This is sufficient, since instead of scheduling all jobs in parallel, we can arrange the parts of schedule performed on different copies of the machine sequentially. We can do this even in the case of the on-line randomized algorithm, with no change of the proofs.

Now we describe the process of assigning the submeshes and packing them into the desired shape. We derive the estimates on the additional volume later.

Instead of assigning a union of squares to a job class, we assign a union of submeshes whose last dimension is n and the first $d-1$ dimensions are the same as for any job in the class. Rounding the volume up uses at most one such submesh for each subclass; we have to prove that the total volume of them is small.

We continue inductively as follows. We group all job classes with the same $d-2$ first coordinates, and pack the submeshes corresponding to them into submeshes with the last two dimensions n and the first $d-2$ dimensions same as the jobs in those classes. Since only one dimension of the submeshes that are being packed varies and it is always a power of two, all but one of the larger submeshes of each size is completely full. We have to prove that the total volume of those submeshes that are not full is small. We continue this process with $d-3$ dimensions, etc., until we get meshes with all dimensions n, which is our goal.

Now we derive the estimates on the total volume. Let

$$F(i,j) = \sum_{l_1 \geq 1} \sum_{l_2 \geq l_1} \cdots \sum_{l_i \geq l_{i-1}} 2^{-l_1} 2^{-l_2} \ldots 2^{-l_{i-1}} 2^{-jl_i}$$

Each subclass contains jobs of size $n2^{-l_1} \times \ldots \times n2^{-l_d}$ for some particular sequence $1 \leq l_1 \leq \ldots \leq l_d$, using the fact that large jobs are not allowed. It is easy to see that the extra volume during the initial rounding is most $n^d F(d-1,1)$, and the extra volume during grouping according to the first i coordinates is at most $n^d F(i,1)$. Hence the total volume of the final meshes is at most $n^d(1+\sum_{i=1}^{d-1} F(i,1))$.

First we prove by induction on i that $F(i,j) \leq e^{2^{2-j}} 2^{1-i-j}$. We use the fact that for any $l' \geq 1$,

$$\sum_{l \geq l'} 2^{-jl} = \frac{1}{1-2^{-j}} 2^{-jl'} \leq (1+2^{1-j})2^{-jl'} \leq e^{2^{1-j}} 2^{-jl'}.$$

Hence

$$\begin{aligned}
F(1,j) &= \sum_{l_1 \geq 1} 2^{-jl_1} \leq e^{2^{1-j}} 2^{-j} \leq e^{2^{2-j}} 2^{-j} \\
F(i,j) &= \sum_{l_1 \geq 1} \sum_{l_2 \geq l_1} \cdots \sum_{l_i \geq l_{i-1}} 2^{-l_1} 2^{-l_2} \ldots 2^{-l_{i-1}} 2^{-jl_i} \\
&\leq e^{2^{1-j}} \sum_{l_1 \geq 1} \sum_{l_2 \geq l_1} \cdots \sum_{l_{i-1} \geq l_{i-2}} 2^{-l_1} 2^{-l_2} \ldots 2^{-l_{i-1}} 2^{-jl_{i-1}} \\
&= e^{2^{1-j}} \sum_{l_1 \geq 1} \sum_{l_2 \geq l_1} \cdots \sum_{l_{i-1} \geq l_{i-2}} 2^{-l_1} 2^{-l_2} \ldots 2^{-(j+1)l_{i-1}} \\
&= e^{2^{1-j}} F(i-1, j+1) \\
&\leq e^{2^{1-j}} e^{2^{1-j}} 2^{1-i-j} = e^{2^{2-j}} 2^{1-i-j}
\end{aligned}$$

Now $\sum_{i=1}^{d-1} F(i,1) \leq \sum_{i=1}^{d-1} e^2 2^{-i} \leq e^2$ and hence we need at most $1+e^2$ copies of the machine. This finishes the proof.

12.3 Randomized scheduling

In this section we generalize the randomized algorithm SAMPLE to higher dimensions.

We obtain a constant competitive ratio independent of d if the dimensions of all jobs and of the machine are powers of two, and there are no large jobs,

i.e., each dimension of a job is smaller than the corresponding dimension of the mesh. Even with these restrictions, this result seems to be surprisingly good, especially when compared to the deterministic algorithm, where the dependence on d is $O((2d \log d)^d)$

The influence of these restrictions is analogous to off-line scheduling in Section 12.2. If the machine is a cube, we can process the large jobs with a loss of only a factor d as follows. We partition them according to the number i of dimensions that are same as the corresponding dimension of the mesh. For each i we process them as $d-i$ dimensional jobs, disregarding the first i dimensions. If the machine is not a cube, the same process loses a factor up to 2^d. If the dimensions of the jobs and of the mesh are not powers of two, the competitive ratio decreases by a factor of 4^d.

In the rest of this section we demonstrate how to modify the algorithm SAMPLE so that it achieves a constant competitive ratio independent of d assuming that the machine is a cube, the dimensions of the jobs and of the machine are powers of two and there are no large jobs. We make the assumption that the machine is a cube only for clarity, it is not necessary.

We already know from Section 12.2 how to schedule the jobs if we know the proportion of the work in each job class. The fact that we have only estimates does not change anything in the algorithm and the proof. This allows us to generalize the steps (5) to (7). Hence it only remains to generalize the sampling step (4), in particular the partition of the mesh and the derivation of the bounds on the probability of a wrong estimate.

Two jobs are defined to be in the same class if they differ only in the last dimension and in the same subclass if all their dimensions are equal. For $1 \leq l_1 \leq \ldots \leq l_d$, the subclass $\mathcal{J}^{(l_1,\ldots,l_d)}$ contains all jobs of size $m_{l_1} \times \ldots \times m_{l_d}$, where $m_l = n/2^l$. Let $l = l_1 + l_2 + \cdots + l_{d-1}$, $l'_1 = l_1$ and $l'_i = 1 + l_i - l_{i-1}$ for $i > 1$. Note that $l'_i \geq 1$ for all i, and $l \geq d-1$ since we do not allow large jobs. To a subclass $\mathcal{J}^{(l_1,\ldots,l_d)}$ we assign a submesh of size $m_{l'_1} \times \cdots \times m_{l'_{d-1}} \times l'_d m_{l'_d}$. The submesh assigned to $\mathcal{J}^{(l_1,\ldots,l_d)}$ is divided into $l'_d 2^{l+1-d}$ of submeshes of size $m_{l_1} \times \cdots \times m_{l_d}$, which means that we schedule $l'_d 2^{l+1-d}$ jobs from $\mathcal{J}^{(l_1,\ldots,l_d)}$ in parallel.

It is easy to verify by induction that for every i and $l_1, \ldots, l_i$, the sub-

meshes for all $l_{i+1}, \ldots, l_d$ fit into two meshes of size $l_1 \times \cdots \times l_i \times n \times \cdots \times n$, and hence all of the submeshes fit on two copies of the machine. We sample first in parallel for all submeshes on one copy of the machine, then in parallel for all submeshes on the other copy of the machine.

If we sample for a sufficient constant multiple of the estimate of the maximal time, the probability that the estimate for $\mathcal{J}^{(l_1,\ldots,l_d)}$ is wrong is bounded by

$$2\alpha^{l'_d 2^{l+1-d}},$$

where α is a small positive constant which we choose later. Similarly to the proof of Claim 11.13, the total error for each class is bounded by

$$2\frac{\alpha^{2^{l+1-d}}}{1-\alpha^{2^{l+1-d}}} \leq 4\alpha^{2^{l+1-d}}.$$

The number of nondecreasing sequences of positive integers of length $d-1$ with sum l is at most 2^{l-d} (in fact, this is the partition number, which is known to be $2^{\Theta(\sqrt{l-d})}$, however, we do not need such a strong bound here), hence the number of different classes with the same l is bounded by 2^{l-d}. Thus the total probability of an error is bounded by

$$4\sum_{l \geq d-1} 2^{l-d}\alpha^{2^{l+1-d}} = 2\sum_{i \geq 1} 2^i \alpha^{2^i} \leq 4\sum_{j \geq 2} \alpha^j \leq \frac{4\alpha^2}{1-\alpha},$$

where in the first inequality we use the fact that a sequence consisting of α^2 repeated twice, α^4 repeated four times..... α^{2^i} repeated 2^i-times..... is bounded by a geometric sequence. For a sufficiently small α, this bound is at most $1/2$, and hence with probability $1/2$ all of the estimates are correct.

The rest of the proof follows as for the two-dimensional meshes.

Part III
Scheduling parallel jobs with dependencies

13 Scheduling on PRAMs with virtualization

The most important result of this section is an optimal $(2+\phi)$-competitive algorithm for scheduling on PRAMs, where $\phi = (\sqrt{5}-1)/2 \approx 0.618$ is the golden ratio. We extend this result and give optimal bounds for the restricted scheduling problem in which each job requests at most λN processors, $0 < \lambda < 1$. In both cases we give matching lower and upper bounds.

This result improves the best known approximation ratio of 3 for the same problem achieved by Wang and Cheng [WC92]. Our algorithm improves their approximation ratio and in addition is on-line, whereas their approximation algorithm uses the information about the running times for scheduling.

First we present our algorithm. The algorithm is greedy; whenever the number of available processors is larger than requested by a job, some job is scheduled. In addition, if the efficiency is lower than some constant α, some job is scheduled even if we have to use virtualization. The constant α is a parameter of the algorithm which we will optimize later. It always satisfies $1/2 < \alpha \leq 1$.

Algorithm PRAM(α)

while not all jobs are finished **do**

if some available job J requests p processors and p processors are available,

then schedule J on the p processors;

else if the efficiency is less than α and some job is available

then schedule a job on all available processors (using virtualization).

Theorem 13.1 *Suppose that each job requests at most λN processors. Then PRAM(α) is σ-competitive for $\sigma = 2 + \frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$ and $\alpha = \frac{1}{2} - \lambda + \frac{1}{2}\sqrt{4\lambda^2+1}$.*

Remark. The above α and σ satisfy $\sigma = 1 + \frac{1}{\alpha}$ and $\alpha^2 + (2\lambda - 1)\alpha - \lambda = 0$ or equivalently $(1-\alpha)(\frac{1}{\alpha} + \frac{1}{\lambda}) = 1$.

Corollary 13.2 *If the number of requested processors is unrestricted ($\lambda = 1$), then PRAM(ϕ) is $(2+\phi)$-competitive, where $\phi = (\sqrt{5}-1)/2$ is the golden ratio.*

Proof of Theorem 13.1. First, observe that at any time when some job is available, the efficiency is at least α, since otherwise another job would be scheduled. Let T be the total time during the schedule when the efficiency is at least α, T' the time when the efficiency is between $1-\alpha$ and α, and $T'' = T_{<(1-\alpha)}$ the remaining time. If some job is scheduled on less than the requested number of processors, then it is scheduled on at least $(1-\alpha)N$ processors. Therefore no job running during T'' is slowed down and any job running during T' is slowed down by at most a factor of $\frac{\lambda}{1-\alpha}$. During T' and T'' there is no job available, hence by Lemma 7.2 we get $\frac{1-\alpha}{\lambda}T' + T'' \leq T_{\max} \leq T_{\text{opt}}$. Because the efficiency of the optimal schedule cannot be greater than one, $\alpha T + (1-\alpha)T' \leq T_{\text{opt}}$. By adding the first inequality and the last one divided by α, we obtain $T + (1-\alpha)(\frac{1}{\alpha} + \frac{1}{\lambda})T' + T'' \leq (1 + \frac{1}{\alpha})T$ Since $(1-\alpha)(\frac{1}{\alpha} + \frac{1}{\lambda}) = 1$, the length of the schedule is bounded by $T + T'' \leq (1 + \frac{1}{\alpha})T_{\text{opt}} = \sigma T_{\text{opt}}$ and the algorithm is σ-competitive. □

Now we prove a matching lower bound.

Theorem 13.3 *Suppose that the largest number of processors requested by a job is λN, where $0 < \lambda \leq 1$, and N is the number of processors of the machine. Then no on-line scheduling algorithm achieves a better competitive ratio than $\sigma = 2 + \frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$ for all N.*

Proof. We first present the proof for fully on-line algorithms; it is divided into two cases, $\lambda = 1$ and $\lambda < 1$. Then we sketch how it can be generalized to all on-line algorithms.

The case of $\lambda = 1$ The strategy of the adversary is to either keep the efficiency of the machine under ϕ, or force the algorithm to schedule a job on a number of processors that is smaller than the requested number by a factor of at least $2 + \phi$.

The job system used by the adversary is illustrated on Figure 6. It has

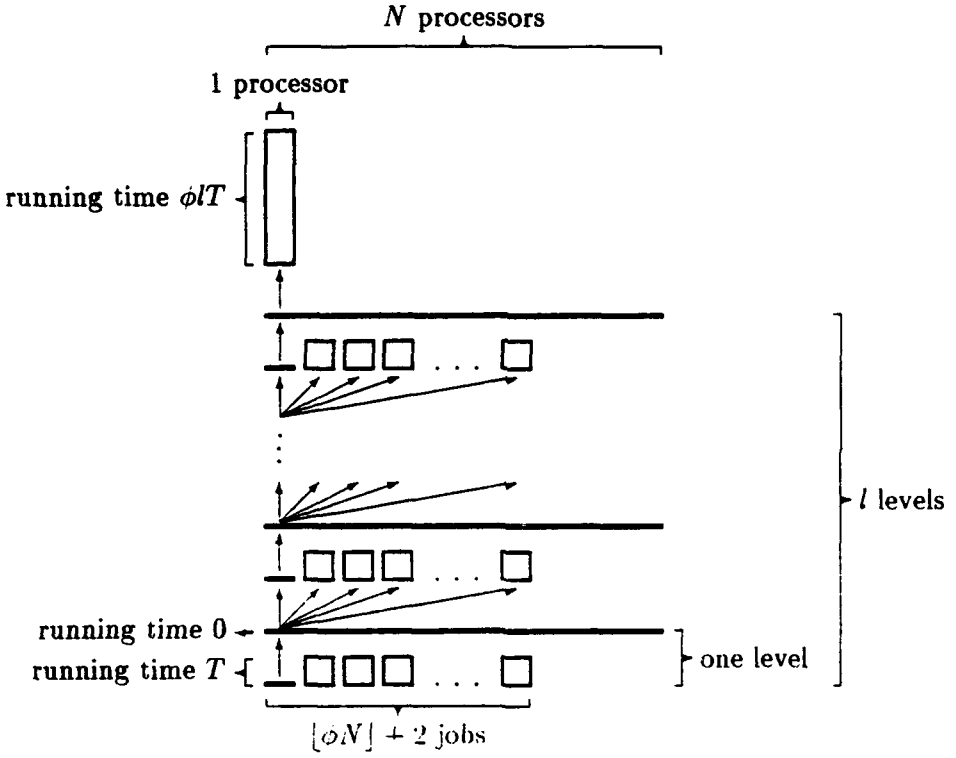


Figure 6: *The job system used in the proof of the lower bound for on-line scheduling on PRAMs with virtualization. (The boxes denote the jobs, the vertical dimension is their running time and the horizontal dimension is their size.)*

l levels; each level has $\lfloor \phi N \rfloor + 2$ sequential jobs and one parallel job of size N. The parallel job depends on one of the sequential jobs from the same level; all sequential jobs depend on the parallel job from the previous level. In addition there is one more sequential job dependent on the last parallel

job. (See Figure 6.)

Now we describe the adversary's strategy for assigning the running times. The adversary can enforce that at each level the parallel job always depends on the sequential job that was scheduled last. This is possible since by our assumption the algorithm is fully on-line and hence cannot distinguish the sequential jobs from each other.

If a parallel job is scheduled on fewer than $(1-\phi)N$ processors, it is slowed down by a factor greater than $1/(1-\phi)$. In this case the adversary assigns a sufficiently long time to this job and removes all other jobs. Therefore the competitive ratio is at least $1/(1-\phi) = 2+\phi$ and we are done.

Otherwise the adversary assigns the running times of all the parallel jobs and all the sequential jobs on which the parallel jobs depend to 0. The running times of all other sequential jobs are set to T for some fixed T, with the exception of the last sequential job, which has running time $T' = \phi lT$. Because of the dependencies and the assumption that no parallel job is scheduled on fewer than $(1-\phi)N$ processors, no parallel job can be scheduled earlier than time T after the previous parallel job has finished. Therefore the schedule takes at least time $lT + T' = (1+\phi)lT$.

The optimal schedule first schedules all jobs with time 0 and then the sequential job with time T' in parallel with all other sequential jobs. The time needed to schedule all other sequential jobs in parallel on $N-1$ processors is $\lceil l(\lfloor \phi N \rfloor + 1)T/(N-1) \rceil$, which is arbitrarily close to ϕlT for sufficiently large N and l. Therefore the length of the schedule is arbitrarily close to ϕlT and the competitive ratio is arbitrarily close to $(1+\phi)/\phi = 2+\phi$. This finishes the proof for $\lambda = 1$.

The case of $\lambda < 1$ The proof of this case is more complicated. The method used in the previous case does not lead to the optimal result. However, if it is iterated with ϕ replaced in the ith iteration by a carefully chosen α_i, then the upper bound is matched in the limit.

The adversary uses a job system similar to the one in the previous case. Given a sequence $\alpha_1, \alpha_2, \ldots$, the job system has a phase for each α_i. Phase i has l levels, and each level has $\lfloor \alpha_i N \rfloor + 2$ sequential jobs and one parallel

job of size $\lfloor \lambda N \rfloor$. The dependencies are the same as in the previous case, i.e., each parallel job depends on one sequential job from the same level and all sequential jobs depend on the parallel job from the previous level. Also the adversary's strategy is similar. The running times of all parallel jobs and all sequential jobs on which the parallel jobs depend are set to 0; the running times of all other sequential jobs are T, except for the jobs from the last level. This scheme only changes if some parallel job is scheduled on too few processors.

Let $\sigma \geq 2$ be the lower bound on the competitive ratio that we want to achieve. We choose $\alpha_1, \alpha_2, \ldots$ so that if the parallel job of the i-th level is scheduled together with more than $\alpha_i N$ sequential jobs, it is slowed down too much. Let $A_i = \frac{1}{i} \sum_{j=1}^{i} \alpha_j$. Let α_i be such that $\sigma = \frac{\lambda}{1-\alpha_1}$ and $\sigma = \frac{\lambda}{1-\alpha_i} + \frac{1-\lambda}{A_{i-1}}$ for $i > 1$. All α_i and A_i are between 0 and 1. Both sequences are decreasing to the same limit L satisfying $\sigma = \frac{\lambda}{1-L} + \frac{1-\lambda}{L}$.

First suppose that no parallel job is scheduled earlier than time T after the previous parallel job has finished. Then the schedule takes time T for each level and the average efficiency of the first i phases is at most $A_i + \frac{1}{N}$. After sufficiently many phases this is arbitrarily close to $L + \frac{1}{N}$. At that point the adversary stops the process and assigns a nonzero time T' to a single sequential job and running time 0 to all the remaining jobs. The time T' is chosen so that the optimal schedule for all the previously scheduled jobs takes time T' on $N - 1$ processors. This forces the competitive ratio to be arbitrarily close to $1 + \frac{1}{L}$.

Now suppose that some parallel job J of phase i is scheduled early. Then J is scheduled on at most $(1 - \alpha_i)N$ processors. We prove that the adversary can achieve a competitive ratio arbitrarily close to σ. For $i = 1$, the job J is slowed down by a factor of $\frac{\lambda}{1-\alpha_1} = \sigma$, so the adversary just runs J long enough. For $i > 1$, let $\bar{T}$ be the time when J is scheduled and let A be the average efficiency of the schedule before $\bar{T}$. Obviously $A \leq A_{i-1}$. The adversary removes all jobs that have not been scheduled and sets the running time of J to $T' = \frac{A}{1-\lambda}\bar{T}$. The optimal schedule runs first all jobs with running time 0 and then all sequential jobs in parallel with the parallel job with running time T'. The time T' is chosen so that the length of the

optimal schedule is within an arbitrarily small factor of T' for large l and N. The schedule generated by the on-line algorithm takes time $\bar{T} + \frac{\lambda}{1-\alpha_i}T' = (\frac{1-\lambda}{A} + \frac{\lambda}{1-\alpha_i})T' \geq (\frac{1-\lambda}{A_{i-1}} + \frac{\lambda}{1-\alpha_i})T' = \sigma T'$. So the competitive ratio is arbitrarily close to $(\bar{T} + \frac{\lambda}{1-\alpha_i}T')/T' \geq \sigma$.

We have shown that if we chose σ such that $\sigma = 1 + \frac{1}{L}$, no on-line algorithm has competitive ratio smaller than σ. We know that $\sigma = \frac{\lambda}{1-L} + \frac{1-\lambda}{L}$. The substitution of $\sigma = 1 + \frac{1}{L}$ and a short calculation shows that the condition for L is equivalent to the equation for α in Theorem 13.1. Therefore $\sigma = 2 + \frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$ is the solution of the equation.

General on-line algorithms We need to modify the job system to handle the case where the on-line algorithm knows the job system in advance. We generate sufficiently many copies of each job, so that the graph is very symmetric and the scheduling algorithm cannot take advantage of its additional knowledge. More precisely, the new job system is a tree of the same depth and each parallel job has the same fan-in-out as before. There is one parallel job dependent on each sequential job except for the last level. So instead of a constant width tree we have a tree which is exponentially larger. The adversary's strategy is the same except for the following modification. When a sequential job is scheduled and it is not the last job of its level, then it and all jobs in the subtree dependent on it are assigned time 0. Thus both the resulting schedule and the optimal schedule have the same length as in the fully on-line case, and the lower bound holds. □

14 Scheduling on meshes and hypercubes with virtualization

In this section we show that the optimal competitive ratio on one-dimensional meshes is $\Theta(\frac{\log N}{\log \log N})$ if both dependencies and virtualization are allowed. This competitive ratio can be achieved by a deterministic algorithm, and our lower bound holds even for randomized algorithms. This proves that in this case randomization does not help, as opposed to scheduling without

dependencies.

The gap between the constant competitive ratio for PRAMs and the optimal $\Theta(\frac{\log N}{\log\log N})$ competitive ratio for one-dimensional meshes is quite big. This is in sharp contrast to scheduling with no dependencies, where we are able to achieve a constant competitive ratio for one-dimensional meshes, only slightly larger than for PRAMs. This shows that the influence of the network topology is amplified by the presence of dependencies.

For higher-dimensional meshes the algorithm can be generalized and is $O((\frac{\log N}{\log\log N})^d)$-competitive. For hypercubes a similar algorithm is $O(\frac{\log N}{\log\log N})$-competitive. However, for these cases we do not have a matching lower bound only.

14.1 Algorithms

First we present an algorithm for a d-dimensional hypercube. As in Section 9, we say that a d-dimensional subcube is *normal* if the coordinates of all its processors are identical except for possibly the last d coordinates. Let h be the smallest power of two such that $h \log h \geq d$. Note that $h = O(\frac{d}{\log d})$. We partition the jobs into h job classes $\mathcal{J}_i$, $1 \leq i \leq h$. A job is in $\mathcal{J}_i$ if it requests a hypercube whose dimension is between $(d+1-i\log h)$ and $(d-(i-1)\log h)$. The hypercube is partitioned into h normal $(d-\log h)$-dimensional subcubes $M_1, \ldots, M_h$. Jobs from $\mathcal{J}_i$ are only scheduled on M_i.

Algorithm HYPERCUBE

while not all jobs are finished **do**

for all i **do**

if there is a job in $\mathcal{J}_i$ available and a normal $(d-i\log h)$-dimensional subcube in M_i is available

then schedule that job on that subcube;

Theorem 14.1 *The competitive ratio of HYPERCUBE for a d-dimensional hypercube with $N = 2^d$ processors is $O(\frac{d}{\log d}) = O(\frac{\log N}{\log\log N})$.*

Proof. The algorithm assigns a $(d - i \log h)$-dimensional subcube to each job from $\mathcal{J}_i$. First, this implies that if there is any job available in $\mathcal{J}_i$, the efficiency of the subcube M_i is equal to 1, and the overall efficiency is at least $1/h$. Second, no job is slowed down by a factor greater than h and hence from Lemma 7.2 it follows that $T_{<\frac{1}{h}} \leq hT_{\max}$. Hence, by Lemma 7.1, the competitive ratio is $O(h) = O(\frac{d}{\log d}) = O(\frac{\log N}{\log \log N})$. □

A similar algorithm can be used for the d-dimensional mesh of $N = n^d$ processors. Let k be the smallest integer such that $k^k > n$. Note that $k = O(\frac{\log n}{\log \log n})$. The jobs are partitioned into $h = k^d$ job classes $\mathcal{J}_i$, $i = (i_1, \ldots, i_d)$, $1 \leq i_j \leq k$. A job belongs to $\mathcal{J}_i$ if it requests a submesh of size $(a_1, a_2, \ldots, a_d)$ such that $n/k^{i_j} < a_j \leq n/k^{i_j - 1}$. The mesh is partitioned into h submeshes M_i of size $\lfloor n/k \rfloor \times \cdots \times \lfloor n/k \rfloor$. The jobs from $\mathcal{J}_i$ are scheduled on submesh M_i only.

Algorithm MESH

while not all jobs are finished **do**

for all $i = (i_1, \ldots, i_d)$ **do**

if there is a job in $\mathcal{J}_i$ available and a $\lfloor n/k^{i_1} \rfloor \times \cdots \times \lfloor n/k^{i_d} \rfloor$ submesh in M_i is available

then schedule that job on such a submesh with the smallest coordinates;

The proof of following theorem is similar to that of Theorem 14.1 and is omitted.

Theorem 14.2 *MESH is $O((\frac{\log N}{\log \log N})^d)$-competitive for a d-dimensional mesh of $N = n^d$ processors.*

14.2 Lower bound

In this section we prove that not even a randomized algorithm for one-dimensional meshes can achieve a better competitive ratio than $\Theta(\frac{\log N}{\log \log N})$. Hence our deterministic algorithm for one-dimensional meshes from Section 14 is within a constant factor of the optimal competitive ratio and even

randomization cannot improve it. This is in contrast to scheduling without dependencies on two-dimensional meshes studied in Section 11, where we have shown that randomization can help significantly.

Our approach to this lower bound is similar to the method used in Section 11.3 to prove a lower bound for deterministic scheduling on two-dimensional meshes without dependencies. The adversary again tries to block a large fraction of processors by jobs that only use a small fraction of all processors. Here we can prove a larger lower bound, since dependencies give the adversary more control over the size of available jobs.

However, it is considerably more difficult to prove a lower bound for randomized algorithms than for deterministic algorithms. For deterministic algorithms, the adversary can simulate the scheduling algorithm and hence its actions can depend on the actions of the scheduler. In contrast, for randomized algorithms, we have to specify the running times, or at least their distribution, in advance, since the adversary has no access to the random bits of the algorithm. This significantly restricts the adversary as opposed to our proof of the lower bound in Section 11.3, where it was crucial to use the possibility of setting the running times according to the actions of the algorithm.

From a technical point of view it is interesting that the dependencies give to the adversary sufficient power to handle randomization. The lower bound for deterministic scheduling on one-dimensional meshes with dependencies in [FKST93] uses a very similar technique to the bound from Section 11.3 for scheduling without dependencies, yet the first one generalizes to randomized algorithms and the other one does not.

Another additional technical difficulty is that the on-line algorithm can use virtualization. Virtualization caused no problem in the deterministic case, since we could argue that if any job is scheduled on a small number of processors, we just assign it a long running time. This argument no longer works, since we have to commit to the distribution of running times beforehand. Thus arguing about a single job is not sufficient. We have to argue that if the algorithm schedules many jobs using virtualization, with high probability one of them has long running time under the distribution of run-

ning times that we choose. To make this argument work, we have to set the parameters in our proof very carefully.

The key technical tool that makes the lower bound work for randomized algorithms is Lemma 14.3. It says that if the algorithm schedules many jobs at once, and we choose some small fraction of them at random, most likely that fraction will block a relatively large number of processors.

We give the proof only for fully on-line algorithms. It can be generalized to all on-line algorithms by the method used in the proof of Theorem 13.3.

The job system and the off-line schedule

Let D be the smallest integer such that $D^{12D} \geq N$; let $T = D^4$ and $S = T^3$. We assume without loss of generality that $N = D^{12D}$ and N is sufficiently large. Note that $D = \Theta(\frac{\log N}{\log \log N})$ and $N = S^D$.

The job system used in the proof is illustrated on Figure 7. It is a tree of depth D^2, similar to a job system used in the proof of Theorem 13.3. All jobs on level $iD + j$ of the tree for $0 \leq i, j < D$ request S^j processors; there are $TN/D^2S^j + 1$ of them. We assign the running times at random, thus, strictly speaking, we have not a single instance of the problem but a distribution on some subset of instances. For each level the running time is 0 for a single randomly chosen job, and for the other jobs it is T with probability $1/T$ and 1 otherwise. All jobs on a given level depend on the job with running time 0 from the previous level, there are no other dependencies.

We call the jobs with running time 0 *critical.*

For each level, the total number of processors requested by non-critical jobs is TN/D^2. Thus the work of the jobs on each level is at least TN/D^2 and the expected work on each level is less than $2TN/D^2$.

First we bound the length of the optimal schedule. The optimal schedule first schedules the chain of critical jobs; this takes no time as their running time is 0. The remaining jobs are independent of each other, hence we can schedule them in time $O(\max(T_{\text{eff}}, T_{\text{max}}))$ using the results on scheduling without dependencies from Section 10. Obviously $T_{\text{max}} = T$. The total expected work is less than $2TN$ and thus the expected length of the optimal schedule is $O(T)$.

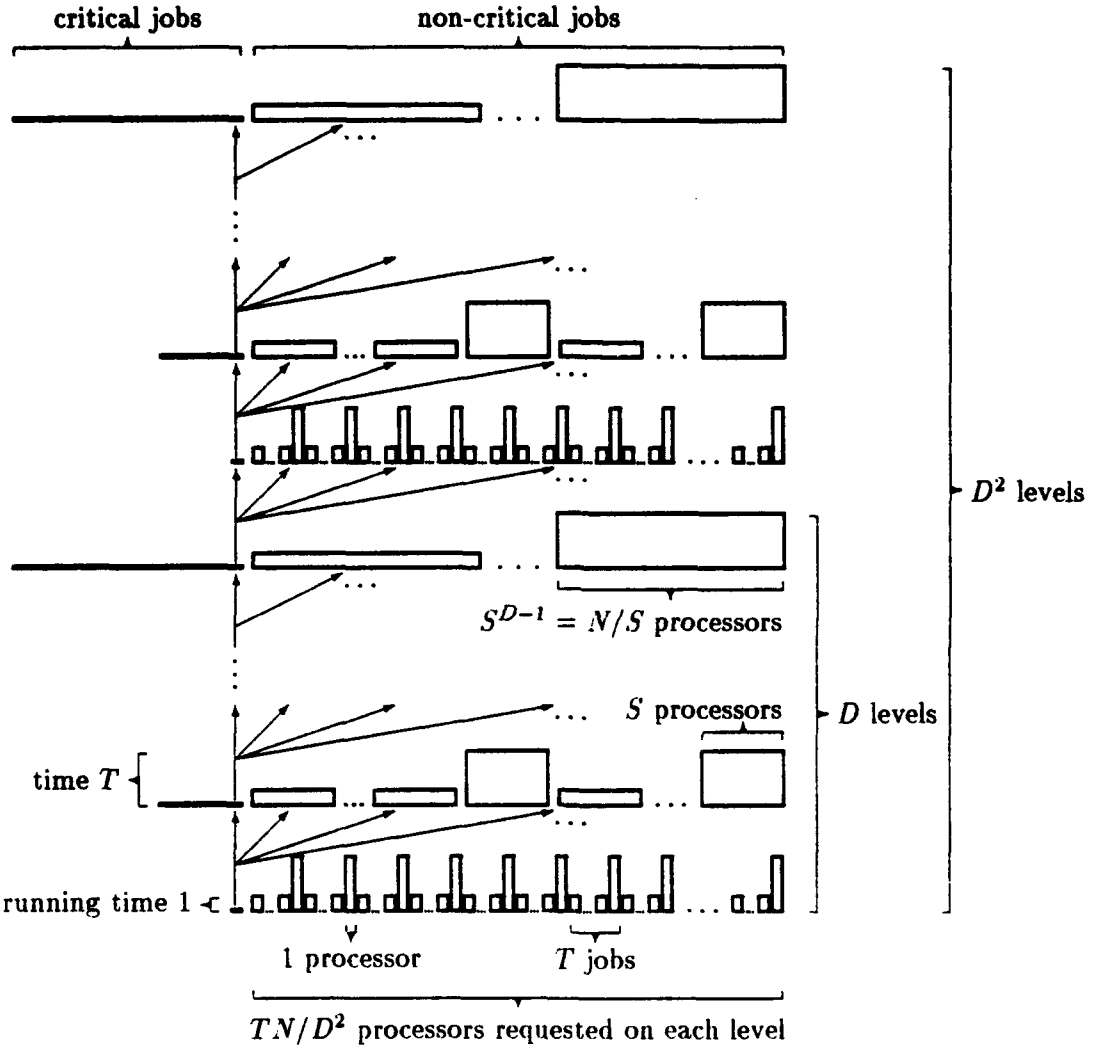


Figure 7: *A typical instance of a job system used in the proof of the lower bound for randomized scheduling on one-dimensional meshes. (The boxes denote the jobs, the vertical dimension is their running time and the horizontal dimension is their size.)*

An overview of the proof

In the rest of the proof we show that the expected length of the schedule generated by any randomized on-line algorithm is at least $\Omega(DT)$, where the expectation is taken over both the random bits of the algorithm and all the instances of the job system as chosen randomly by the adversary. This is sufficient to conclude that the competitive ratio is at least $\Omega(D)$.

It is crucial that the adversary gives the jobs on each level in random order. Since they are indistinguishable to the on-line algorithm, no matter what the algorithm does, they will be scheduled in a random order. In particular the expected fraction of the non-critical jobs from the given level scheduled before the critical one is $1/2$.

The on-line algorithm has to schedule the critical job on the given level before it can schedule any jobs on the next level. If the non-critical jobs scheduled before the critical job on the given level are scheduled on some contiguous segment of the mesh, we expect that there on most segments T times larger than the size of a job there will be at least one job with running time T. If this is true, then such a segment cannot be used for time T for scheduling jobs at least S times longer. Lemma 14.3 proves a similar statement for a general case in which the jobs are not necessarily scheduled on one contiguous segment of the mesh.

Therefore for each D consecutive levels with increasing size of jobs the space cannot be reused efficiently. A constant fraction of levels uses only a small fraction of the machine, namely $O(N/D)$ processors. Since the position of the critical job is random, the expected work of the jobs scheduled before the critical job is $\Omega(TN/D^2)$ on any of these levels. For N sufficiently large the fraction of these jobs that can be scheduled in parallel is negligible. Therefore, if they are scheduled on $O(N/D)$ processors, the expected time until the critical job is scheduled is at least $\Omega(T/D)$. Thus the on-line algorithm needs expected time of $\Omega(DT)$ for all D^2 levels.

These arguments get more complicated if the algorithm uses virtualization. We no longer can completely avoid scheduling jobs in a space which is intuitively unusable, since with virtualization it is always possible schedule a large job on an arbitrarily small segment. However, if this happens too

often, with large probability one of the jobs that use virtualization has long running time. Our parameters are carefully chosen so that this argument is sufficient; in fact, this is the main reason why S is as large as D^{12}, without virtualization a smaller power would be sufficient.

Simplifying assumptions about the on-line algorithm

We first introduce more terminology and make some assumption about the behavior of the algorithm. These assumption are possible, since we can modify any on-line algorithm so that it satisfies them and the competitive ratio is smaller or only slightly larger.

We divide the schedule into phases and subphases as follows. The *subphase* j of the *phase* i, $0 \leq i, j < D$, is the time interval during which the first level with an unscheduled critical job is level $iD + j$ (for convenience we number the levels, phases and subphases from 0).

We use the phrase *just before the current subphase* to refer to the beginning of the subphase if it is the subphase 0 of any phase, or to the end of the previous subphase otherwise.

We assume that during subphase j of phase i only the jobs from level $iD + j$ of the job system are scheduled. This assumption is possible without loss of generality, since once the critical job is scheduled, the on-line algorithm knows that no other jobs depend on the remaining jobs of this level. Thus these remaining jobs from all levels can be scheduled together at the end of the schedule by the constant-competitive algorithm for scheduling without dependencies. This changes the competitive ratio only by an additive constant.

Since the jobs on each level are ordered randomly by the adversary, no matter what the actions of the randomized algorithm are, the jobs from each level are scheduled in a random order. Since we average over all instances of the job system generated by the adversary, it is equivalent to assume that the running times of the jobs are assigned as follows. At the beginning of each subphase we decide randomly the position in which the critical job on that level will be scheduled. Then whenever a job is scheduled, if it has the correct position, it has running time 0. Otherwise we randomly assign its

running time to be T with probability $1/T$ or 1 otherwise. We make the random decision about the running time of a non-critical job only at the moment when that job would finish if its running time were 1 or at the end of the subphase. (Note that the actual running time can be increased by virtualization.) Since the algorithm is on-line, it does not know the running times of jobs until they finish, and thus this delay of the decision does not change its actions. All random choices that we make are independent. Of course, the algorithm can make some additional random choices.

A non-critical job is called *undecided* if we did not yet assign its running time.

We make two assumptions related to virtualization. Both of these assumptions and their use later are very relaxed. We could tighten them by giving precise constants in places where we use asymptotic notation, however the improvement in the final result obtained by that would be very small.

First, we assume that no non-critical job is scheduled on at most $o(1/DT)$ fraction of the number of processors it requests. Otherwise its expected running time is at least $\omega(DT)$. In that case the on-line algorithm could just use the deterministic $O(D)$-competitive algorithm for the remaining jobs. The schedule would finish in time $O(DT)$, and hence the performance of the on-line algorithm would improve.

Second, we assume that it never happens that there are T undecided jobs each running on at most $o(1/D)$ fraction of the number of processors requested by it. Otherwise the probability that none of these jobs will have running time T is $(1 - 1/T)^T \leq 1/e$, since their running times are chosen independently. Hence with at least a constant probability one of these jobs will have running time T, and since it is scheduled on $o(1/D)$ fraction of the requested processors, the length of the schedule is at least $\omega(DT)$. Thus the on-line algorithm again performs better if it uses the deterministic algorithm for the remaining jobs.

The measure of progress

We say that a processor is *used* if it is assigned to a job with running time T scheduled during the current phase. During the subphase j of any phase we

say that a processor is *blocked* if the length of the largest segment of unused processors containing it is at most D^2TS^j (a used processor is considered blocked). The *blocked space* is defined to be the number of blocked processors.

From the definition of the blocked space and our assumptions it follows that in the subphase j of any phase no job of size S^j is scheduled in the space that was blocked just before the current subphase as long as the length of the current phase is at most T. This is trivially true if $j = 0$, since then "just before the current subphase" refers to the beginning of the phase, and no space is blocked at that time. For $j > 0$ "just before the current subphase" refers to the end of the previous subphase. Thus any job scheduled in a space that was blocked at that time would be scheduled between two jobs of running time T that were scheduled during the current phase, and thus are still running. By the definition of blocked space the space between those two jobs is at most $D^2TS^{j-1} = S^j/D^2T = o(S^j/DT)$. Scheduling a job in such a small space would violate our first assumption about virtualization.

Since the algorithm is randomized, we cannot ensure that the adversary always blocks some fixed amount of space, as opposed to the deterministic proof in [FKST93]. Instead, we measure the expected sum of the blocked space and the length of the schedule. This measure is very much like a potential function in amortized analysis. From the overview of the proof it follows that $1/D$ fraction of the space should have approximately the same weight as a time interval of length T/D.

Formally we define the *waste* at a given time to be the sum of the blocked space divided by N/D plus the current length of the schedule divided by $T/100D$. We measure the waste in *units*: one unit of waste corresponds to N/D of the blocked space or to time $T/100D$ in the length of the schedule. The *increase of the waste* is the difference between the current waste and the waste just before the current subphase.

Note that the waste at the beginning of a subphase and at the end of the previous subphase can be different, since blocked space is defined in the context of the current subphase. The waste can decrease only at the beginning of any phase or after time T of any phase.

Estimating the progress

The following lemma says that if many undecided jobs are running at once, the expected increase of the waste is high. Once we know that, the rest of the proof follows easily.

Lemma 14.3 *Suppose that the time since the beginning of the current phase is less than T. If at least $23N/D$ processors are assigned to undecided jobs, then the expected increase of the waste at the end of the current subphase is at least least 2 units.*

Proof. We prove that the expected number of processors that were not blocked just before the current subphase but are blocked after the running times of the undecided jobs are assigned is at least $2N/D$. This is sufficient, since then by the definition the expected increase of waste is at least 2 units.

Divide the processors that were not blocked just before the current subphase into segments of length at most $2DTS^j$ so that no segment is shorter than DTS^j unless it is adjacent to a blocked processor or the end of the mesh on both ends. (This is possible since any segment of at least DTS^j processors can be divided into segments of size between DTS^j and $2DTS^j$.) Mark each of the segments with at least DTS^j processors if at least $1/D$ fraction of its processors is assigned to undecided jobs.

Every marked interval contains at least TS^j processors assigned to undecided jobs, hence it intersects at least T undecided jobs. The probability that all these jobs will be assigned running time 1 is at most $(1 - 1/T)^T \leq 1/e$. Therefore for any two marked intervals with at most $D^2TS^j/2$ processors between them, the probability that the segment between them will be blocked is at least $(1 - 1/e)^2 \geq 1/3$. as the two events are independent. (There is a possibility that one of the undecided jobs is intersected by both marked intervals, in which case the events are not quite independent. However, then both marked intervals intersect T undecided jobs distinct from the common one, and the bound is still true.)

Now we show that there are many marked intervals. It then follows that a constant fraction of them lies between two marked intervals that are

sufficiently close, and hence on average a constant fraction of marked intervals will be blocked.

The segments shorter than DTS^j are all blocked and were not blocked just before the current subphase. Hence if their total length is at least $2N/D$, we are done; otherwise they contain at most $2N/D$ processors assigned to undecided jobs. The unmarked segments with at least DTS^j processors contain a total of at most N/D processors assigned to undecided jobs. Therefore at least $20N/D$ of the processors assigned to the undecided jobs are in the marked segments and the number of the marked segments is at least $(20N/D)/2DTS^j = 10N/D^2TS^j$.

Let the envelope of a marked segment be the largest segment of the mesh containing it which does not intersect any other marked segment and does not contain any processor that was blocked at the end of the previous subphase. Each processor is contained in at most two envelopes, hence the sum of sizes of all envelopes is at most $2N$. It follows that there are at least $6N/D^2TS^j$ marked segments with envelope of size at most $D^2TS^j/2$, since otherwise the sum of the sizes of the envelopes of the remaining more than $4N/D^2TS^j$ marked segments is more than $2N$. Each of these segments with a small envelope is adjacent at both ends to a marked interval, blocked processor or the end of the mesh, hence it will be blocked with probability at least $1/3$. Thus the total expected length of the marked intervals that will be blocked is at least

$$\frac{1}{3}\frac{6N}{D^2TS^j}DTS^j = 2N/D.$$

By the definition of marked segments this area was not blocked at the end of the previous phase. □

Now we are ready to prove the main theorem of this section.

Theorem 14.4 *No randomized on-line scheduling algorithm can achieve a better competitive ratio than* $\Omega(\frac{\log N}{\log\log N})$ *for an one-dimensional mesh of* N *processors.*

First we prove that the expected increase of the waste at the end of each subphase is at least 2 units as long as the length of the current phase is at most T.

If at any time during this subphase the number of processors assigned to undecided jobs is more than $23N/D$, the expected increase of the waste is high by Lemma 14.3. If this is not true at any point of the subphase, we prove that the expected length of the subphase is at least $T/50D$, and hence the increase of the waste is at least 2 units.

From our second assumption about virtualization it follows that at any time at most $23DN/S^j$ undecided jobs are scheduled on any $23N/D$ processors. By the assumption no T undecided jobs are scheduled on $o(1/D)$ fraction of the number of processors they request, hence those $23DN/S^j$ undecided jobs would have to be scheduled on at least $\Omega((23DN/S^j - T)S^j/D) = \Omega(N)$ processors.

The expected number of non-critical jobs scheduled before the critical one is $TN/2D^2S^j$. By the previous argument only $23DN/S^j$ of them can be assigned running time at the end of the subphase. Thus the expected total work done by all jobs on the given level while they are undecided is at least $TN/2D^2 - 23DN \geq (1 - o(1))TN/2D^2$. Since undecided jobs are scheduled on at most $23N/D$ processors, for a sufficiently large N it takes expected time $(1 - o(1))T/46D \geq T/50D$ to perform the required work, which concludes the proof that the expected increase of the waste during the subphase is at least 2 units.

Thus if the length of some phase is at most T, the total expected increase of the waste during all D subphases of that phase is at least $2D$. The total amount of the blocked space is at most N, which corresponds to D units of the waste. Hence at least D units of the expected increase of the waste are contributed by the length of the schedule, and therefore the expected length of the phase is at least $\Omega(T)$. It follows that the expected length of the schedule is $\Omega(DT)$.

This expectation is taken not only over the random bits of the algorithm, but also over the random instances produced by the adversary; therefore we still have to argue that this proves that the competitive ratio is at least $\Omega(D)$. Suppose that the competitive ratio is $o(D)$. Then for each instance the expected length of the on-line schedule is at most $o(D)$ time the length of the optimal schedule. Averaging over all instances we get that the ex-

pected length of the on-line schedule is $o(DT)$, since the average length of the optimal schedule is $O(T)$. This contradiction finishes the proof. □

15 Scheduling without virtualization

In this section we demonstrate the importance of virtualization for on-line scheduling with dependencies. We prove that if a job is allowed to require whole machine, no deterministic on-line algorithm can achieve a better competitive ratio than N, and randomization improves the competitive ratio by at most a factor of two.

We can achieve a matching upper bound in the deterministic case by scheduling one job at a time, without even attempting to schedule the jobs in parallel. This demonstrates that virtualization is essential in the design of competitive scheduling algorithms in our model of on-line scheduling. It also shows yet another difference between on-line scheduling with dependencies and scheduling without dependencies. We have seen in Part II that without dependencies neither the size of the largest job nor virtualization changes the competitive ratios dramatically.

By the same method we prove a lower bound on the competitive ratio of a deterministic on-line algorithm if the number of processors required by a job is restricted to be at most some constant fraction of the machine. We prove a matching upper bound for PRAMs. This again gives us a tradeoff between the size of the largest job and the competitive ratio. Compared to the analogous tradeoff with virtualization allowed in Section 13, here the optimal competitive ratio is significantly higher and it is more dependent on the maximal size of the jobs.

We only give the proof for fully on-line algorithms. It can be generalized to all on-line algorithms by the method used in the proof of Theorem 13.3.

Theorem 15.1 *Assuming that virtualization is not allowed, the following holds.*

(i) No deterministic on-line scheduling algorithm can achieve a smaller competitive ratio than N on any machine with N processors.

(ii) No randomized on-line scheduling algorithm can achieve a smaller competitive ratio than $N/2$ on any machine with N processors.

(iii) Suppose that the largest number of processors requested by a job on a machine with N processors is λN, for a fixed $0 < \lambda < 1$. Then no deterministic on-line scheduling algorithm can achieve a smaller competitive ratio than $1 + \frac{1}{1-\lambda}$.

Proof. We prove the last part of the theorem first, since the other two parts then follow easily.

(iii) The proof is similar to that of Theorem 13.3. The job system used by the adversary has $N - 1$ levels. Each level has $N - \lfloor \lambda N \rfloor + 2$ sequential jobs and one parallel job requiring $\lfloor \lambda N \rfloor$ processors. The parallel job depends on one of the sequential jobs from the same level: all sequential jobs depend on the parallel job from the previous level. In addition there is one more sequential job dependent on the last parallel job.

In the beginning the algorithm can schedule only sequential jobs. The adversary enforces that on each level the sequential job which the parallel job depends on is started last; this is possible since the algorithm cannot distinguish between the sequential jobs. The adversary terminates this sequential job and keeps the other sequential jobs running for some sufficiently large time T. During this time the scheduling algorithm cannot schedule the parallel job. As soon as the parallel job is scheduled, the adversary terminates it and all remaining jobs of this level. This process is repeated until all jobs except the last sequential job have been scheduled. The adversary assigns time $T' = (N - \lfloor \lambda N \rfloor + 1)T$ to the last job. The total length of the generated schedule is $(N-1)T + T' = (2N - \lfloor \lambda N \rfloor)T$.

The off-line algorithm first schedules all jobs of time 0; then schedules the sequential job with running time T' and in parallel with it all the other sequential jobs. There are $(N-1)(N - \lfloor \lambda N \rfloor + 1)$ such jobs, all with running time at most T and independent of each other. The schedule for them on $N - 1$ processors takes time at most $(N - \lfloor \lambda N \rfloor + 1)T = T'$. So the length of the off-line schedule is at most T' and the competitive ratio is at least $\frac{(N-1)T+T'}{T'} = 1 + \frac{N-1}{N-\lfloor \lambda N \rfloor + 1}$. This is arbitrarily close to $1 + \frac{1}{1-\lambda}$ for large N and constant λ.

(i) The proof is essentially a special case of the proof of (iii) for $\lambda = 1$. Each level of the job system has two sequential jobs and one parallel job requiring the whole machine. All sequential jobs with no dependent jobs have running time T. As before, the adversary enforces that these jobs are scheduled sequentially, while the optimal schedule runs all N of them in parallel. This gives the competitive ratio N. (In this case we use a simpler job system which works directly even for the algorithms that are not fully on-line, see [FKST93].)

(ii) We use the same job system as in the proof of (i). As the two sequential jobs are not distinguishable, the adversary can guarantee that with probability at least 1/2 the randomized on-line algorithm schedules the job with nonzero running time first. In that case the parallel job from that cannot be scheduled until this sequential job finishes. Thus the expected length of the schedule produced by the on-line algorithm is at least half of the total running time of jobs with nonzero running time, which gives the competitive ratio of $N/2$. □

Now we show that this lower bound is tight for PRAM. In fact, any greedy strategy achieves this bound.

Algorithm GREEDY

while not all jobs are finished **do**

if some job J requires p processors and p processors are available,

then schedule J on the p processors.

Theorem 15.2 *Suppose that the largest number of processors requested by a job is λN, where $0 < \lambda < 1$. Then GREEDY is $(1 + \frac{1}{1-\lambda})$-competitive.*

Proof. No job requests more than $\lfloor \lambda N \rfloor$ processors. Therefore if the efficiency is less than $1 - \lambda$, there is no available job. By Lemma 7.2 this time is smaller than the total time along some path in the dependency graph and hence $T_{<(1-\lambda)} \leq T_{\max} \leq T_{\text{opt}}$. Lemma 7.1 finishes the proof. □

16 Tree dependency graphs

In this section we prove Theorem 4.1, which says that scheduling job systems whose dependency graphs are trees is as difficult as scheduling general job systems.

First note that a similar theorem is easy to prove if we restrict ourselves to fully on-line algorithms. Suppose that we have a fully on-line algorithm for tree dependency graphs and a general dependency graph $\mathcal{F}$. We dynamically construct a tree subgraph $\mathcal{F}'$ of $\mathcal{F}$ and use the algorithm on $\mathcal{F}'$. We can do this because the fully on-line algorithm does not know the dependencies in advance. For each job J we only keep the edge from a job J' such that J became available when J' finished. The generated schedule is a valid schedule for both $\mathcal{F}$ and $\mathcal{F}'$, and the optimal schedule for $\mathcal{F}'$ can only improve if some dependencies are removed. Therefore the achieved competitive ratio for $\mathcal{F}$ is no greater than the ratio for the tree dependency graph $\mathcal{F}'$.

In the next theorem we prove the same statement for the more difficult case is when the on-line algorithm may know the dependency graph in advance.

Theorem 16.1 *Let an on-line scheduling problem with dependencies be given (i.e., a specific architecture and simulation factors). Then the optimal competitive ratio for this problem is equal to the optimal competitive ratio for a restricted problem in which we allow only job systems whose dependency graphs are trees as inputs.*

Proof. Obviously the optimal competitive ratio for the restricted problem is at most the ratio for the general problem. To prove the reverse implication, assume that we have a σ-competitive algorithm S for the restricted problem. Using it, we show how to schedule a general job system so that the competitive ratio is at most σ.

Given a general dependency graph $\mathcal{F}$, we create a job system with a tree dependency graph $\mathcal{F}'$. Then we use the schedule for $\mathcal{F}'$ produced by S to schedule $\mathcal{F}$. We determine the running times of jobs in $\mathcal{F}'$ dynamically based on the running times of jobs in $\mathcal{F}$.

The set of jobs of $\mathcal{F}'$ is the set of all directed paths in $\mathcal{F}$ starting with any job that has no predecessor. There is a directed edge (p, q) in $\mathcal{F}'$ if p is a prefix of q. If $J \in \mathcal{F}$ is the last node of $p \in \mathcal{F}'$, then p is called a *copy* of J. The resource requirements of each copy of J are the same as those of J. A path p is the *last copy* of J if it is the last copy of J to be scheduled. Let $\mathcal{F}''$ be the subgraph of $\mathcal{F}'$ consisting of all last copies and all dependencies between them.

Our scheduling algorithm works as follows. We run S on $\mathcal{F}'$. Suppose S schedules $p \in \mathcal{F}'$. If p is the last copy of some J, then we schedule J to the same set of processors as p was scheduled by S. If p is not the last copy, then we remove p and all jobs that dependent on it. If a job $J \in \mathcal{F}$ finishes, we stop its last copy $p \in \mathcal{F}'$.

Notice that if p is the last copy of J, then we schedule J at the same time, on the same set of processors and with the same running time as S schedules p. All other copies of J are immediately stopped.

To show correctness of our schedule, we need to prove that when the last copy of J is available to S, J is available to us. Suppose this is not the case. Then there is some $J' \in \mathcal{F}$ such that J depends on J' and J' has not finished yet. Then the last copy of J', say $q \in \mathcal{F}'$, is also not finished, and there is a copy p of J such that q is a prefix of p. So p is a copy of J that is not available yet, a contradiction.

The schedule S generated for $\mathcal{F}'$ and our schedule for $\mathcal{F}$ have the same length. Only the jobs from $\mathcal{F}''$ (i.e., the last copies of the jobs from $\mathcal{F}$) are relevant in $\mathcal{F}'$: all other jobs have running time 0 and can be scheduled at the end. By construction, every schedule for $\mathcal{F}$ corresponds to a schedule for $\mathcal{F}''$ and therefore to a schedule of $\mathcal{F}'$. So the competitive ratio for $\mathcal{F}$ is not larger than the competitive ratio for $\mathcal{F}'$, which is at most σ by the assumption of the theorem. □

Algorithmically, the above reduction from general constraints to trees is not completely satisfactory, because $\mathcal{F}'$ can be exponentially larger than $\mathcal{F}$. Nevertheless, it proves an important property of on-line scheduling from the viewpoint of competitive analysis.

Part IV
Randomized scheduling of sequential jobs

17 The model and the previous results

The problem of on-line scheduling of sequential jobs was introduced in 1966 by Graham [Gra66]. In this model we have m processors and a sequence of jobs arriving one by one; there are no dependencies between the jobs. The jobs are sequential, i.e., they require only one processor. When a job arrives, we know its running time and we have to assign it to one of the processors immediately, without knowledge of the jobs that arrive later. The jobs cannot be preempted.

As in our model for parallel jobs, the goal is to minimize the makespan, the time when the last job is completed. In the randomized model we measure the expected makespan. The performance of an on-line algorithm is again measured by the competitive ratio.

In Section 18 we improve the previously known lower bounds on the competitive ratio of randomized on-line algorithm for this problem. This results was proved independently by Chen, van Vliet and Woeginger [CvVW94a]. Now we survey the previous results.

While the deterministic case has been studied extensively [Gra66, GW93, BFKV92, KPT94], much less is known about the randomized case; all of the following results are by Bartal, Fiat, Karloff, and Vohra [BFKV92]. Only the case of $m = 2$ is solved completely; an optimal 4/3-competitive algorithm is known and provably better than any deterministic algorithm. For $m = 3$ a nontrivial lower bound of 1.4 was proved. For $m > 3$, the best known lower bound was the easy 4/3 bound, not even matching the bound for $m = 3$. For $m \geq 3$ no randomized algorithm with a better competitive ratio than the best deterministic algorithm is known.

For a long time the best deterministic algorithm was the $(2 - \frac{1}{m})$-

competitive Graham's algorithm [Gra66]. For $m = 2$ and $m = 3$, Graham's algorithm is provably optimal. For larger m, it was improved several times during the last few years [GW93, BFKV92, KPT94]. For sufficiently large m the best known algorithm is 1.945-competitive [KPT94]. For $m > 3$, an algorithm slightly better than Graham's is presented in [GW93]. The best lower bound on the competitive ratio of the deterministic scheduling algorithm for large m is approximately 1.837 [BKR]; an earlier bound of $1 + 1/\sqrt{2} \approx 1.707$ is valid for any $m > 3$ [FKT89].

For a related model, deterministic preemptive on-line scheduling of sequential jobs, Chen, van Vliet and Woeginger [CvVW94b] proved the same lower bound as we prove for randomized nonpreemptive algorithms, and gave a matching algorithm.

18 An improved lower bound

We prove that the competitive ratio of any randomized on-line scheduling algorithm for m machines is at least $1 + 1/((\frac{m}{m-1})^m - 1)$. This matches the known tight bound of $4/3$ for $m = 2$ and improves the previous bounds for all $m > 2$. If m is large, this bound approaches $1 + 1/(e-1) \approx 1.582$. Table 8 compares the bounds for a few values.

number of processors	our bound	previous bound
2	$4/3 \approx 1.333$	4/3
3	$27/19 \approx 1.421$	1.4
4	$256/175 \approx 1.463$	4/3
5	$3125/2101 \approx 1.487$	4/3
6	$46656/31031 \approx 1.504$	4/3
∞	$1 + 1/(e-1) \approx 1.582$	4/3

Table 8: *Lower bounds on the competitive ratio for randomized on-line scheduling of sequential jobs.*

The intuition for our lower bound can be better understood if we look at the algorithm and the lower bound for two processors from [BFKV92]. Their algorithm guarantees that the ratio of the expected loads on the two processors is 2 : 1 most of the time (more exactly, unless there is a job with a very long running time), where the load of a processor is defined as the total running time of all jobs assigned to the processor, also called height in [BFKV92]. Their lower bound essentially shows that any optimal algorithm has to maintain this ratio of expected loads, and that it is not possible to maintain a better ratio. The optimal algorithm can balance the loads exactly, hence the competitive ratio is $2/1.5 = 4/3$.

In the case of m processors we show that the best ratio of loads which might be possible to maintain is $1 : x : x^2 : \ldots : x^{m-1}$, where $x = m/(m-1)$. The optimal schedule can balance the loads to be $(1 + x + \cdots + x^{m-1})/m = m(x^m - 1)/(x - 1)$, hence the competitive ratio is at least $x^{m-1}/(m(x^m - 1)/(x - 1))$, which gives our bound. See Figures 8 and 9 for an illustration.

We first prove a general lemma which applies to any sequence of jobs. For our proof the last m jobs of the sequence are most important. In fact in the particular sequence that we use later, the only purpose of the other jobs is to pad the sequence so that the optimal schedule can always balance the loads exactly.

Let a sequence of jobs $\mathcal{J}$ be given. Denote the last m jobs of $\mathcal{J}$ by $J_1, \ldots, J_m$ and their running times by $t_1, \ldots, t_m$. Let $\mathcal{J}_i$ be an initial segment of $\mathcal{J}$ ending by J_i (i.e., $\mathcal{J}_m = \mathcal{J}$). Let S_i be the sum of the running times of all jobs in $\mathcal{J}_i$ and let $T_{\mathrm{opt}}(\mathcal{J}_i)$ be the length of an optimal schedule for $\mathcal{J}_i$. For a given randomized algorithm A, $\mathrm{E}[T_A(\mathcal{J}_i)]$ denotes the expected length of the schedule it generates on $\mathcal{J}_i$.

The definition of the competitive ratio σ implies that for any choice of the sequences of jobs $\mathcal{J}_i$ we have

$$\frac{\sum_{i=1}^{m} \mathrm{E}[T_A(\mathcal{J}_i)]}{\sum_{i=1}^{m} T_{\mathrm{opt}}(\mathcal{J}_i)} \leq \frac{\sum_{i=1}^{m} \sigma T_{\mathrm{opt}}(\mathcal{J}_i)}{\sum_{i=1}^{m} T_{\mathrm{opt}}(\mathcal{J}_i)} = \sigma.$$

The following lemma gives an upper bound on the denominator of the left-hand side.

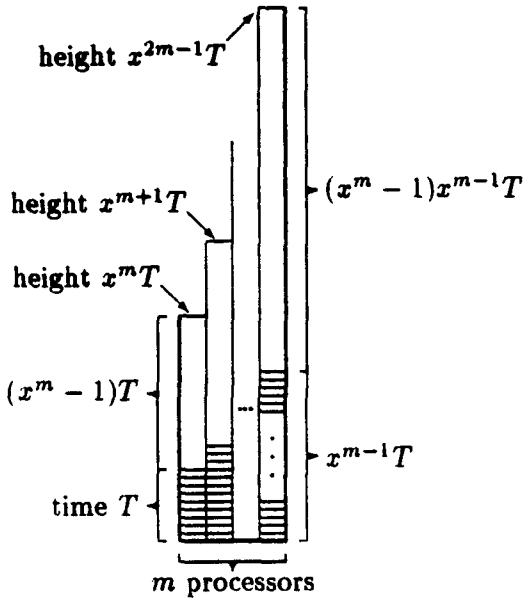


Figure 8: *An optimal on-line schedule for the sequence of jobs used in the lower bound for randomized on-line scheduling of sequential jobs.*

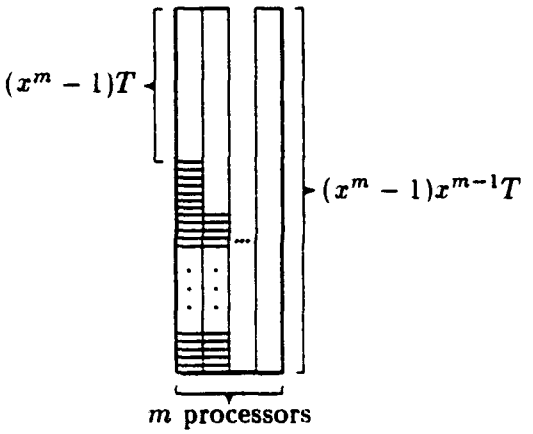


Figure 9: *An optimal off-line schedule for the sequence of jobs used in the lower bound for randomized on-line scheduling of sequential jobs.*

Lemma 18.1 *For any randomized on-line scheduling algorithm A for m machines, $\sum_{i=1}^{m} \mathrm{E}[T_A(\mathcal{J}_i)] \geq S_m$.*

Proof. Fix a sequence of random bits used by the algorithm A. Let T_i be the makespan of the schedule generated by the algorithm A for the jobs in $\mathcal{J}_i$ with the fixed random bits. Since the algorithm is on-line, the schedule for $\mathcal{J}_i$ is obtained from the schedule for $\mathcal{J}_{i-1}$ by adding J_i to one of the processors.

Order the processors so that the load of ith processor does not change after J_i is scheduled. There always exists such an ordering: Designate a processor to be the ith one, if J_i is the last job scheduled on it. Clearly at most one processor is designated as the ith one. If no processor is designated as the jth one for some j, pick one of the remaining processors arbitrarily; note that no job J_i is scheduled on these processors. This defines an ordering satisfying the condition.

Let L_i be the load of ith processor after scheduling all jobs. Obviously $\sum_{i=1}^{m} L_i = S_m$. The load of ith processor is L_i already after scheduling $\mathcal{J}_i$ because of our condition on the order of the processors. Therefore $T_i \geq L_i$ and $\sum_{i=1}^{m} T_i \geq \sum_{i=1}^{m} L_i = S_m$ for any choice of the random bits.

The lemma follows by the linearity of expectation. □

Let $x = m/(m-1)$. We choose our sequence of jobs so that the following property is satisfied. If all the jobs preceding J_i are scheduled so that the ratio of loads is $1 : x : \ldots : x^{m-1}$, then after scheduling J_i on the least loaded processor the ratio of the loads is preserved.

Let $T = (m-1)^{2m-1}$. The sequence $\mathcal{J}$ consists of $(1 + \cdots + x^{m-1})T$ jobs of running time 1 followed by m jobs with running time $t_i = (x^m - 1)x^{i-1}T$. The value of T is chosen so that all running times are integers. Optimal on-line and off-line schedules are illustrated on Figures 8 and 9.

After scheduling the jobs preceding J_i so that the ratio is as described above, the loads of the processors are $x^{i-1}T, \ldots, x^{i+m-2}T$. For $i = 1$ this is obvious. It is maintained inductively, since by our choice of running times we have $x^{i-1}T + t_i = x^{i+m-1}T$, which is exactly the condition we need to preserve the ratio of the loads. The next theorem says that no on-line algorithm can do better, even if it is randomized.

Theorem 18.2 *For any randomized on-line scheduling algorithm for m machines, the competitive ratio σ_m is at least $1 + 1/((\frac{m}{m-1})^m - 1)$.*

Proof. First we prove that $T_{\text{opt}}(\mathcal{J}_i) = t_i$. The total running time of all jobs in $\mathcal{J}_i$ is

$$\begin{aligned} S_i &= (1 + \cdots + x^{m-1})T + (x^m - 1)(1 + \cdots + x^{i-1})T \\ &= (x^i + \cdots + x^{m-1})T + x^m(1 + \cdots + x^{i-1})T \\ &= x^i(1 + \cdots + x^{m-1})T \\ &= x^i \frac{x^m - 1}{x - 1} T = \frac{x}{x-1} t_i = m t_i. \end{aligned}$$

Since running times of all jobs are integers, at most m of them are greater than 1 and t_i is the maximal running time, the loads of the m processors can be balanced perfectly and $T_{\text{opt}}(\mathcal{J}_i) = t_i$.

Using Lemma 18.1 and $S_m = x^m(1 + \cdots + x^{m-1})T$ from the previous computation, the competitive ratio σ_m for any randomized on-line algorithm A is bounded by

$$\begin{aligned} \sigma_m &\geq \frac{\sum_{i=1}^m \mathrm{E}[T_A(\mathcal{J}_i)]}{\sum_{i=1}^m T_{\text{opt}}(\mathcal{J}_i)} \geq \frac{S_m}{\sum_{i=1}^m t_i} = \\ &= \frac{x^m(1 + \cdots + x^{m-1})T}{(1 + \cdots + x^{m-1})(x^m - 1)T} = \frac{x^m}{x^m - 1} = 1 + \frac{1}{(\frac{m}{m-1})^m - 1}. \end{aligned}$$

□

We conjecture that there exists a randomized algorithm which matches this bound, but we are unable to prove this at the present time.

References

[ASE92] Noga Alon, Joel H. Spencer, and Paul Erdös. *The Probabilistic Method.* Wiley, 1992.

[BCH+88] Sandeep N. Bhatt, Fan R. K. Chung, Jia-Wei Hong, F. Thomson Leighton, and Arnold L. Rosenberg. Optimal simulations by butterfly networks. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 192–204. ACM, 1988.

[BDBK+90] Shai Ben-David, Allan Borodin, Richard M. Karp, Gabor Tardos, and Avi Wigderson. On the power of randomization in online algorithms. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 379–386. ACM, 1990.

[BDW86] Jacek Blażewicz, Mieczysław Drabowski, and Jan Węglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transactions on Computers*, c-35(5):389–393, 1986.

[BFKV92] Yair Bartal, Amos Fiat, Howard Karloff, and Rakesh Vohra. New algorithms for an ancient scheduling problem. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 51–58. ACM, 1992. To appear in Journal of Computer and System Sciences.

[BKR] Yair Bartal, Howard Karloff, and Yuval Rabani. A new lower bound for m-machine scheduling. To appear in Information Processing Letters.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT-Press, Cambridge MA, 1990.

[CL88] Guan-Ing Chen and Ten-Hwang Lai. Scheduling independent jobs on hypercubes. In *Proceedings of the 5th Annual Symposium on Theoretical Aspects of Computer Science*, pages 273–280, 1988.

[CvVW94a] Bo Chen, André van Vliet, and Gerhard J. Woeginger. A lower bound for randomized on-line scheduling algorithms. Technical Report 9414/A, Econometric Institute, Erasmus University, Rotterdam, the Netherlands, 1994.

[CvVW94b] Bo Chen, André van Vliet, and Gerhard J. Woeginger. An optimal algorithm for preemptive on-line scheduling. Technical Report 9404/A, Econometric Institute, Erasmus University, Rotterdam, the Netherlands, 1994.

[DL81] Jianzhong Du and Joseph Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal of Discrete Mathematics*, 2(4):473–487, 1981.

[FKST92] Anja Feldmann, Ming-Yang Kao, Jiří Sgall, and Shang-Hua Teng. Optimal online scheduling of parallel jobs with dependencies. Technical Report CMU-CS-92-189, Carnegie Mellon University, 1992.

[FKST93] Anja Feldmann, Ming-Yang Kao, Jiří Sgall, and Shang-Hua Teng. Optimal online scheduling of parallel jobs with dependencies. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 642–651. ACM, 1993.

[FKT89] U. Faigle, W. Kern, and Gy. Turan. On the performance of on-line algorithms for partition problems. *Acta Cybernetica*, 9:107–119, 1989.

[FST91] Anja Feldmann, Jiří Sgall, and Shang-Hua Teng. Dynamic scheduling on parallel machines. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 111–120. IEEE, 1991.

[FST94] Anja Feldmann, Jiří Sgall, and Shang-Hua Teng. Dynamic scheduling on parallel machines. *Theoretical Computer Science*, 1994. To appear in the special issue on dynamic and online algorithms.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness.* Freeman, 1979.

[Gra66] R. L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, pages 1563–1581, November 1966.

[GW93] Gábor Galambos and Gerhard J. Woeginger. An on-line scheduling heuristic with better worst case ratio than Graham's list scheduling. *SIAM Journal on Computing*, 22(2):349–355, 1993.

[HB84] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[Hoe63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *American Statistical Association Journal*, pages 13–29, March 1963.

[HR90] Torben Hagerup and Christine Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33:305–308, 1990.

[KA86] S. Rao Kosaraju and Mikhail J. Atallah. Optimal simulation between mesh-connected arrays of processors. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 264–272. ACM, 1986.

[KPT94] David R. Karger, Steven J. Phillips, and Eric Torng. A better algorithm for an ancient scheduling problem. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 132–140. ACM-SIAM, 1994.

[LRK78] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26:22–35, 1978.

[LST90] Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithm for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[MPT93] Rajeev Motwani, Steven Phillips, and Eric Torng. Nonclairvoyant scheduling. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 422–431. ACM-SIAM, 1993.

[Ran87] Abhiram G. Ranade. How to emulate shared memory. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 185–194. IEEE, 1987.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.

[Sle80] Daniel D. Sleator. A 2.5 times optimal algorithm for packing in two dimensions. *Information Processing Letters*, 10(1):37–40, 1980.

[SOG+94] Jaspal Subhlok, David R. O'Hallaron, Thomas Gross, Peter A. Dinda, and Jon Webb. Communication and memory requirements as the basis for mapping task adn data parallel programs. Technical Report CMU-CS-94-106, Carnegie Mellon University, 1994.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[ST93] David B. Shmoys and Éva Tardos. Scheduling unrelated machines with costs. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 448–454. ACM-SIAM, 1993.

[Ste93] A. Steinberg. A strip packing algorithm with absolute performance bound 2. Manuscript, 1993.

[Sub93] Jaspal Subhlok. Automatic mapping of task and data parallel programs for efficient execution on multicomputers. Technical Report CMU-CS-93-212, Carnegie Mellon University, 1993.

[SWW91] David B. Shmoys, Joel Wein, and David P. Williamson. Scheduling parallel machines on-line. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 131–140. IEEE, 1991.

[TSWY94] John Turek, Uwe Schwiegelshohn, Joel L. Wolf, and Philip S. Yu. Scheduling parallel tasks to minimize average response time. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 112–120. ACM-SIAM, 1994.

[TWY92] John Turek, Joel L. Wolf, and Philip S. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332. ACM, 1992.

[WC92] Qingzhou Wang and Kam Hoi Cheng. A heuristic of scheduling parallel tasks and its analysis. *SIAM Journal on Computing*, 21(2):281–294, 1992.