Optimal On-line Scheduling of Parallel Jobs with Dependencies

Anja Feldmann* Ming-Yang Kao† Jiří Sgall‡ Shang-Hua Teng§

Abstract

We study the following general on-line scheduling problem. Parallel jobs arrive on a parallel machine dynamically according to the dependencies between them. Each job requests a certain number of processors in a specific communication configuration, but its running time is not known until it is completed. We present optimal on-line algorithms for PRAMs and one-dimensional meshes, and efficient algorithms for hypercubes and general meshes. For PRAMs we obtain optimal tradeoffs between the competitive ratio and the largest number of processors requested by any job.

Our results demonstrate that on-line scheduling with dependencies differs from scheduling without dependencies in several crucial aspects. First, it is essential to use virtualization, i.e., to schedule parallel jobs on fewer processors than requested. Second, the maximal number of processors requested by a job has significant influence on the performance. Third, the geometric structure of the network topology is an even more important factor than in the absence of dependencies.

1 Introduction

This paper investigates on-line scheduling problems for parallel machines. Parallel jobs arrive dynamically according to the dependencies between them (precedence constraints); a job arrives only when all jobs it depends on are completed. Each job requests a part of a parallel machine with a certain number of processors and a specific communication configuration. The running time of a job can be determined only by actually running the job.

*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

†Department of Computer Science, Duke University, Durham, NC 27706. Supported in part by NSF Grant CCR-9101385.

‡E-mail sgallj@mbox.cesnet.cz. Mathematical Institute, AV ČR, Žitná 25, 115 67 Praha 1, Czech Republic; partially supported by grants A119107 and A1019602 of AV ČR. This work was done at School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, U.S.A.

§Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801 steng@cs.uiuc.edu and Department of Computer Science, University of Minnesota, Minneapolis, MN 55455 steng@cs.umn.edu. Supported in part by an NSF CAREER award (CCR-9502540) and an Alfred P. Sloan Research Fellowship. Most part of this work was done while the author was at Xerox Palo Alto Research Center, Palo Alto, CA and at the Department of Mathematics and the Lab. for Computer Science, MIT, Cambridge, MA.

A *scheduling algorithm* computes a schedule that satisfies the resource requirements and respects the dependencies between the parallel jobs. The performance of an on-line scheduling algorithm is measured by the *competitive ratio* [22]: the worst-case ratio of the total time (the makespan) of a schedule it computes to that of an optimal schedule, computed by an off-line algorithm that may know all the jobs and their dependencies, resource requirements and running times in advance.

A schedule is required to be *nonpreemptive* and *without restarts*, i.e., once a job is scheduled, it is run on the same set of processors until completion. We consider this a highly desirable approach to avoid high overheads incurred by resetting a machine and its interconnection networks, reloading programs and data, etc.

An important fact is that a parallel job can be scheduled on fewer processors than it requests using a technique called *virtualization*. The job is executed by a smaller set of processors, each of them simulating several processors requested by the job. This yields good results if the mapping of the requested processors onto the smaller set preserves the network topology, which is a property shared by many commonly used architectures such as PRAMs (shared memory parallel machines), meshes, and hypercubes. The work of a job is preserved and the running time increases proportionally. This case is also called *ideally malleable jobs*, as opposed to non-ideally malleable jobs, where the work is not preserved (i.e., the running time is not exactly proportional to the number of processors); we do not consider the case of non-ideally malleable jobs.

1.1 Our results

We present optimal on-line scheduling algorithms for PRAMs and one-dimensional meshes, and efficient algorithms for hypercubes and meshes of arbitrary dimension; most of our algorithms use virtualization. These results are summarized in Theorem 3.1.

Note that the running time of a scheduling algorithm is not measured by the competitive ratio. However, often good competitive algorithms are also fast. In particular, all our algorithms are very fast and simple to implement.

With virtualization (for ideally malleable jobs) we obtain the following results. For PRAMs the optimal competitive ratio is $2 + \phi$, where $\phi \approx 0.618$ is the golden ratio (or, more precisely, its inverse). In the restricted case when the size of a job (the number of processors requested by a job) is bounded by a constant fraction of the total number of processors, we obtain a tradeoff between the maximal size of a job and the optimal competitive ratio. As the size of jobs decreases, the optimal competitive ratio improves from $2 + \phi$ to 2 (see Figure 1). This shows that the size of jobs is a more important factor than in the model without dependencies. Without dependencies, the optimal competitive ratio on PRAMs for jobs with unrestricted size is 2 [8], which is the same as the optimal ratio for jobs requesting just one processor.

For more complex network topologies such as hypercubes and one-dimensional meshes with N processors, we obtain $O(\frac{\log N}{\log \log N})$-competitive algorithms, moreover, for one-dimensional meshes we prove that this is optimal. This bound is much larger than the constant ratio for PRAM. This shows that the network topology is a more important factor than in the model without dependencies, where it is possible to achieve a constant competitive ratio even for hypercubes and one-dimensional meshes [8].

Without virtualization (for non-malleable jobs) we show that if the size of jobs is not restricted, the competitive ratio is N, which implies that no on-line scheduling algorithm can use more than one processor efficiently. If we restrict the size of jobs, we again obtain a tradeoff, but the competitive ratio is much larger than in the model without virtualization (see Figure 2). For PRAMs a greedy algorithm achieves the competitive ratio given by this tradeoff. These results demonstrate that virtualization is a necessary technique for scheduling parallel jobs efficiently. Again, this is different from the case without dependencies, where the use of virtualization does not influence the competitive ratio [8].

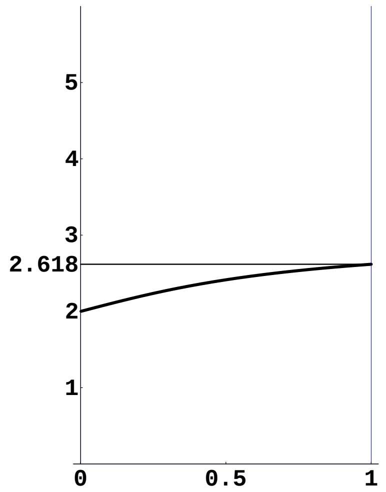

Figure 1: The relation between λ and the competitive ratio for PRAMs, using virtualization

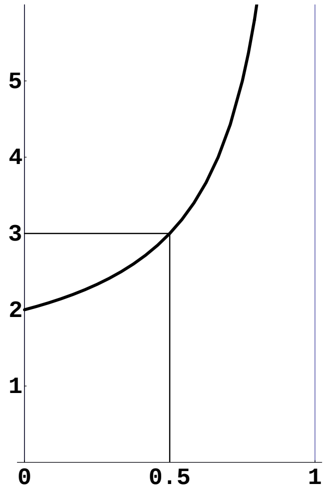

Figure 2: The relation between λ and the competitive ratio for PRAMs, without using virtualization

The main reason why scheduling with dependencies is much harder is that it is impossible to separate large jobs from small ones, for we cannot control when the large jobs become available. Without dependencies, we can always schedule the large jobs first, separately from smaller jobs, thus avoiding many difficulties.

We also show that to schedule job systems whose dependency graphs are trees is as difficult as to schedule job systems with general dependencies. In other words, on every network topology the optimal competitive ratio is the same for both cases.

In Sections 2 and 3, we state the basic definitions and our results. In Sections 4 and 5 we prove the results for PRAMs and other network topologies with virtualization. In Section 6, we give the bounds for scheduling without virtualization. In Section 7 we prove the result about tree dependency graphs. A preliminary version of this paper appeared in [7]. The claim of the lower bound for hypercubes in [7] is incorrect: while we conjecture that the $O(\log n/\log\log n)$-competitive algorithm is optimal similarly as for the one-dimensional mesh, we cannot prove this conjecture at the present time.

1.2 Related work

The new feature of our model is that it takes into account the dependencies between jobs. The previous work [8] assumes that all jobs are available at the beginning and independent of each other. In that case the competitive ratios for PRAMs, hypercubes, lines and two dimensional meshes are $(2-\frac{1}{N})$, $(2-\frac{1}{N})$, 2.5 and $O(\sqrt{\log\log N})$, respectively; except for the line these ratios are known to be optimal. These results can be extended to the model in which each job is released at some fixed time, instead of being known from the beginning, but the jobs must be independent of each other [21].

Both the work mentioned in the previous paragraph and this paper consider only deterministic algorithms. Subsequently, these results were extended to randomized algorithms in [18, 19]: Without dependencies it is possible to achieve a constant competitive ratio even for meshes of arbitrary constant dimension; on the other hand with dependencies our lower bound for one-dimensional meshes can be extended to the randomized algorithms, and hence randomization does not help.

These results together with the results of this paper give us a comprehensive picture of the importance of dependencies for on-line scheduling of parallel jobs. With dependencies between jobs, it is not even immediately clear that there exist efficient on-line scheduling algorithms at all. In principle jobs along a critical path in a dependency graph could necessarily accumulate substantial delays, which to some degree indeed happens. Even the presence of simple dependencies makes on-line scheduling much harder than the scheduling of independent jobs; the performance is much more sensitive to factors such as the use of virtualization, the size of jobs, and the network topology. On the other hand, the combinatorial structure of dependencies is not so important.

There is a considerable amount of literature on off-line scheduling, for a recent survey see [14]. It is NP-hard to find the optimal solution of off-line scheduling even for very simple special cases, and often it is NP-hard even to find an approximate solution within some factor, see [9, 2, 6, 15]. Our result for PRAM can be viewed as a generalization of the result of Graham [10] that list scheduling is an approximation algorithm with an approximation ratio of $(2-\frac{1}{N})$ for scheduling of sequential jobs on N processors with dependencies. Wang and Cheng [23] give an approximation algorithm for scheduling ideally malleable parallel jobs on PRAMs with dependencies which achieves approximation ratio of 3. Their algorithm is not on-line. We improve this result in Section 4 by presenting an on-line algorithm which achieves a competitive ratio of approximately 2.618, which is optimal for on-line algorithms.

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 [1, 12, 17]. 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. Virtualization was originally developed for the design of efficient and portable parallel programs for large-scale problems [3, 11, 13]. Our work complements these results by showing that virtualization is a necessary technique for efficient on-line scheduling of parallel jobs as well.

Recently there have been many results on on-line scheduling, for a survey see [20]. Similar model of on-line scheduling with unknown running times was considered in [21], who studied the case of sequential jobs running on machines of different speeds. Many of the new results consider scheduling of jobs with unknown running times without dependencies with the objective to minimize the total (or average) completion time, rather than makespan; good bounds have been proved for sequential jobs [16], ideally malleable parallel jobs [5], and even for a rather general class of non-ideally malleable jobs [4]. Other papers study also the scenario in which the jobs arrive one by one, or arrive over time but with known running times; see the survey [20] for the references.

2 Definitions

2.1 Parallel machines and parallel job systems with dependencies

For the purpose of scheduling, a parallel machine with a specific network topology is modeled as an undirected graph where each node u represents a processor p_u and each edge $\{u, v\}$ represents a communication link between p_u and p_v.

The *resource requirements* of a job are defined as follows. Given a graph H representing a parallel machine, let $\mathcal{G}$ be a set of subgraphs of H, each called a *job type*, that contains all singleton subgraphs and H itself. A *parallel job* J is characterized by a pair $J = (G, t)$ where $G \in \mathcal{G}$ is the requested subgraph and t is the running time of J on G. The *work* of J is $t|G|$, where $|G|$ is the *size* of the job J, i.e., the number of processors requested by J. A *sequential job* is a job requesting one processor.

A *parallel job system* $\mathcal{J}$ is a collection of parallel jobs. A *parallel job system with dependencies* is a directed acyclic graph $\mathcal{F} = (\mathcal{J}, \mathcal{E})$ describing the dependencies between the parallel jobs. A directed edge $(J, J') \in \mathcal{E}$ indicates that the job J' depends on the job J, i.e., J' cannot start until J has finished.

2.2 Virtualization

During the execution of a parallel job system, an available job may request p processors while only $p' < p$ processors are available. Using *virtualization* [11], it is possible to simulate a virtual machine of p processors on p' processors. 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* [1, 12]. Neither the running time nor the work can be decreased by virtualization, i.e., $\alpha(G, G') \geq \max(1, \frac{|G|}{|G'|})$. If the network topology of G can be efficiently mapped on G', the work does not increase. In this paper we assume that the simulation assumptions always preserve the work, i.e., we consider only ideally malleable jobs. This is a reasonable assumption, since efficient mappings exist for all network topologies we consider. For example, the computation on a d-dimensional hypercube can be simulated efficiently on a d'-dimensional hypercube for $d' < d$ by increasing the running time by a simulation factor of $2^{d-d'}$.

2.3 The scheduling problem and the performance measure

A *scheduling problem* is specified by a network topology together with a set of available job types and all simulation factors. A family of machines with a growing number of processors and the same network topology usually determines the job types and simulation factors in a natural way (see Section 2.4 for examples). An *instance* of the scheduling problem is a parallel job system with dependencies $\mathcal{F}$ and a machine H with the given topology. The output is a *schedule* for $\mathcal{F}$ on the machine H. A schedule is an assignment of a subgraph $G_i' \subseteq H$ and a time interval t_i' to each job $(G_i, t_i) \in \mathcal{F}$ such that the length of t_i' is $\alpha(G_i, G_i')t_i$; at any given time all subgraphs assigned to different currently running jobs are pairwise disjoint, and each job is scheduled after all the jobs it depends on have finished.

A scheduling algorithm is *off-line* if it receives the complete information as its input, i.e., all jobs and their dependencies and running times. It 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. It 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 it has no information about the future jobs and their dependencies.

We measure the performance by the *competitive ratio* [22]. Let $T_S(\mathcal{F})$ be the length of a schedule computed by an algorithm S. Let $T_{\mathrm{opt}}(\mathcal{F})$ be the length of an optimal schedule. A scheduling algorithm S is *σ-competitive* if $T_S(\mathcal{F}) \leq \sigma T_{\mathrm{opt}}(\mathcal{F})$ for every $\mathcal{F}$. Note that to decide whether a given schedule is optimal is coNP-complete [9].

2.4 Network topologies

PRAM: A parallel random access machine with N processors is modeled by a complete graph. Available job types are all PRAMs with at most N processors. A job (G, t) can run on any $p \leq |G|$ processors in time $\frac{|G|}{p}t$, i.e., the simulation factor is $\frac{|G|}{p}$.

Hypercube: A d-dimensional hypercube has $N = 2^d$ processors indexed from 0 to $N-1$. Two processors are connected if their indices differ by exactly one bit. Available job types are all d'-dimensional subcubes for $d' \leq d$. Each job J is characterized by the dimension d' of the requested subcube. The job J can run on a d''-dimensional hypercube in time $2^{d'-d''}t$ for $d'' \leq d'$.

Line: A line (one-dimensional mesh) has N processors $\{p_i \,|\, 0 \leq i < N\}$. Processor p_i is connected with processors p_{i+1} and p_{i-1} (if they exist). Available job types are all segments of at most N connected processors. The job (G, t) can run on a line of p processors in time $\frac{|G|}{p}t$ for $p \leq |G|$.

For $d \geq 2$, d-dimensional meshes and d-dimensional jobs are defined similarly.

3 Overview of the results

Theorem 3.1 *Let N be the number of processors of a machine. Suppose that no job requests more than λN processors, where $0 < \lambda \leq 1$.*

Without virtualization we have the following results:

network topology	*upper bound*	*lower bound*
arbitrary, $\lambda = 1$	N	N
PRAM, $0 < \lambda < 1$	$1 + \frac{1}{1-\lambda}$	$1 + \frac{1}{1-\lambda}$

With virtualization we have the following results:

network topology	*upper bound*	*lower bound*
PRAM, $\lambda = 1$	$2 + \phi$	$2 + \phi$
PRAM, $0 < \lambda \leq 1$	$2 + \frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$	$2 + \frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$
hypercube	$O(\frac{\log N}{\log\log N})$	
line	$O(\frac{\log N}{\log\log N})$	$\Omega(\frac{\log N}{\log\log N})$
d-dimensional mesh	$O((\frac{\log N}{\log\log N})^d)$	$\Omega(\frac{\log N}{\log\log N})$

All our algorithms that achieve the upper bounds are fully on-line and simple to implement. In contrast, all our lower bounds apply to all on-line algorithms, not just to fully on-line algorithms.

The competitive ratios for PRAMs are the best constant 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.

All our lower bound results assume that the running time of a job may be zero. This slightly unrealistic assumption 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.

For PRAMs, both with and without virtualization, if λ approaches 0, the competitive ratio approaches 2 (see Figures 1 and 2), which is the optimal ratio for PRAM without dependencies. This means that if the size of the jobs is small, the dependencies have little influence on the performance of on-line scheduling.

The next theorem shows that the structure of a dependency graph is less important than it seems at first.

Theorem 3.2 *Let an on-line scheduling problem, i.e., a specific network topology and simulation factors, be given. Then the optimal competitive ratio for this problem is equal to the optimal competitive ratio for a restricted problem in which we allow as inputs only job systems whose dependency graphs are trees.*

In our proofs regarding algorithms that are not fully on-line we sometimes use a job system that is exponentially large, but this does not change the performance of our algorithms. It only leaves open a possibility that if we restrict the size of job systems, it might be possible to improve performance by looking at a dependency graph (which is likely to make the scheduling algorithm slow). However, our proofs can be modified to show that such an improvement cannot be very significant. Also, in practice we are interested in efficient scheduling over a long period of time, in which case the number of jobs is large.

4 Scheduling on PRAMs with virtualization

The most important result of this section is an optimal $(2+\phi)$-competitive algorithm for scheduling ideally malleable jobs on PRAMs, where $\phi = \frac{\sqrt{5}-1}{2} \approx 0.618$ 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 [23]. 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.

Let $T_{\max}(\mathcal{F})$ be the maximal sum of running times of jobs along any path in a dependency graph $\mathcal{F}$. Clearly $T_{\max} \leq T_{\mathrm{opt}}(\mathcal{F})$ because an optimal algorithm has to schedule these jobs sequentially.

Suppose S is a scheduling algorithm for a parallel machine H. The *efficiency* of S at time t is the number of busy processors divided by $|H|$. For $\alpha \leq 1$, let $T_{<\alpha}(S, \mathcal{F})$ be the total time during which the efficiency of S is less than α.

The next lemma is from [8]. It is very useful for analyzing scheduling algorithms, and it is valid also for job systems with dependencies. It says that if a schedule has high efficiency except for a short period of time, then the schedule cannot be much longer than an optimal one. This is true because we require the amount of work to be preserved in all schedules for the same job system.

Lemma 4.1 *Let S be a schedule for a parallel job system $\mathcal{F}$ such that the work of each job is preserved. If $T_{<\alpha}(S, \mathcal{F}) \leq \beta T_{\mathrm{opt}}(\mathcal{F})$, where $\alpha \leq 1$ and $\beta \geq 0$, then $T_S(\mathcal{F}) \leq (\frac{1}{\alpha} + \beta) T_{\mathrm{opt}}(\mathcal{F})$.*

Another useful lemma is from [10], and it also applies to our model with parallel jobs. This lemma enables us to bound the time when the efficiency is low, which we need to do in order to apply Lemma 4.1.

Lemma 4.2 *Let S be a schedule for a parallel job system $\mathcal{F}$. Then there exists a path of jobs $J_k, \ldots, J_1, J_0$ in the dependency graph such that whenever there is no job available to be scheduled, some job J_i is running.*

Now we present our algorithm for PRAMs. The basic idea is to maintain an efficiency of at least α for some constant $0 < \alpha \leq 1$. In order to do so, we sometimes have to schedule large jobs on fewer processors than they request.

The algorithm maintains a queue $\mathcal{Q}$ that contains all jobs available at the current time. When a job becomes available, it is added to $\mathcal{Q}$ immediately. First($\mathcal{Q}$) is a function such that for each nonempty $\mathcal{Q}$, First($\mathcal{Q}$) $\in \mathcal{Q}$. It may choose an arbitrary job in $\mathcal{Q}$. In practice we suggest selecting the job based on the best fit heuristic, i.e., if possible select the largest job which can be scheduled on the available processors without virtualization.

Algorithm PRAM(α), ($\frac{1}{2} < \alpha \leq 1$)
while not all jobs are finished **do**
  if the efficiency is less than α
    then begin
      schedule First($\mathcal{Q}$) on the requested number of processors or
      on all available processors, whichever is less;
      $\mathcal{Q} \leftarrow \mathcal{Q} - \text{First}(\mathcal{Q})$;
    end.

Theorem 4.3 *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 4.4 *If the number of requested processors is unrestricted ($\lambda = 1$), then PRAM(ϕ) is $(2+\phi)$-competitive, where $\phi = \frac{\sqrt{5}-1}{2}$ is the golden ratio.*

Proof of Theorem 4.3. First, observe that when $\mathcal{Q}$ is nonempty, 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 4.2 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 $T + (1-\alpha)(\frac{1}{\alpha} + \frac{1}{\lambda})T' + T'' \leq (1 + \frac{1}{\alpha})T_{\text{opt}}$. Since $(1-\alpha)(\frac{1}{\alpha} + \frac{1}{\lambda}) = 1$, the competitive ratio is $\frac{T+T'+T''}{T_{\text{opt}}} \leq 1 + \frac{1}{\alpha} = \sigma$. □

Now we prove a matching lower bound.

Theorem 4.5 *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 by a factor of $2+\phi$ than the requested number.

The job system used by the adversary has 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 3.)

Now we describe the adversary's strategy. The adversary can enforce that at each level the

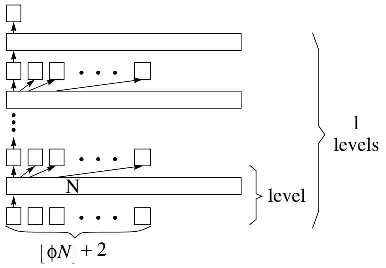

Figure 3: Example of the job system used in the proof of Theorem 5.2

parallel job always depends on the sequential job that was scheduled last. This is possible since the algorithm is fully on-line and hence cannot distinguish between the sequential jobs.

If a parallel job is scheduled on fewer than $(1-\phi)N$ processors, it is slowed down by a factor greater than $\frac{1}{1-\phi}$. 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 $\frac{1}{1-\phi} = 2+\phi$ and we are done.

Otherwise the adversary assigns the running time 0 to all parallel jobs and all the sequential jobs on which the parallel jobs depend. All other sequential jobs are assigned running time T for some preset time T, except for the last sequential job, which is assigned running time $T' = \phi l T$. The previous paragraph shows that 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 length of this schedule is $\lceil \frac{l(\lfloor \phi N \rfloor + 1)T}{N-1} \rceil$, which is arbitrarily close to $\phi l T$ for sufficiently large N and l. Therefore the competitive ratio is arbitrarily close to $\frac{1+\phi}{\phi} = 2+\phi$. 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 last level. This scheme only changes if some parallel job is scheduled on too few processors.

Let $\sigma \geq 2$ be the competitive ratio that we want to achieve as a lower bound. 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. The time T' is chosen so that the optimal schedule for the previously scheduled jobs takes time T' on $N-1$ processors. This forces the competitive ratio to be arbitrarily close to $1 + \frac{1}{L}$.

So 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 sets the running time of J to $T' = \frac{A}{1-\lambda}\bar{T}$ and removes all jobs that have not been scheduled. Then the generated schedule takes time $\bar{T} + \frac{\lambda}{1-\alpha_i}T' = (\frac{1-\lambda}{A} + \frac{\lambda}{1-\alpha_i})T' \geq (\frac{1-\lambda}{A_{i-1}} + \frac{\lambda}{1-\alpha_i})T' = \sigma T'$. The optimal schedule runs first all parallel jobs and then all sequential jobs together with the parallel job with running time T'. The time T' was chosen so that the length of the optimal schedule is within an arbitrarily small factor of T' for large l and N. 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 4.3. 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 fanout 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. □

5 Scheduling on hypercubes and meshes with virtualization

In contrast to scheduling on PRAMs, jobs on hypercubes and meshes require a structured subregion of a parallel machine. Thus the geometry of the underlying network topology has to be considered.

First we present an algorithm for a d-dimensional hypercube. A *normal* k-dimensional subcube is a subcube of all processors with all but the first k coordinates fixed. Let h be the smallest power of two such that $h \log h \geq d$. Notice 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 of dimension between $d - i\log h + 1$ and $d - (i-1)\log h$. $\mathcal{Q}_i$ is a queue for the available jobs in $\mathcal{J}_i$. The hypercube is partitioned into h normal $(d-\log h)$-dimensional subcubes $M_1, \ldots, M_h$. Jobs from $\mathcal{J}_i$ are scheduled on M_i only.

Algorithm HYPERCUBE
while not all jobs are finished **do**
  for all i such that $\mathcal{Q}_i \neq \emptyset$ **do**
    if a normal $(d - i \log h)$-dimensional subcube in M_i is available
      then begin
        schedule First($\mathcal{Q}_i$) on that subcube;
        $\mathcal{Q}_i \leftarrow \mathcal{Q}_i -$ First($\mathcal{Q}_i$);
      end.

Theorem 5.1 *The competitive ratio of HYPERCUBE is $O(\frac{d}{\log d}) = O(\frac{\log N}{\log \log N})$ for a d-dimensional hypercube with $N = 2^d$ processors.*

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 Q_i is nonempty, the efficiency of the subcube M_i is equal to 1, and the overall efficiency is at least $\frac{1}{h}$. Second, no job is slowed down by a factor greater than h and hence from Lemma 4.2 it follows that $T_{<\frac{1}{h}} \leq hT_{\max}$. Hence, by Lemma 4.1, 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 $k^k > n$. Notice that $k = O(\frac{\log n}{\log \log n})$. We maintain $h = k^d$ queues. The jobs are partitioned into h 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 $(a_1, a_2, ..., a_d)$ such that $\frac{n}{k^{i_j}} < a_j \leq \frac{n}{k^{i_j - 1}}$. $\mathcal{Q}_i$ is a queue for the available jobs from $\mathcal{J}_i$. The mesh is partitioned into h submeshes M_i of size $\lfloor \frac{n}{k} \rfloor \times \cdots \times \lfloor \frac{n}{k} \rfloor$. 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 such that $\mathcal{Q}_i \neq \emptyset$ **do**
    if an $\lfloor \frac{n}{k^{i_1}} \rfloor \times \cdots \times \lfloor \frac{n}{k^{i_d}} \rfloor$ submesh in M_i is available
      then begin
        schedule First($\mathcal{Q}_i$) on such a submesh with the smallest coordinates;
        $\mathcal{Q}_i \leftarrow \mathcal{Q}_i -$ First($\mathcal{Q}_i$);
      end.

The proof of following theorem is similar to that of Theorem 5.1 and is omitted.

Theorem 5.2 *MESH is $O((\frac{\log N}{\log \log N})^d)$-competitive for a d-dimensional mesh of $N = n^d$ processors.*

5.1 A lower bound

In this section, we prove that the competitive ratios of our algorithm for one dimensional meshes is within a constant factor of the optimal competitive ratio.

This result can be generalized to randomized algorithms, as shown in [18, 19]. The proof is based on similar ideas, however, the argument is significantly more delicate as the adversary has to specify the running times independently of the random choices of the algorithm.

Our approach to this lower bounds is similar to the method used in [8]. The adversary tries to block a large fraction of processors by small jobs that use only a small fraction of all processors. Dependencies give the adversary more control over the size of available jobs, which results in larger lower bounds.

Theorem 5.3 *No deterministic 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.*

Proof. Put $s = 2\lceil \log N \rceil$ and $t = \lfloor \frac{1}{2} \log_s N \rfloor$. Notice that $t = \Theta(\frac{\log N}{\log\log N})$, $t \leq s$ and $s^{2t} \leq N$.

Our job system has $t^2 N$ independent chains, each consisting of t^2 jobs, which is a total of $t^4 N$ jobs. In each chain the sizes of jobs are $s^2, s^4, \ldots, s^{2t}$, repeated s times, i.e., the $jt+i$-th job in each chain has size s^{2i}.

During the schedule the adversary ensures that only one job in each chain has nonzero running time, i.e., whenever he removes a job with non-zero time, he also removes all remaining jobs in its chain. The adversary also maintains that at any moment all available jobs are at the same position in their chain. The *i-th subphase* of the *j-th phase* is the part of the process when each available job is the $jt+i$-th job in its chain.

A *normal k-segment* is a segment of k processors starting at a processor with whose position is divisible by k. A segment is *used* if at least one of its processors is busy.

The adversary chooses some time T and then reacts to the actions of the algorithm by the following steps.

SINGLE JOB: If the algorithm schedules a job of size s^{2i} on a segment with fewer than $2s^{2i-1}$ processors, then the adversary removes all other jobs, both running and waiting, and runs this single job for a sufficiently long time.

CLEAN UP: If the time since the last CLEAN UP or since the beginning of the schedule is equal to T and there was no SINGLE JOB step, the adversary ends the current phase, i.e., he removes all running jobs, all remaining jobs in those chains, and all remaining jobs of the current phase in all chains.

DECREASE EFFICIENCY: If in the i-th subphase of any phase at least $\frac{3N}{t-1}$ processors are busy, the adversary does the following. For every used normal s^{2i+1}-segment he selects one running job that uses this segment (i.e., at least one processor of it). He keeps running these jobs and removes all other running jobs and all remaining jobs in their chains, and all available jobs of length s^{2i}. This ends the current subphase.

Evaluation of the adversary's strategy

We can assume that the algorithm never allows a SINGLE JOB step, otherwise it obviously cannot be t-competitive.

Our analysis is based on the following lemma.

Lemma 5.4 *During the i-th subphase of any phase,*

(i) only jobs of size s^{2i} and smaller are running and only jobs of size s^{2i} are available to be scheduled;

(ii) the total length of used normal s^{2i-1}-segments is at least $\frac{(i-1)N}{t-1}$.

Proof. We prove this inductively. At the beginning of the schedule (i) and (ii) are trivial.

During any subphase no job is removed, so (i) and (ii) remain true. ¿From one subphase to the next one, (i) is obviously maintained.

To prove that (ii) is maintained, we first show that at the beginning of each subphase the number of busy processors is at most $\frac{N}{s}$. This is trivial for the first subphase of each phase. Otherwise the subphase is started by a DECREASE EFFICIENCY step, in which for each normal s^{2i-1}-segment only one job is left running. By (i), these jobs have size at most s^{2i-2} and thus the number of busy processors is at most $\frac{N}{s}$, because the normal segments are pairwise disjoint.

So during the i-th subphase new jobs must be scheduled on at least $\frac{3N}{t-1} - \frac{N}{s} \geq \frac{2N}{t-1}$ processors to cause the next DECREASE EFFICIENCY step. Scheduled jobs have length s^{2i} and therefore (to avoid SINGLE JOB) at least half of the processors assigned to them are in whole normal s^{2i-1}-segments. These segments could not be used at the beginning of the subphase. So during the subphase the total length of used normal s^{2i-1}-segments increases by at least $\frac{N}{t-1}$ and at the end of the subphase it is at least $\frac{iN}{t-1}$. The length of used s^{2i+1}-segments is at least as large, since the normal segments of different lengths are aligned. During the DECREASE EFFICIENCY step the used normal s^{2i+1}-segments remain used, hence their total length does not decrease. This proves that (ii) is true at the beginning of the next subphase.

¿From (ii) it follows that at the beginning of the t-th subphase all normal s^{2t-1}-segments are used and so no available job can be scheduled. Hence a CLEAN UP step ends every phase and both (i) and (ii) are also true at the beginning of the next phase. □

The last paragraph of the proof also proves that every phase takes time T. Every DECREASE EFFICIENCY or CLEAN UP step removes at most n chains. Thus there are sufficiently many chains available for t phases, and the whole schedule takes at least time tT.

Every chain contains at most one job with non-zero time, so $T_{\max} \leq T$, at most $\frac{1}{t}$ of the length of the schedule; moreover all these jobs are independent and we can use the results on scheduling without dependencies from [8]. The efficiency of the on-line schedule was at most $\frac{3}{t}$ all time, so by the results from [8] there is an off-line schedule $O(t)$-times shorter. This finishes the proof of Theorem 5.3 □

6 Scheduling without virtualization

The difficulty of on-line scheduling without virtualization is best seen by the lower bounds in this section. Theorem 6.1 implies that no efficient scheduling is possible if virtualization is not used and the number of processors requested by a job is not restricted. This demonstrates the importance of virtualization in the design of competitive scheduling algorithms. It also shows that on-line scheduling with dependencies is fundamentally different from scheduling without dependencies. Without dependencies neither the size of the largest job nor virtualization changes the optimal competitive ratios dramatically [8].

Theorem 6.1 *Without virtualization, no on-line scheduling algorithm can achieve a better competitive ratio than N on any machine with N processors.*

Remark. The corresponding upper bound can be achieved by scheduling one job at a time.

Proof. The job system used by the adversary consists of N independent chains. Each chain starts with a sequential job and has N sequential jobs alternating with N parallel jobs requesting N processors. The adversary assigns the running times dynamically so that exactly one sequential job in each chain has running time T for some predetermined T, and all other jobs have running time 0.

In the beginning the on-line algorithm can only schedule sequential jobs. The adversary keeps one of the sequential jobs running and assigns running time 0 to all other sequential jobs on the first level, both running and available. Only one processor is busy, while all other processors are idle because no parallel job can be scheduled. The adversary keeps the chosen sequential job running for time T. Then he terminates it and sets the running times of all remaining jobs in this chain to 0. This process is repeated N times. Each time at most one parallel job of each chain can be processed, hence the schedule takes time at least NT.

The optimal schedule first schedules all jobs before the sequential jobs with running time T, then the N sequential jobs with time T in parallel, and then the remaining jobs. The total time is T, and hence the competitive ratio of the on-line scheduling algorithm is at least N. $\square$

Now we prove a lower bound on the competitive ratio if the number of processors requested by a job is restricted. The proof is given for the fully on-line algorithms only. It can be generalized to all on-line algorithms by the method used in the proof of Theorem 4.5.

Theorem 6.2 *Suppose that the largest number of processors requested by a job on a machine with N processors is λN. Then no scheduling algorithm without virtualization can achieve a smaller competitive ratio than $1 - \frac{1}{1-\lambda}$.*

Proof. The proof is similar to that of Theorem 4.5. 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 requesting $\lfloor \lambda N \rfloor$ 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 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 adversary assigns to each job a time of either 0, at most T, or T'. Moreover, all jobs with nonzero time are independent of each other. The off-line algorithm first schedules all jobs with running 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 $\lambda < 1$. $\square$

Now we show that this lower bound is tight for PRAM. In fact, a simple greedy algorithm achieves this bound.

Algorithm GREEDY
while not all jobs are finished **do**
  if the resource requirements of First($\mathcal{Q}$) can be satisfied
    then begin
      schedule First($\mathcal{Q}$) on a requested subgraph;
      $\mathcal{Q} \leftarrow \mathcal{Q} - \text{First}(\mathcal{Q})$;
    end.

Theorem 6.3 *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 4.2 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 4.1 finishes the proof. $\square$

7 Tree dependency graphs

In this section we prove Theorem 3.2. First notice 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 $\mathcal{F}'$.

The more difficult case is Theorem 3.2, where the on-line algorithm may know the dependency graph in advance.

Theorem 3.2 *Let an on-line scheduling problem, i.e., a specific network topology and simulation factors, be given. Then the optimal competitive ratio for this problem is equal to the optimal competitive ratio for a restricted problem in which we allow as inputs only job systems whose dependency graphs are trees.*

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 depend 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. $\square$

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.

8 Conclusions

In this paper we have presented efficient algorithms for one of the most general scheduling problems, which is of both practical and theoretical interest. We have shown that virtualization, the size of jobs, and the network topology are important issues in our model, but the structure of the dependency graph is not as important. Subsequent work also shows that randomization cannot help to improve the performance.

Acknowledgment

We would like to thank Avrim Blum, Steven Rudich, Danny Sleator, Andrew Tomkins and Joel Wein for helpful comments and suggestions.

References

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

[2] Jacek Błażewicz, Mieczysław Drabowski, and Jan Węglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Trans. Comput.*, c-35(5):389–393, 1986.

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

[4] Tim Brecht, Donald D. Chinn, Xiaotie Deng, and Jeff Edmonds. On preemptively scheduling parallel jobs with changing execution characteristics. Manuscript, 1996.

[5] Xiaotie Deng, Nian Gu, Tim Brecht, and KaiCheng Lu. Preemptive scheduling of parallel jobs on multiprocessors. In *Proc. of the 7th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 159–167. ACM-SIAM, 1996.

[6] Jianzhong Du and Joseph Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM J. Disc. Math.*, 2(4):473–487, 1989.

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

[8] Anja Feldmann, Jiří Sgall, and Shang-Hua Teng. Dynamic scheduling on parallel machines. *Theoretical Comput. Sci.*, 130(1):49–72, 1994. Preliminary version appeared in Proc. of the 32nd Ann. IEEE Symp. on Foundations of Computer Sci., 111–120, IEEE, 1991.

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

[10] R. L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical J.*, 45:1563–1581, November 1966.

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

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

[13] H. T. Kung. Computational models for parallel computers. Technical report CMU-CS-88-164, Carnegie-Mellon University, 1987.

[14] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S. C. Graves, A. H. G. Rinnooy Kan, and P. Zipkin, editors, *Handbooks in Operations Research and Management Science, Vol. 4: Logistics of Production and Inventory*, pages 445–552. North-Holland, 1993.

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

[16] Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. In *Proc. of the 4th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 422–431. ACM-SIAM, 1993.

[17] Abhiram G. Ranade. How to emulate shared memory. In *Proc. of the 28th Ann. IEEE Symp. on Foundations of Computer Sci.*, pages 185–194. IEEE, 1987.

[18] Jiří Sgall. *On-Line Scheduling on Parallel Machines.* PhD thesis, Technical Report CMU-CS-94-144, Carnegie-Mellon University, Pittsburgh, PA, U.S.A., 1994.

[19] Jiří Sgall. Randomized on-line scheduling of parallel jobs. *J. of Algorithms*, 21:149–175, 1996. Preliminary version appeared in Proc. of the 3rd Israel Symp. on Theory of Computing and Systems, 241–150, IEEE, 1995.

[20] Jiří Sgall. On-line scheduling — a survey. Submitted.

[21] David B. Shmoys, Joel Wein, and David P. Williamson. Scheduling parallel machines on-line. In *Proc. of the 32nd Ann. IEEE Symp. on Foundations of Computer Sci.*, pages 131–140. IEEE, 1991.

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

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