Optimal Online Scheduling of Parallel Jobs with Dependencies

Anja Feldmann* Ming-Yang Kao† Jiří Sgall‡

Shang-Hua Teng§

December 1992

Abstract

We study the following general online scheduling problem. Parallel jobs arrive dynamically according to the dependencies between them. Each job requests a certain number of processors with a specific communication configuration, but its running time is not known until it is completed. We present optimal online algorithms for PRAMs, hy-

*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.

‡School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. On leave from Mathematical Institute, ČSAV, Žitná 25, 115 67 Praha 1, Czechoslovakia

§Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139. Part of the work was done at Xerox Palo Alto Research Center.

percubes and one-dimensional meshes, and obtain optimal tradeoffs between the competitive ratio and the largest number of processors requested by any job. Our work shows that for efficient online scheduling it is necessary to use *virtualization*, i.e., to schedule parallel jobs on fewer processors than requested while preserving the work.

Assume 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 a machine. With virtualization, our algorithm for PRAMs has a competitive ratio of $2 + \frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$. Our lower bound shows that this ratio is optimal. As λ goes from 0 to 1, the ratio changes from 2 to $2+\phi$, where $\phi \approx 0.618$ is the golden ratio. For hypercubes and one-dimensional meshes we present $\Theta(\frac{\log N}{\log\log N})$-competitive algorithms as well as matching lower bounds. Without virtualization, no online scheduling algorithm can achieve a competitive ratio smaller than N for $\lambda = 1$. For $\lambda < 1$, the lower bound is $1 + \frac{1}{1-\lambda}$ and our algorithm for PRAMs achieves this competitive ratio.

We prove that tree constraints are *complete* for the scheduling problem, i.e., any algorithm that solves the scheduling problem if the dependency graph is a tree can be converted to solve the general problem equally efficiently. This shows that the structure of a dependency graph is not as important for online scheduling as it is for offline scheduling, although even simple dependencies make the problem much harder than scheduling independent jobs.

Keywords: scheduling, dependencies, parallel computing, parallel jobs, virtualization, online algorithms, competitive analysis

AMS(MOS) subject classification: 68Q99, 68Q25, 68Q22

1 Introduction

This paper investigates the following general online 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 machine with a certain number of processors and a specific communication configuration. The running time of a job can only be determined by actually running the job.

A *scheduling algorithm* computes a schedule that satisfies the resource requirements and respects the dependencies between the parallel jobs. The performance of an online scheduling algorithm is measured by the *competitive ratio* [12]: the worst-case ratio of the total time (the makespan) of a schedule it computes to that of an optimal schedule, computed by an offline algorithm that knows all jobs, 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 the most reasonable 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. 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 on the smaller set

preserves the network topology, which is true for common architectures including PRAMs, meshes and hypercubes. The work of a job is preserved and the running time increases proportionally. This technique is called *virtualization* [2, 8, 10, 11]. We show that it is essential for efficient online scheduling.

We prove that in the restricted model where a job cannot be scheduled on fewer processors than it requests, no efficient online scheduling is possible, unless some additional restrictions are imposed, such as a bound on the number of processors a job can request. This result demonstrates a fundamental difference between online scheduling with and without dependencies — the previous work [5] shows that for scheduling without dependencies virtualization is not essential.

We present optimal online scheduling algorithms for PRAMs (shared memory parallel machines), hypercubes and one-dimensional meshes, most of them use virtualization. These results are summarized in the tables in Theorem 3.1. All our lower bounds apply even if an online algorithm knows the dependencies and resource requirements of all jobs in advance. In contrast all our algorithms are fully online, i.e., they receive this information on line as well.

With virtualization we obtain the following results. For PRAMs we show a tradeoff between the optimal competitive ratio and the largest number of processors requested by a job. Let λN be the largest number of processors requested, where $0 < \lambda \leq 1$ and N is the number of processors of a machine. Our algorithm for PRAMs has an optimal competitive ratio of $2 + \frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$.

For $\lambda = 1$, i.e., with no restrictions on the number of processors requested by a job, this ratio equals $2 + \phi$ where $\phi \approx 0.618$ is the golden ratio. If λ is close to 0, the competitive ratio approaches 2, which is the optimal ratio for scheduling sequential jobs even without dependencies [5, 13]. Our algorithms for hypercubes and one-dimensional meshes have competitive ratios of $\Theta(\frac{\log N}{\log \log N})$. We prove matching lower bounds showing that these competitive ratios are within constant factors of the optimal ones. The difference between the above optimal ratios for different topologies shows that the network topology of a parallel machine needs to be considered in order to schedule parallel jobs efficiently.

Without virtualization, we again obtain a tradeoff between the optimal competitive ratio and the largest number of processors requested by a job. For $\lambda < 1$, we have a stronger lower bound of $1 + \frac{1}{1-\lambda}$ for an arbitrary network topology. We give an algorithm for PRAMs with a competitive ratio matching this lower bound. For $\lambda = 1$, i.e., the number of requested processors is not restricted, we have a much more dramatic result. We prove a lower bound of N on the competitive ratio. This implies that no online scheduling algorithm can use more than one processor efficiently! Consequently, if there are dependencies between jobs, virtualization is a necessary technique for scheduling parallel jobs efficiently.

We show that tree dependency graphs are *complete* for the scheduling with dependencies — any online 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. This is easy to prove for fully online algorithms, but a more difficult proof is required for

the algorithms that may know the dependencies and resource requirements in advance. This result shows that the combinatorial structure of a dependency graph is much less relevant for our model than for offline scheduling, although even the presence of simple dependencies makes the scheduling problem much harder than online scheduling of independent jobs. This completeness result enables us to focus on simple dependency graphs in the lower bound proofs.

The new feature of our model is that it takes into account the dependencies between jobs. The previous work [5] 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 optimal.[1] 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 are independent of each other [13]. With the dependencies between jobs, online scheduling becomes much more difficult. It is not immediately obvious that there exist efficient online scheduling algorithms at all. In principle jobs along a critical path in a dependency graph could be forced to accumulate substantial delays. We show that this indeed happens if the communication topology is complex, e.g., hypercubes and one-dimensional meshes, or if the jobs request many processors and virtualization is not used.

Virtualization is an important technique originally developed for the design of efficient and portable parallel programs for large-scale problems [2,

[1] These improved results for PRAMs, hypercube and line will appear in the journal version of [5]

8, 10, 11]. Our work complements these results and shows that virtualization is a necessary technique for efficient online scheduling of parallel jobs as well.

Other work on online scheduling assumes that jobs are independent of each other and their release times are independent of scheduling [3, 4, 5, 7, 13, 14]. Furthermore, only the work in [5, 14] considers parallel jobs and the network topology of the parallel machine in question. Most of the other cited work emphasizes other issues, e.g., different speeds of processors. Dependencies between the jobs were previously considered only in the classical offline model when the dependency graph and the running times are given in advance (for example [6]).

In Sections 2 and 3 we state the basic definitions and our results. In Sections 4 and 5 we prove the results which are independent of the network topology — the completeness of tree dependency graphs and the lower bound for scheduling without virtualization. The results for PRAMs and other architectures are given in Sections 6 and 7.

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 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 a parallel machine G. The *work* of J is $|G|t$, 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 just 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. Each 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* [8, 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, 9]. 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 [1, 9]. For example, the computation on a d-dimensional hypercube might be simulated on a d'-dimensional hypercube for $d' < d$ by increasing the running time by the simulation factor $2^{d-d'}$.

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 detailed examples.) An *instance* of the problem is a parallel job system with dependencies $\mathcal{F}$ and a machine with the given topology. The output is a *schedule* efor $\mathcal{F}$ on the machine H. It 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 no two subgraphs assigned to different currently running jobs are overlapping and each job is scheduled after all the jobs it depends on have finished.

A scheduling algorithm is *offline* if it receives as its input the complete information: all jobs including their dependencies and running times. It is *online* 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 online* if it is online 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 dependencies.

We measure the performance by *competitive ratio* [12]. Let $T_S(\mathcal{F})$ be the length of a schedule computed by an algorithm S, and let $T_{\text{opt}}(\mathcal{F})$ be the length of an optimal schedule. A scheduling algorithm S is *σ-competitive* if $T_S(\mathcal{F}) \leq \sigma T_{\text{opt}}(\mathcal{F})$ for every $\mathcal{F}$. Note that deciding whether a given schedule

is optimal is coNP-complete [6].

2.4 Some network topologies

PRAM: For the purpose of online scheduling, a parallel random access machine of N processors is a complete graph. Available job types are all PRAMs of 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 analogically.

3 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, line	$O(\frac{\log N}{\log\log N})$	$\Omega(\frac{\log N}{\log\log N})$
d-dimensional mesh	$O((\frac{\log N}{\log\log N})^d)$	$\Omega(\frac{\log N}{\log\log N})$

Remarks. All algorithms in the upper bounds are fully online. All lower bounds apply to all online algorithms, not only to fully online ones.

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 0. This slightly unrealistic assumption can be removed easily. As all our proofs are constructive, we simply replace a zero time by a unit time and make all other running times sufficiently large. This only decreases the lower bounds by arbitrarily small additive constants.

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

Theorem 3.2 *Let S be an online scheduling algorithm for an arbitrary network topology. Suppose that S is σ-competitive for all job systems whose dependency graphs are trees. Then we can construct from S an online algorithm which is σ-competitive for all job systems with general dependency graphs.*

4 The completeness of tree dependency graphs

In this section we prove Theorem 3.2. Notice that a similar theorem for fully online algorithms is easy to prove: Suppose that we have a fully online algorithm for tree dependency graphs. If we run it on a general dependency graph, it behaves exactly the same as on a tree subgraph where for each job J we keep only the edge from a job J' such that J became available when J' finished. The generated schedule is the same, and the optimal schedule can only improve if some dependencies are removed. Therefore the competitive ratio of the algorithm does not change for general dependency graphs.

The important case is Theorem 3.2, where the algorithms may know the dependency graph in advance.

Proof of Theorem 3.2. For 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 $\mathcal{F}$. The running times of jobs in $\mathcal{F}'$ are determined 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 any copy of J are the same as those of J. A path p is the *last copy* of J if this is the last copy of J to be scheduled. Let $\mathcal{F}''$ be the subgraph of $\mathcal{F}'$ consisting of the last copies of all jobs of $\mathcal{F}$ and all dependencies between them.

Our scheduling strategy works as follows. We run S on $\mathcal{F}'$. Suppose S schedules $p \in \mathcal{F}'$. There are two possibilities. (i) p is the last copy of some J. In this case we schedule J to the same set of processors as p was scheduled by S. (ii) p is not the last copy. Then we remove p and all jobs dependent on it. If a job $J \in \mathcal{F}$ is finished, we stop its last copy $p \in \mathcal{F}'$.

Notice that if p is the last copy of J, then we schedule J in 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 that our schedule is correct, 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 job have running time 0 and can be scheduled at the end. By construction, any schedule for $\mathcal{F}$ corresponds to a schedule of

$\mathcal{F}''$ and therefore to a schedule of $\mathcal{F}'$. So the competitive ratio for $\mathcal{F}$ is not larger than the competitive ratio for $\mathcal{F}'$ and this is at most σ according to 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 online scheduling from the viewpoint of competitive analysis.

5 Why is virtualization necessary?

The difficulty of online scheduling without virtualization is best seen by the strong lower bounds of this section. Theorem 5.1 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 online scheduling with dependencies is fundamentally different from scheduling without dependencies. Without dependencies neither the size of the largest job nor virtualization changed the optimal competitive ratios dramatically [5].

Theorem 5.1 *Without virtualization, no online scheduling algorithm can achieve a better competitive ratio than N on any machine with N processors.*

Remark. Notice that 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. See Figure 1 for illustration. 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.

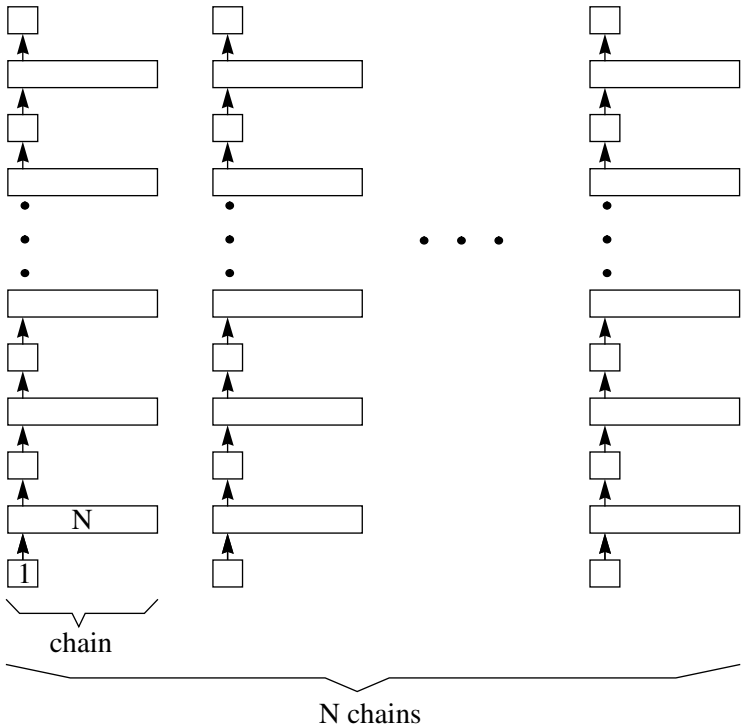

Figure 1: Example of the job system as used in the proof of Theorem 5.1

In the beginning the 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.

There is only one processor 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 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 online scheduling algorithm is at least N. □

Now we prove a lower bound on the competitive ratio if the number of processors requested by a job is restricted.

Theorem 5.2 *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. We first present a proof for fully online scheduling algorithms. We then sketch how to modify the job system for any online algorithm.

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. See Figure 2 for illustration.

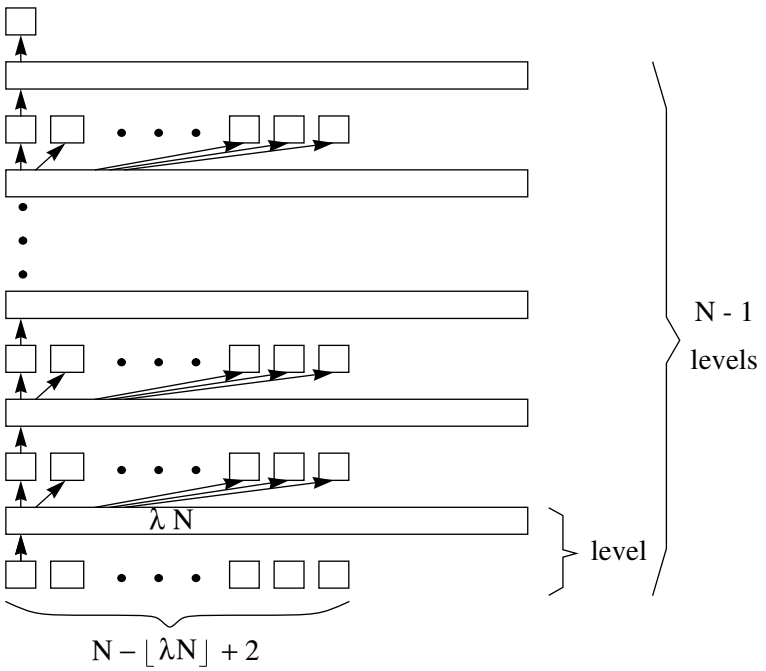

Figure 2: Example of the job system as used in the proof of Theorem 5.2

In the beginning the algorithm can schedule only sequential jobs. The adversary forces that the sequential job on which the parallel job depends 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. Note that during this time the scheduling algorithm cannot schedule the parallel job. As soon as the parallel job is scheduled, the adversary immediately 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 assigned 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 offline 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 $(N-1)(N - \lfloor \lambda N \rfloor + 1)$ of 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 offline 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.

We now modify the job system to handle the case where the online algorithm knows the job system in advance. The proof is similar to the proof of Theorem 3.2. We generate sufficiently many copies of each job, so that the graph is very symmetric and the scheduling algorithm cannot take an advantage of the additional knowledge. 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 wide tree which is exponentially larger. The adversary strategy is the same except for the following modification. When a sequential job is scheduled, then except for the last one on each level the whole subtree of jobs dependent on it is assigned time 0. Thus the resulting schedule has the same length as in the fully online case and the lower bound holds as well. □

6 Scheduling on PRAMs

Let $T_{\max}(\mathcal{F})$ be the maximal sum of running times of jobs along any path in a dependency graph $\mathcal{F}$. Clearly $T_{\max} \leq T_{\text{opt}}(\mathcal{F})$.

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 each $\alpha \leq 1$, let $T_{<\alpha}(S, \mathcal{F})$ be the total time during which the efficiency of S is less than α.

The next lemma has proven to be very useful for analyzing scheduling algorithms [5]. The lemma is valid also for job systems with dependencies.

Lemma 6.1 *Let S be a schedule for a parallel job system $\mathcal{F}$ such that the work of each job is preserved. If $T_{<\alpha}(S, \mathcal{F}) \leq \beta T_{\text{opt}}(\mathcal{F})$, where $\alpha \leq 1$ and $\beta \geq 0$, then $T_S(\mathcal{F}) \leq (\frac{1}{\alpha} + \beta) T_{\text{opt}}(\mathcal{F})$.*

6.1 Scheduling on PRAMs without virtualization

We start by giving a generic algorithm for online scheduling of parallel job systems with dependencies. By "generic", we mean that the scheme does not directly depend on the underlying network topology. Although not highly competitive in general, it is the best possible algorithm for PRAMs without virtualization.

This algorithm maintains a queue $\mathcal{Q}$ that contains all jobs available at the current time. All jobs that become available are 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 available job in $\mathcal{Q}$, although in practice we suggest

selecting an available job based on best fit principle.

```
algorithm GENERIC
repeat
   if the resource requirements of First(Q) can be satisfied
      then begin
         schedule First(Q) on a requested subgraph;
         Q ← Q − First(Q);
      end;
until all jobs are finished.
```

Let T_{empty} be the total time when $\mathcal{Q}$ is empty. The following lemma gives an upper bound on T_{empty}.

Lemma 6.2 *For all online parallel job systems with dependencies $\mathcal{F}$, $T_{\text{empty}} \leq T_{\max}(\mathcal{F}) \leq T_{\text{opt}}(\mathcal{F})$.*

Proof. Consider the schedule generated by GENERIC. Let J_0 be the job that finishes last. Let I_1 be the last time interval before J_0 is started for which $\mathcal{Q}$ is empty. Then there is a job J_1 running at the end of I_1 that is an ancestor of J_0 in the dependency graph, otherwise J_0 would be available already during I_1 and $\mathcal{Q}$ would not be empty. By the same method construct I_2, J_2, I_3, J_3, ..., I_k, J_k, until there is no interval for which $\mathcal{Q}$ is empty 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. Moreover all time intervals for which $\mathcal{Q}$ is empty are covered by the running times of these selected jobs, and hence $T_{\text{empty}} \leq T_{\max}(\mathcal{F})$. □

Now we show that the competitive ratio of GENERIC matches the lower bound from Section 5.

Theorem 6.3 *Suppose that the largest number of processors requested by a job is λN, where $0 < \lambda < 1$. Then GENERIC 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. It follows that $T_{<(1-\lambda)} \leq T_{\text{empty}} \leq T_{\text{opt}}$. Lemma 6.1 finishes the proof. □

If all jobs are sequential, we can prove an exact bound in terms of N. For sequential jobs the network topology is irrelevant, and thus this bound applies to an arbitrary network topology. This bound matches previous results [7].

Theorem 6.4 *Suppose that each job in the parallel job system is a sequential job. Then GENERIC is $(2-\frac{1}{N})$-competitive for an arbitrary network topology.*

Proof. Whenever $\mathcal{Q} \neq \emptyset$, the efficiency is 1. Let T be the time when the efficiency is 1, let $T' = T_{<1}$ be the remaining time. By Lemma 6.2, $T' \leq T_{\text{empty}} \leq T_{\text{opt}}$. As in Lemma 6.1, we have $T + \frac{1}{N}T' \leq T_{\text{opt}}$. Combining these two inequalities we obtain $T + T' \leq (1 + \frac{N-1}{N})T_{\text{opt}} = (2 - \frac{1}{N})T_{\text{opt}}$. □

6.2 Scheduling on PRAMs with virtualization

The main result of this section is an optimal $(2 + \phi)$-competitive algorithm, where $\phi = \frac{\sqrt{5}-1}{2} \approx 0.618$ is the golden ratio. We also study the problem where each job requests at most λN processors, where $0 < \lambda < 1$. In this case the competitive ratio can be improved. We give tight lower and upper bounds.

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

algorithm PRAM(α), ($0 < \alpha \leq 1$)
repeat
  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;
      $\mathcal{Q} \leftarrow \mathcal{Q} -$ First($\mathcal{Q}$);
    end;
until all jobs are finished.

Theorem 6.5 *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 $\alpha = \frac{1}{2} - \lambda + \sqrt{\lambda^2 + \frac{1}{4}}$.*

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 6.6 *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 6.5. First, observe that when $\mathcal{Q}$ is nonempty, the efficiency is at least α, 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 a factor at most $\frac{\lambda}{1-\alpha}$. Therefore we obtain as in Lemma 6.2 $\frac{1-\alpha}{\lambda}T' + T'' \leq T_{\max} \leq T_{\text{opt}}$. Because the efficiency of the optimal schedule cannot be greater than one, we have $\alpha T + (1-\alpha)T' \leq T_{\text{opt}}$. By adding the first inequality and the last one divided by α, we obtain $T + (1-\alpha)(\frac{1}{\alpha} + \frac{1}{\lambda})T' + T'' \leq T_{\text{opt}}$. Since $(1-\alpha)(\frac{1}{\alpha} + \frac{1}{\lambda}) = 1$, the competitive ratio is $\frac{T+T'+T''}{T_{\text{opt}}} \leq 1 + \frac{1}{\alpha} = \sigma$. □

We prove a matching lower bound. The proof is given only the fully online algorithms, it can be generalized to all online algorithms by the same method as in Theorem 5.2.

Theorem 6.7 *Suppose that the largest number of processors requested by a job is λN, where N is the number of processors of the machine and $0 < \lambda \leq 1$. Then no online scheduling algorithm achieves a better competitive ratio than $\sigma = 2 + \frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$ for all N.*

Proof. The proof is divided into two cases, $\lambda = 1$ and $\lambda < 1$.

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

The job system used by the adversary is similar to that in Theorem 5.2. It 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. As in Theorem 5.2 the adversary can force that the parallel job always depends on the sequential job that is scheduled last at that level.

Now we describe the adversary strategy. 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 sufficiently long time to this job and removes all other jobs. Therefore the competitive ratio is larger than $\frac{1}{1-\phi} = 2+\phi$ 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 $T' = \phi lT$. 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 ϕlT 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 by α which decreases each iteration, 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. Phase i has l levels, each level has $\lfloor \alpha_i N \rfloor + 2$ sequential jobs and one parallel job of size $\lfloor \lambda N \rfloor$. The dependencies are as before, each parallel job depends on one sequential job on the same level and all sequential jobs depend on the parallel job from the previous level. Also the adversary 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 is at most $A_i + \frac{1}{N}$ if i phases have been scheduled. 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. 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 $\frac{\lambda}{1-\alpha_1} = \sigma$, so the adversary just runs J long enough. For $i > 1$ let $\bar{T}$ be the time when J is scheduled, 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 yet 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 of time T'. 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 $\frac{\bar{T}+\frac{\lambda}{1-\alpha_i}T'}{T'} \geq \sigma$.

So if we chose σ such that $\sigma = 1 + \frac{1}{L}$, then no 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 6.5 and therefore $\sigma = 2 + \frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$ is the solution of the equation. □

7 Scheduling on hypercubes and meshes

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 coordinates except the first k 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 between $d - i \log h$ 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
repeat
    for all i such that Q_i ≠ ∅ do
        if a normal (d − i log h)-dimensional subcube in M_i is available
            then begin
                schedule First(Q_i) on that subcube;
                Q_i ← Q_i − First(Q_i);
            end;
until all jobs are finished.
```

Theorem 7.1 *The competitive ratio of HYPERCUBE is* $O(\frac{d}{\log d}) = O(\frac{\log N}{\log \log N})$ *for the* 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 no job is slowed down by a factor greater than h and as in Lemma 6.2 we obtain $T_{\text{empty}} \leq hT_{\max}$ (T_{empty} now refers to the time when all queues are empty). Second, if $\mathcal{Q}_i$ is nonempty then the efficiency of the subcube M_i is equal to 1, and the overall efficiency is

at least $\frac{1}{h}$. Thus, by Lemma 6.1, 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 $\mathcal{J}_i$, $i = (i_1, \ldots, i_d)$, $1 \le i_j \le k$. A job belongs to $\mathcal{J}_i$ if it requests a submesh of size $(a_1, a_2, ..., a_d)$ such that $\frac{n}{k^{i_j}} < a_j \le \frac{n}{k^{i_j-1}}$. $\mathcal{Q}_i$ is a queue for the available jobs from $\mathcal{J}_i$. The mesh is partitioned into h submeshes M_i of size $\lfloor\frac{n}{k}\rfloor \times \cdots \times \lfloor\frac{n}{k}\rfloor$. Jobs from $\mathcal{J}_i$ are scheduled on submesh M_i only.

algorithm MESH
repeat
  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;
until all jobs are finished.

The proof of following theorem is similar to that of Theorem 7.1 and omitted.

Theorem 7.2 *MESH is $O((\frac{\log N}{\log\log N})^d)$-competitive for a d-dimensional mesh of $N = n^d$ processors.*

7.1 Lower bounds

In this section, we prove that the competitive ratios of our algorithms for one dimensional meshes and hypercubes are within a constant factor of the optimal competitive ratio.

Our approach to these lower bounds is similar to [5]. 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, and therefore we are able to achieve larger lower bounds than without them.

Theorem 7.3 *No online scheduling algorithm can achieve a better competitive ratio than* $\Omega(\frac{\log N}{\log\log N})$ *for a one-dimensional mesh of* N *processors.*

Proof. Put $s = 2\lceil \log N \rceil$ and $t = \lceil \frac{1}{2} \log_s N \rceil = \Theta(\frac{\log N}{\log\log N})$.

Our job system has $t^2 N$ independent chains, each consisting of t^2 jobs, a total of $t^4 N$ jobs. In each chain the sizes of jobs are $1, s^2, s^4, \ldots, s^{2t-2}$, repeated s times, i.e., the $jt+i$-th job in each chain has size s^{2i-2}. Notice that s^{2t-2} is approximately n.

During the process 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 position 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 is equal to T and there was no SINGLE JOB step, the adversary ends the current phase, i.e., he removes all running jobs and 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 $\frac{3N}{t}$ 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}, thus ending the current subphase.

Evaluation of the adversary strategy

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

Our analysis is based on the following lemma.

Lemma 7.4 *In the i-th subphase of any phase the following statements are true.*

(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 $\frac{in}{t}$.

Proof. We prove this inductively. At the beginning of the whole process (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.

Now we consider the normal s^{2i-1}-segments during the i-th subphase. In the previous DECREASE EFFICIENCY step 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 so their total area 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 $\frac{3N}{t} - \frac{N}{s} \geq \frac{2N}{t}$ processors to induce the next DECREASE EFFICIENCY step. Scheduled jobs have length s^{2i} and therefore (to avoid SINGLE JOB) at least half of their area consist of whole normal s^{2i-1}-segments. These segments could not be used at the beginning of the subphase, so during the subphase the total area of used normal s^{2i-1}-segments increases by at least $\frac{N}{t}$ and at the end of the subphase it is at least $\frac{i+1}{t}$. The area 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 so the area does not decrease. This proves that (ii) is true at the beginning of the next subphase.

From (ii) it follows that at the end of the $(t-1)$-th subphase all normal s^{2t-3}-segments are used and so no available job can be scheduled. Therefore a CLEAN UP step ends every phase and both (i) and (ii) are also true at the beginning of the next phase. □

From the last paragraph of the proof of the lemma it also follows that every phase takes time T. Every DECREASE EFFICIENCY or CLEAN UP step removes at most n chains. It follows that there are sufficiently many chains available for t phases. Therefore the whole schedule takes at least time tT long.

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 [5]. The efficiency of the online schedule was at most $\frac{3}{t}$ all time, therefore there is an offline schedule $O(t)$-times shorter. □

Theorem 7.5 *No online scheduling algorithm for a d-dimensional hypercube of $N = 2^d$ processors can achieve a better competitive ratio than $\Omega(\frac{d}{\log d}) = \Omega(\frac{\log N}{\log \log N})$.*

Proof. The proof is similar to the previous one provided that normal segments are replaced by subcubes with $s = 2^k$ processors. Thus we have to choose s to be a power of 2. It is easy to verify that this changes the bound only by a constant factor. □

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 that important. A main open question is whether randomization helps to improve the performance.

Acknowledgement

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

References

[1] S.N. Bhatt, F.R.K. Chung, J.-W.Hong, F.T. Leighton, and A.L. Rosenberg. Optimal simulations by butterfly networks. In *STOC*, pages 192–204. ACM, 1988.

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

[3] E. Davis and J.M. Jaffe. Algorithm for scheduling tasks in unrelated processors. *JACM*, pages 712–736, 1981.

[4] F.G. Feitelson and L. Rudolph. Wasted resources in gang scheduling. In *Proc. of the 5th Jerusalem Conference on Information Technology*, 1990.

[5] A. Feldmann, J. Sgall, and S.-H. Teng. Dynamic scheduling on parallel machines. In *FOCS*, pages 111–120, IEEE, 1991.

[6] M.R. Garey and D.S. Johnson. *Computer and Intractability: a guide to the theory of NP-completeness.* Freeman, San Francisco, 1979.

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

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

[9] S.R. Kosaraju and M.J. Atallah. Optimal simulation between mesh-connected arrays of processors. In *STOC*, pages 264–272. ACM, 1986.

[10] H.T. Kung. Computational Models for Parallel Computers. Technical report, CMU-CS-88-164, 1987.

[11] J.L. Peterson and A. Silberschatz. *Operating System Concepts.* Addison-Wesley Publishing Company, 1985.

[12] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *CACM*, 28(2):202–208, 1985.

[13] D.B. Shmoys, J. Wein, and D.P. Williamson. Scheduling parallel machines on-line. In *FOCS*, pages 131–140, 1991, IEEE.

[14] Q. Wang and K.H. Cheng A Heuristic of Scheduling Parallel Tasks and its Analysis. *SIAM Journal on Computing.*, pages 281–294, 1992.