Dynamic scheduling on parallel machines[*]

Anja Feldmann[†] Jiří Sgall [‡] Shang-Hua Teng [§]

School of Computer Science

Carnegie Mellon University

5000 Forbes Avenue

Pittsburgh, PA 15213-3890

[*]A preliminary version of this paper appeared in the 32nd Annual Symposium on Foundations of Computer Science, 1991 [5].

[†]This work was supported in part by fellowship of the Studienstiftung des deutschen Volkes.

[‡]On leave from Mathematical Institute, AV ČR, Žitná 25, 115 67 Praha 1, Czech Republic.

[§]That work was supported in part by National Science Foundation grand DCR-8713489. Revision work on this paper was done while the author was at Xerox Palo Alto Research Center. Current Address: Department of Mathematics, MIT, Cambridge, MA 02139.

Abstract

We study the problem of online job-scheduling on parallel machines with different network topologies. An online scheduling algorithm schedules a collection of parallel jobs with known resource requirements but unknown running times on a parallel machine.

We give an $O(\sqrt{\log\log N})$-competitive algorithm for online scheduling on a two-dimensional mesh of N processors and we prove a matching lower bound of $\Omega(\sqrt{\log\log N})$ on the competitive ratio. Furthermore, we show tight constant bounds of 2 for PRAMs and hypercubes, and present a 2.5-competitive algorithm for lines. We also generalize our two-dimensional mesh result to higher dimensions. Surprisingly, our algorithms become less and less greedy as the geometric structure of the network topology becomes more complicated. The proof of our lower bound for the two-dimensional mesh actually shows that no greedy-like algorithm can perform well.

1 Introduction

A sequence of jobs is scheduled on a parallel machine with a specific network topology. Each job arrives with known resource requirements but an unknown running time requirement (its running time is determined dynamically). The operating system on a such parallel machine has to assign a virtual machine to each job [13] so that each job is oblivious to the presence of other jobs that may be running simultaneously on the same hardware. The job scheduling problem is to design an online scheduling algorithm that minimizes the total time required to process all jobs.

This problem is quite general and quite complex; no good solution is yet known. As an approximation, we assume that all jobs together with their resources requirements are given at the beginning of the scheduling process. But their running times are not given, and can only be determined by actually processing the jobs.

The resource requirement of each job is characterized by the maximal degree of parallelism it can efficiently use. (In the case of a mesh, for example, we might know that a job will run efficiently on a 100 by 100 mesh, but not on a larger one.) In our model, scheduling is non-preemptive and without restarts, i.e., once the job is started, it cannot be moved or stopped for later resumption or restart. The objective of the scheduling algorithm is to find a schedule of the jobs on a parallel machine with a specific geometric network topology that minimizes the total amount of time (the makespan).

The measure we use to evaluate performance of a scheduling algorithm is the competitive ratio: the worst-case ratio of the algorithm's performance to that of an optimal offline algorithm that knows all the running times in advance [16].

As far as we know, this paper is the first study of online scheduling on parallel machines which considers the specifics of the underlying geometric structure of the network topology. It is a first step in a new area of research, and opens up many interesting new problems.

In contrast to previous work [2, 3, 7, 12, 15], we study scheduling on a number of parallel machines. The introduction of concrete network topologies captures real parallel systems (such as iWarp and CM2). Currently, these machines are usually used by a single job or divided into a few fixed partitions each capable of running a single job. To use parallel machines in a more efficient and flexible way, we need to study such general scheduling problems.

By the technique of [15] we can weaken the requirement that all jobs are given at the beginning of the scheduling process. We can modify our algorithms by this technique so that they achieve similar competitive ratios even in the case when each job is released at some fixed time independent of the schedule and its resource requirements are known at that time. However this still does not capture the most interesting case where there are dependencies among jobs. For a study of this more general problem see [4].

The underlying geometric structure of the network topology makes the online scheduling problem difficult. All our algorithms make use of particular aspects of the topology of the parallel machine in question. This especially applies to the case of a mesh machine. In the proof of the lower bound we show that it is always possible to find a small set of running jobs that restricts any efficient scheduling of other jobs. This is based on a geometric observation stated in lemma 8.1 which is interesting on its own.

It is worth noting that the more complicated the underlying geometric structure becomes, the less greedy our optimal algorithms are. While the optimal algorithm for the PRAM is greedy, our optimal algorithm for the mesh is not greedy at all; at each time it only uses a small portion of the mesh efficiently. The proof of the lower bound actually shows that this is not an arbitrary choice—no greedy-like algorithm can achieve a substantially better competitive ratio than $\Omega(\log\log N)$. This shows that the general heuristic of using greedy algorithms for scheduling can be misguided despite the fact that it works well in many previous scheduling algorithms [2, 3, 7, 15].

Efficient parallel algorithms have been developed for a wide variety of problems and for various parallel machines [8]. Mesh-based parallel machines are widely used for solving large-scale problems in scientific computing, image processing, and a number of other important areas [10]. In practice, meshes are used by only one process at a time also because no good dynamic scheduling algorithm for meshes was known. Our results demonstrate that mesh-based parallel machines could work efficiently on multiple tasks at the same time (space sharing).

In Sections 2 and 3 we give the definitions and state the main results. In Sections 4, 5 and 6 we present algorithms with constant competitive ratios for PRAMs, hypercubes and lines of processors (one-dimensional meshes). In Section 7 we give an $O(\sqrt{\log\log N})$-competitive algorithm for two-dimensional meshes. This algorithm is within a constant factor of the best possible, which is proved in Section 8. In Section 9 we generalize the previous result to higher dimensions.

2 Definitions and preliminaries

In this section we formally define the dynamic scheduling problem on parallel machines and introduce some notation. We then give a useful lemma for analyzing scheduling algorithms.

2.1 Parallel machines and parallel jobs

A parallel machine with a specific network topology can be viewed as a graph where each node u represents a processor p_u and each edge (u, v) represents the communication link between p_u and p_v.

To describe the amount of resources parallel jobs require, let $\mathcal{G}$ be a set of graphs (e.g., a set of 2-dimensional meshes), which are called *job-types*. In general, $\mathcal{G}$ is a set of structured subgraphs of the network topology which always contains both the singleton graph and the machine itself. A *$\mathcal{G}$-job* or *parallel job J* is characterized by a pair $J = (g, t)$ where $g \in \mathcal{G}$ and t is the running time of J on a parallel machine with network topology g. The *work* needed for J is $|g|t$, where $|g|$ denotes the size of the graph g, i.e., the number of processors required by J. It is assumed that g correctly reflects the inherent parallelism.

A *parallel job system* $\mathcal{J}$ is a collection of parallel jobs ($\mathcal{G}$-jobs).

2.2 Simulation among parallel machines

During the execution of a parallel job system the next available job may require p processors while there are only $p' < p$ processors available in the system.

In general, a job requiring machine g and running time t can be simulated by an machine g' in time $\alpha(g, g')t$, where $\alpha(g, g') \geq 1$ is the *simulation factor* [1, 9]. We assume without loss of generality that during simulation the amount of work is at least preserved. For example, the computation on an r-dimensional hypercube might be simulated on an $r' < r$-dimensional hypercube by increasing the running time by a factor of at least $2^{r-r'}$.

In fact, none of our algorithms uses simulation. However, our lower bounds remain valid even for algorithms using simulation.

2.3 The scheduling problem

The scheduling problem has three components $\mathcal{A}$, $\mathcal{G}$, $\mathcal{R}$, where $\mathcal{A}$ is the network topology of the given parallel machine, $\mathcal{G}$ is the set of the job-types, and $\mathcal{R}$ are the simulation assumptions stating all of the simulation factors.

Given a parallel job system $\mathcal{J}$ of $\mathcal{G}$-jobs $J_i = (g_i, t_i)$, a *schedule* of $\mathcal{J}$ on a parallel machine with network topology $\mathcal{A}$ is an assignment of a subgraph g'_i of $\mathcal{A}$ and a time interval to each job t'_i in $\mathcal{J}$ consistent with the job-type g_i and the simulation assumption $\alpha(g_i, g'_i)$ such that at any given time no two subgraphs assigned to different running jobs are overlapping.

A scheduling algorithm is *offline* if it receives the running time of each job as part of its input. It is *online* if it does not get any knowledge of the running time of the jobs in advance.

It is NP-hard to find the optimal schedule offline, even for a very restricted variations of the scheduling problem [6]. But in most cases it is easy to find a solution within some constant factor of the optimal by using similar heuristics as for bin packing. We do this for the case of two-dimensional mesh in Section 8.5.

We use the model of non-preemptive scheduling without restarts because it is closest to its practical counterpart. The model with restart gives better scheduling algorithms in theory but is an oversimplification of the practical scheduling problem. In practice there are other significant costs and software complications associated with restart, such as reloading and reallocation of buffers and communication resources.

2.4 The complexity measure

The *length* of a schedule generated by algorithm S, denoted by $T_S(\mathcal{J})$, is the time S needs to process all the jobs in $\mathcal{J}$ using the simulation assumptions in $\mathcal{R}$. Let $T_{opt}(\mathcal{J})$ denote the length of an *optimal* offline schedule.

The complexity measure we are using is the competitive ratio [16]. A scheduling algorithm S is *σ-competitive* if $T_S(\mathcal{J}) \leq \sigma T_{opt}(\mathcal{J})$ for every $\mathcal{J}$. Note that deciding whether a given schedule is optimal is NP-complete [6].

An online scheduling problem can be viewed as a game between the scheduling algorithm and an adversary who dynamically determines the running time of each job. The scheduling algorithm tries to schedule jobs to subgraphs of the parallel machine and the adversary decides when to remove jobs from it, namely, it decides when a job finishes. For technical simplicity, we assume without loss of generality that both the removal of a job from a subgraph and the scheduling of a job onto a subgraph take no time. The goal of the adversary is to maximize the competitive ratio of the online algorithm while the goal of the scheduling algorithm is to minimize it.

2.5 Basic properties

A processor is *busy* at time t if it belongs to a subgraph on which a job is scheduled. On an N processor machine define $T_{\mathit{eff}}(\mathcal{J}) = (\sum_i |g_i|t_i)/N$, i.e., the time required to schedule $\mathcal{J}$ if every processor was busy for the entire length of the schedule. Let $t_{max}(\mathcal{J}) = \max_i t_i$ be the maximal running time of a single job. Then define $T_{min}(\mathcal{J}) = \max(T_{\mathit{eff}}(\mathcal{J}), t_{max}(\mathcal{J}))$. It is easy to see that $T_{min}(\mathcal{J}) \leq T_{opt}(\mathcal{J})$ for all $\mathcal{J}$. Using a simulation assumption does not influence this fact, because it can only increase the amount of work or the running time of any job.

The *efficiency* of a schedule at time t is the number of busy processors divided by N. Similarly, we can define the efficiency with respect to any subgraph and the efficiency of a currently running job J or a set $\mathcal{C}$ of currently running jobs, denoted by $\mathrm{eff}(J)$ and $\mathrm{eff}(\mathcal{C})$, respectively. Finally, for $\alpha \leq 1$ let $T_{<\alpha}(S, J)$ be the total time during which the efficiency of S is less than α. The following is a basic property of scheduling.

Lemma 2.1 *Let $\alpha \leq 1$, $\beta \geq 0$ and S be a schedule for a set $\mathcal{J}$ of jobs that uses no simulation assumption. If $T_{<\alpha}(S, \mathcal{J}) \leq \beta T_{opt}(\mathcal{J})$ then $T_S(\mathcal{J}) \leq (\frac{1}{\alpha} + \beta)T_{opt}(\mathcal{J})$.*

Proof: Because S uses no simulation assumption, the optimal algorithm has to do at least the same amount of work as S. The work done by S is at least $\alpha(T_S(\mathcal{J}) - T_{<\alpha}(S, \mathcal{J}))$, hence $T_{opt}(\mathcal{J}) \geq \alpha(T_S(\mathcal{J}) - T_{<\alpha}(S, \mathcal{J}))$ and $T_S(\mathcal{J}) \leq \frac{1}{\alpha}T_{opt}(\mathcal{J}) + T_{<\alpha}(S, \mathcal{J})) \leq (\frac{1}{\alpha} + \beta)T_{opt}(\mathcal{J})$. □

2.6 Some network topologies

In this section we introduce the network topologies of some specific parallel machines such as PRAM, line, meshes and hypercubes.

PRAM: For the purpose of online scheduling, a parallel random access machine of N processors can be viewed as a parallel system whose underlying network topology is a complete graph of N vertices (an N-clique). Therefore, each subset of processors forms a subsystem which is also a PRAM. Each job J is characterized by a pair (p, t) where p denotes the number of processors that J requires and t is the running time of J when using p processors. The simulation assumption for a PRAM is that a job $J = (p, t)$ can be processed on $p' \leq p$ processors in $(\frac{p}{p'})t$ time.

Hypercube: An m-dimensional hypercube has $N = 2^m$ processors where two processors are directly connected if their binary representations differ by exactly one bit. Available job-types include all d-dimensional subcubes for $d \leq m$. Each job J in the hypercube is characterized by a pair (d, t) where d denotes the dimension of the subcube required by J and t is the running time of J on a d-dimensional hypercube. The simulation assumption is that each job $J = (r, t)$ can be processed on an r'-dimensional hypercube, $r' \leq r$, in $2^{r-r'}t$ time.

Line: A line (one-dimensional mesh) of N processors is a parallel system consisting of N processors $\{p_i | 0 \leq i < N\}$ where processor p_i is directly connected with processors p_{i+1} and p_{i-1} (if they exist). Each 1-*dimensional job* J is characterized by a pair (a, t), where a is the number of connected processors that J requires and t is the running time. The simulation assumption for a line is that each job $J = (a, t)$ can be processed on an $a' \leq a$ processor line in time $\frac{a}{a'}t$.

Mesh: A two-dimensional mesh $n_1 \times n_2$ of N processors is a parallel system consisting of N processors $\{p_{i,j} | 0 \leq i < n_1, 0 \leq j < n_2\}$ where processor $p_{i,j}$ is directly connected with processors $p_{i,j\pm1}, p_{i\pm1,j}$ (if they exist). Each 2-*dimensional job* is characterized by a triple (a, b, t), meaning J requires an $(a \times b)$-submesh and t is its running time. Since an $a \times b$ mesh is isomorphic to a $b \times a$ mesh, we assume without loss of generality that $a \geq b$. The mesh simulation assumption is that each job (a, b, t) can be processed on a $a' \times b'$ mesh in time $(t \max(1, \frac{a}{a'}) \max(1, \frac{b}{b'}))$.
For $d \geq 3$, d-dimensional meshes and d-dimensional jobs are defined similarly.

3 Overview of our results

Our main results are matching upper and lower bounds for scheduling on a two-dimensional mesh. We also give tight bounds for PRAMs and hypercubes and a 2.5

competitive algorithm for the one-dimensional mesh and show how to generalize the two-dimensional mesh algorithm to the d-dimensional case.

An important feature of all our algorithms is that they do not make use of the simulation assumption while the offline algorithm is allowed use it. This changes dramatically when precedence constraints are introduced; there the simulation assumption is essential for any efficient scheduling algorithm, see [4] for details.

The natural approach to scheduling is a *greedy* method, i.e., a method that schedules a job whenever there are some processors available. An interesting result is that such greedy methods only work for simple network topologies; as network topologies get more complex, greedy methods perform much worse than optimal algorithms because they tend to scatter the available processors so that they become unusable for larger jobs.

For PRAMs, any greedy method is optimal. For one-dimensional meshes and hypercubes we already need to modify the approach; only the largest job is scheduled whenever processors are available, smaller jobs are postponed. Finally for the two-dimensional mesh we have to abandon the greedy approach completely; the optimal algorithm begins by using only a fraction of the whole mesh and continues with a larger fraction only when the current fraction becomes to small. The lower bound proof actually shows that a worse competitive ratio is unavoidable, using any greedy approach will square the competitive ratio.

Theorem 3.1 *The following table summarizes our results with N being the number of processors.*

Network topology	*Upper bound*	*Lower bound*
two-dimensional mesh	$O(\sqrt{\log\log N})$	$\Omega(\sqrt{\log\log N})$
PRAM	$2-\frac{1}{N}$	$2-\frac{1}{N}$
hypercube	$2-\frac{1}{N}$	$2-\frac{1}{N}$
one-dimensional mesh	2.5	$2-\frac{1}{N}$
d-dimensional mesh	$O(2^d d\log d\sqrt{\log\log N}+2^d(d\log d)^d)$	$\Omega(\sqrt{\log\log N})$

Our lower bound for mesh relies on the two-dimensional structure of jobs. This structure is preserved if we use no simulation assumption or the simulation assumption introduced in Section 2, i.e., if we only allow to simulate a mesh by a mesh which is smaller or equal in both dimensions; this simulation has no overhead and therefore preserves the work of each job.

If we allow overhead, it is possible to simulate every mesh by a square mesh of the same area while increasing the work at most by a constant factor. Using the same ideas as for scheduling on a line, it is possible to develop an algorithm for scheduling square meshes with a constant competitive ratio. Our lower bound shows that this more complicated and less efficient simulation is a necessary if we want to improve the competitive ratio.

4 The PRAM

The PRAM is one of the most common models for the design of parallel algorithms [14, 8]. In this section we present the lower and upper bounds on the competitive ratio of online scheduling on PRAMs.

The following example (due to Shmoys, Wein and Williamson [15]) gives us a lower bound of $2 - \frac{1}{N}$ for any competitive algorithms on any parallel machine of N processors and for any simulation assumption, because it only uses sequential jobs (jobs requiring only one processor). Take a job system of $N(N-1)+1$ sequential jobs. The adversary assigns time 1 to all jobs except to the job that has been started last, to which it assigns time N. Then the length of the schedule is at least $2N-1$. Because the optimal schedule takes time N, which is easy to verify, the competitive ratio is at least $2 - \frac{1}{N}$.

We now prove that the natural greedy approach below leads to an optimal $2 - \frac{1}{N}$ competitive algorithm.

ALGORITHM: GREEDY
while there is an unscheduled job J_i **do**
  if J_i requires p processors and p processors are available,
    then schedule J_i on the p processors.

Theorem 4.1 GREEDY *is* $(2 - \frac{1}{N})$*-competitive for a PRAM with* N *processors.*

Proof: Suppose that the algorithm generates a schedule of length T for a job system $\mathcal{J}$. Let p be the minimal number of busy processors during the whole schedule. Consider the last time τ when only p processors were busy. Let J be some job running at that time and suppose it takes time t. Before J is scheduled, there could not have been p processors available - otherwise J would be scheduled earlier. After J is finished, there also cannot be p processors available: at any time after τ there has to be some job J' running that was scheduled after τ, as the efficiency is no longer minimal; and if there were p processors available, J' would only require $n - p$ processors and it would already have been scheduled before τ, a contradiction.

So for the time $T - t$ when J is not running, the efficiency is at least $\frac{N-p+1}{N}$, and it is at least $\frac{p}{N}$ all time. If $t \geq \frac{N}{2N-1}T$ then $T_{opt} \geq t_{max} \geq t \geq \frac{N}{2N-1}T$ and the competitive ratio is at most $\frac{2N-1}{N} = 2 - \frac{1}{N}$. If $t \leq \frac{N}{2N-1}T$ then $T_{\mathit{eff}} = \frac{N-p+1}{N}(T-t) + \frac{p}{N}t$. It is easily checked that this value is smallest for $p = 1$ and $t = \frac{N}{2N-1}T$, giving us $T_{opt} \geq T_{\mathit{eff}} \geq \frac{N}{2N-1}T$ and the competitive ratio is at most $2 - \frac{1}{N}$. □

5 The hypercube

From a theoretical viewpoint, hypercubes have some very nice properties. Furthermore, there exist parallel computers which use a hypercube network topology. In this section we present optimal algorithms for dynamic scheduling on hypercubes.

The algorithm is not completely greedy any more. It first sorts the jobs according to decreasing resource requirements and then schedules them in this order. We suppose that the jobs $J_i = (d_i, t_i)$ are sorted by size, $d_1 \geq d_2 \geq \ldots \geq d_m$, where m is the number of jobs.

We say that a d-dimensional subcube is normal if all its processors only differ in last d bits. This implies that if two normal subcubes intersect then one of them is a subcube of the other one. To ensure that the space is used efficiently, the jobs are only scheduled on normal subcubes.

ALGORITHM: ORDERED
for i = 0 to m **do**

if there is a normal d_i subcube available,
then schedule the job J_i on it

Theorem 5.1 ORDERED *is* $(2 - \frac{1}{N})$*-competitive for a hypercube of* N *processors.*

Proof: From the properties of normal subcubes and the scheduling algorithm it follows that whenever there is any processor available, then there is a whole normal d-dimensional subcube available, where d is the dimension of the job scheduled last. Therefore any waiting job can be scheduled, as they are sorted according to decreasing dimension. This proves that the efficiency is 1 as long as there is some unscheduled job left. The rest of the proof is same as in Theorem 4.1. □

6 The line

In this section we present two algorithms for scheduling on a line of N processors (a one-dimensional mesh). To satisfy the structure of the resource requirements on PRAMs we could use any processors. Hypercubes are nicely divisible into subcubes with the same properties so that choosing the right subcube is again easy. This is not the case in the mesh models, as we do not require the number of processors nor the job-types to be powers of 2. This is one of the reasons for all the complications in the mesh models.

Our first algorithm uses the same strategy as the algorithm for hypercubes, but for line this is only 3-competitive. The second algorithm we give is more complex and has a 2.5-competitive ratio. The best lower bound we know is the general $2 - \frac{1}{N}$ one which still leaves a gap between the bounds.

These algorithms are used as building blocks for the two-dimensional mesh algorithms. The first algorithm is sufficient for that purpose, as we only worry about the asymptotic growth rate but we present the 2.5-competitive algorithm because it is interesting in its own right.

For both algorithms we again suppose that the jobs $J_i = (a_i, t_i)$ are sorted by size, $a_1 \geq a_2 \geq \ldots \geq a_m$, where m is the number of jobs. During the schedule, let $\mathcal{C}$ denote the set of currently running jobs and $F(\mathcal{C})$ denote the set of remaining intervals induced by $\mathcal{C}$.

The first algorithm is still somewhat greedy, with the restriction that jobs are scheduled in order of decreasing size. We call this algorithm ORDERED.

ALGORITHM: ORDERED
for i = 0 to m **do**
  if there is an interval I in $F(\mathcal{C})$, $I = [u \mathinner{..} v]$, such that $u + 1 - v \geq a_i$,
    then schedule the job J_i to the interval $[u \mathinner{..} u + a_i - 1]$.
    otherwise wait

Lemma 6.1 *Let S be the schedule generated by ORDERED for a job system $\mathcal{J}$, then as long as $i \leq m$, the efficiency of S is at least $\frac{1}{2}$. Therefore we have $T_{<\frac{1}{2}}(S, \mathcal{J}) \leq t_{max}(\mathcal{J})$.*

Proof: Let J_i be the largest unscheduled job. Then for all intervals $I \in \mathcal{C}$, $|I| \geq a_i$, while for all intervals $I' \in F(\mathcal{C})$, $|I'| \leq a_i$. Moreover, the number of intervals in $\mathcal{C}$ is equal to the number of intervals in $F(\mathcal{C})$. □

Consequently by Lemma 2.1, we have $T_S(\mathcal{J}) \leq (2+1)T_{opt}(\mathcal{J}) \leq 3T_{opt}(\mathcal{J})$, which proves the next theorem.

Theorem 6.2 ORDERED *is* 3-*competitive.* □

The second algorithm, an even less greedy one, is basically a refinement of the previous one. If we could achieve that there are always two adjacent jobs between any two unused intervals, the efficiency would be $\frac{2}{3}$ instead of previous $\frac{1}{2}$. It is impossible to achieve this arrangement all the time, but by placing the jobs carefully (keeping this idea in mind) we can maintain the efficiency at least $\frac{2}{3}$.

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

The jobs are divided into the set $\mathcal{J}'$ of jobs requiring at most $\frac{1}{3}$ of the processors and the set $\mathcal{J}'' = \mathcal{J} - \mathcal{J}'$. When we refer to the largest job in $\mathcal{J}'$ or $\mathcal{J}''$, we always mean the largest unscheduled job. A substantial part of the algorithm and the proof is concerned with large jobs, but we feel that this is not the main issue, and recommend you to concentrate on the other parts.

ALGORITHM: CLUSTERS

Phase 1

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

if there is a cluster I with efficiency less then $\frac{2}{3}$, **then**

1. **if** there is no left job in I **then**
   (a) **if** I is the leftmost cluster and there is some job available in $\mathcal{J}''$, schedule the largest job in $\mathcal{J}''$ as a left job in I,
   (b) **otherwise** schedule the largest job in $\mathcal{J}'$ as a left job in I;
2. **if** there is no right job in I, schedule the largest job in $\mathcal{J}'$ as a right job in I;
3. **if** there is no middle job in I, schedule the largest job in $\mathcal{J}'$ as a middle job in I, positioned so that either its left end is in the left third of the cluster or its right end is in the right third of the cluster;
4. **otherwise** schedule the largest job in $\mathcal{J}'$ adjacent to the middle job and divide I into two clusters with two jobs each.

otherwise wait

Phase 2

Considering the whole line as a single cluster,

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

5. **if** there is no left job, schedule the largest job in $\mathcal{J}''$ as a left job,
6. **if** there is no right job, all jobs from J' are finished and the efficiency is at most $\frac{1}{2}$, schedule the largest job in $\mathcal{J}''$ as a right job;

   otherwise wait

Theorem 6.3 CLUSTERS *is correct and* 2.5-*competitive.*

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

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

If there are both left and right jobs running in I and the efficiency is less then $\frac{2}{3}$, the space between the two has to be greater than $\frac{1}{3}$ of the length of I This is true because either the left or the right job has to be smaller than $\frac{1}{3}$ of the length of I, and the same argument works for the largest job in J' as it is at most as large as any running job. This justifies step 3.

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

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

The algorithm ensures that the efficiency is at least $\frac{2}{3}$ as long as there is some job available in J'. So in the case that the job which finishes last is from J', we get by Lemma 2.1 that the competitive ratio is at most $\frac{3}{2} + 1 = 2.5$.

Otherwise let T be the length of the whole schedule. We distinguish two cases depending on the efficiency at the time when the last job from $\mathcal{J}'$ finishes. If it is more than $\frac{1}{2}$, we know that the efficiency is at least $\frac{1}{2}$ until only the last job from $\mathcal{J}''$ is running - it is at least $\frac{2}{3}$ during Phase 1, it is at least $\frac{1}{2}$ before step 5. is invoked the first time and once step 6. is invoked it is again at least $\frac{2}{3}$, as two jobs from $\mathcal{J}''$ are scheduled simultaneously. So $T_{<\frac{1}{2}} \leq t_{max}$. If $T_{<\frac{1}{2}} \geq \frac{2}{5}T$, then $T_{opt} \geq t_{max} \geq \frac{2}{5}T$ and the competitive ratio is at most 2.5. Otherwise $T_{opt} \geq T_{\mathit{eff}} \geq \frac{1}{2}(T - T_{<\frac{1}{2}}) + \frac{1}{3}T_{<\frac{1}{2}} = \frac{1}{2}T - \frac{1}{6}T_{<\frac{1}{2}} \geq \frac{1}{2}T - \frac{1}{15}T > \frac{2}{5}T$ and the competitive ratio is again at most 2.5.

If the efficiency at the time when the last job from $\mathcal{J}'$ is finished, is at most $\frac{1}{2}$, then we know that the efficiency can only be less than $\frac{2}{3}$ at the beginning of Phase 2 just after the last job from $\mathcal{J}'$ is scheduled and at the end of Phase 2 when only one job is running. So $T_{<\frac{2}{3}} \leq 2t_{max}$, and the efficiency is always at least $\frac{1}{3}$. If $T_{<\frac{2}{3}} \geq \frac{4}{5}T$, then $t_{max} \geq \frac{2}{5}T$ and we are again done. Otherwise $T_{opt} \geq T_{\mathit{eff}} \geq \frac{2}{3}(T - T_{<\frac{2}{3}}) + \frac{1}{3}T_{<\frac{2}{3}} = \frac{2}{3}T - \frac{1}{3}T_{<\frac{2}{3}} \geq \frac{2}{3}T - \frac{4}{15}T = \frac{2}{5}T$ and the competitive ratio is at most 2.5. $\square$

7 The two-dimensional mesh

The mesh architecture is of particular interest not only because it is one of the most practical and widely used parallel architectures [10], but also the combinatorial problem behind it is very interesting and challenging.

Because of the more complex geometric structure of the two-dimensional mesh, the greedy approach does not work too well. A better strategy is to first partition the jobs according to one dimension, and then to use one of the algorithms for the line from the previous section. Jobs of different sizes are either scheduled in consecutive phases, or their schedules are combined to more complex schedules.

The next paragraph states some general methods to build scheduling algorithms which are then used to construct an $O(\log\log N)$-competitive algorithm (7.2) and an optimal $O(\sqrt{\log\log N})$-competitive algorithm (7.3). The optimal algorithm uses the suboptimal algorithm to schedule the large jobs which always seem to complicate the arguments.

7.1 Helpful techniques

We start with some useful definitions. Throughout this section we work with an $n_1 \times n_2$ mesh of N processors and assume that $n_1 \geq n_2$ without loss of generality.

Definition 7.1

1. *Let $\mathcal{J}_1, \ldots, \mathcal{J}_h$ be a partition of the set of d-dimensional jobs $\mathcal{J}$ and let $S_1, \ldots, S_h$ be schedules for $\mathcal{J}_1, \ldots, \mathcal{J}_h$, respectively. The* serial composition, $S = S_1 \odot S_2 \odot \cdots \odot S_h$, *denotes a schedule for $\mathcal{J}$ that first uses S_1 for $\mathcal{J}_1$, then (when S_1 finishes) it uses S_2 for $\mathcal{J}_2$, and so on until S_h finishes.*

2. *Let $\mathcal{J}$ be a set of two-dimensional jobs (a_i, b_i, t_i) with $a_i \leq \frac{n_1}{h}$ for all i. Let $\mathcal{J}_1, \ldots, \mathcal{J}_h$ be a partition of jobs in $\mathcal{J}$ and $U_1, \ldots, U_h$ be a partition of the $n_1 \times n_2$ mesh into h submeshes of size $\lfloor \frac{n_1}{h} \rfloor \times n_2$ submeshes (see Figure 1). Let S_j be a schedule that schedules jobs of $\mathcal{J}_j$ only on U_j. The* parallel composition, $S = S_1 \oplus S_2 \oplus \cdots \oplus S_h$, *denotes a schedule that simultaneously applies $S_1, \ldots, S_h$.*

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

4. *Define $\mathcal{J}' = \{(a, b, t) \in \mathcal{J} | a \leq \frac{n_1}{(\log n_1)^2}\}$ to be the set of small jobs.*

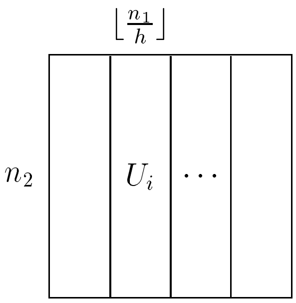

Figure 1: A partition of a $n_1 \times n_2$ mesh for the parallel composition

Now we describe the algorithm CLASS which is a simple modification of ORDERED to schedule one job class $\mathcal{J}^{(l)}$ on a two-dimensional $n_1 \times n_2$ mesh. Divide the mesh into 2^l submeshes of size $\frac{n_1}{2^l} \times n$ (more precisely, some of them will have size $\lfloor \frac{n_1}{2^l} \rfloor \times n_2$ and some $\lceil \frac{n_1}{2^l} \rceil \times n_2$) and then use ORDERED to schedule the jobs on these submeshes, viewing them as several one-dimensional meshes (disregarding the first dimension). It is easy to see that ORDERED can be applied to several one-dimensional meshes at once, without any change in performance. Compared to ORDERED, the efficiency of CLASS drops by at most a factor of two, because some jobs have a smaller first dimension than the submesh (but only up to a factor of two, as all of them are from $\mathcal{J}^{(l)}$). This leads to the following lemma.

Lemma 7.2 *Algorithm CLASS has an efficiency of at least $\frac{1}{4}$ as long as there is an unscheduled job. Therefore we have $T_{<\frac{1}{4}}(S, \mathcal{J}^{(l)}) \leq t_{max}(\mathcal{J}^{(l)})$.* □

The following lemma states a basic property of serial composition.

Lemma 7.3 (Serial Composition)
For all $\alpha \leq 1$, $T_{<\alpha}(S_1 \odot S_2 \odot \cdots \odot S_h, \mathcal{J}) = \sum_{i=1}^{h} T_{<\alpha}(S_i, \mathcal{J}_i)$. □

From this we can construct the following $O(\log N)$-competitive online scheduling algorithm SERIAL. First partition the jobs into the job classes $\mathcal{J}^{(l)}$ and generate a schedule S_l for each job class $\mathcal{J}^{(l)}$ using CLASS. Then apply serial composition to generate a complete schedule S from all subschedules S_l, $S = S_0 \odot S_1 \odot \cdots \odot S_{\lfloor \log n \rfloor}$.

Lemma 7.4 *For each set $\mathcal{J}$ of two-dimensional jobs* SERIAL *is* $(4 + order(\mathcal{J}))$*-competitive.*

Proof: By Lemma 7.2 we have $T_{<\frac{1}{4}}(S_l, \mathcal{J}^{(l)}) \leq t_{max}(\mathcal{J}^{(l)})$. By Lemma 7.3 we get $T_{<\frac{1}{4}}(S, \mathcal{J}) = \sum T_{<\frac{1}{4}}(S_l, \mathcal{J}^{(l)})$, and thus $T_{<\frac{1}{4}}(S, \mathcal{J}) \leq order(\mathcal{J}) T_{min}(\mathcal{J})$. An application of Lemma 2.1 finishes the proof. □

We show in the next section how to use parallel composition to achieve a better competitive ratio. First we state a basic property of parallel composition.

Lemma 7.5 (Parallel Composition) *Let $S = S_1 \oplus S_2 \oplus \cdots \oplus S_h$ be a parallel composed schedule of S. If for $0 \leq \alpha, \beta \leq 1$ at time t the efficiency is at least α (with respect to the submesh U_j of size $\left\lfloor \frac{n_1}{h} \right\rfloor \times n_2$) for at least βh of all the composed schedules (S_j's), then the efficiency of S is at least $\alpha\beta$ at time t.*

Proof: Let α_j be the efficiency of S_j at time t (with respect to submesh U_j). Then the efficiency of S is $(\sum_j \frac{\alpha_j N}{h})/N \geq \alpha\beta$. □

7.2 An $O(\log\log N)$-competitive Algorithm

In this section we construct an $O(\log\log N)$-competitive online scheduling algorithm for a $n_1 \times n_2$ mesh of N processors. Later we will prove a better bound, but this algorithm is used as a building block.

The algorithm schedules small and large jobs separately, using different strategies. We first describe the algorithm for the set of small jobs $\mathcal{J}'$. The large jobs are discussed later in Lemma 7.8.

The algorithm *dynamically* partitions the jobs of J' into $h = O(\log\log N)$ subsets $\mathcal{J}_1, \ldots, \mathcal{J}_h$ and then generates a schedule S_l for each job class $\mathcal{J}_l$ such that $T_{<\frac{1}{4}}(S_l, \mathcal{J}_l) \leq t_{max}(\mathcal{J})$. The final result is the serial composition $S = S_1 \odot \cdots \odot S_h$.

ALGORITHM PARALLEL
let $i = 1$;
repeat

1. **let** $\mathcal{J}_i$ be all jobs that have not been scheduled yet; **let** $h_i = order(\mathcal{J}_i)$; assign to each nonempty job class $\mathcal{J}_i^{(l)}$ a $\left\lfloor \frac{n_1}{h_i} \right\rfloor \times n_2$ submesh denoted by $U_{i,l}$;
2. apply CLASS to all nonempty sets $\mathcal{J}_i^{(l)}$ on $U_{i,l}$ until the first time when the total efficiency of the running jobs (in the whole $n_1 \times n_2$ mesh) is less than $\frac{1}{8}$; then stop (all instances of) CLASS but allow all running jobs to be finished;
3. **let** $i = i + 1$;

until all jobs are scheduled

Let S_i be the part of schedule generated during the ith pass through the repeat-until loop; we call it *phase* i. The time when CLASS is stopped is called the *checkpoint* of the phase i. Let p be the number of phases generated.

Claim 7.6 *In the above algorithm $h_{i+1} \leq \frac{h_i}{2}$, for all phases $i < p$.*

Proof: Following Lemma 7.2, if $\mathcal{J}_i^{(l)}$ is not empty after the phase i, then the efficiency of the schedule for the jobs in $\mathcal{J}_i^{(l)}$ (with respect $U_{i,l}$) is at least $\frac{1}{4}$ at the checkpoint of the phase i. Therefore, by Lemma 7.5, the efficiency at the checkpoint of phase i

is at least $\frac{h_{i+1}}{4h_i}$. On the other hand, by the definition of a checkpoint, the efficiency is less than $\frac{1}{8}$. Hence $h_{i+1} \le \frac{h_i}{2}$. $\square$

Lemma 7.7 PARALLEL *is* $O(\log(order(\mathcal{J}')))$*-competitive for* $\mathcal{J}'$.

Proof: The schedule generated by the algorithm is of the form $S = S_1 \odot \cdots \odot S_p$. Each S_i is of the form $S_i = \bigoplus_{\{l|\mathcal{J}_i^{(l)} \neq \emptyset\}} S_{i,l}$, where $S_{i,l}$ is a schedule generated by applying CLASS to $\mathcal{J}_i^{(l)} - \mathcal{J}_{i+1}^{(l)}$ on submesh $U_{i,l}$. The time between the checkpoint of phase i and the beginning of phase $i+1$ is bounded by t_{max}. By Claim 7.6, the number of phases is bounded by $\log(order(\mathcal{J}'))$. Thus, by Lemmas 7.3, 7.5 and 2.1, PARALLEL is $(8 + \log(order(\mathcal{J}')))$-competitive. (In fact, if we stop CLASS when the efficiency is below $\frac{\log\log N}{(\log\log\log N)^2}$ instead of $\frac{1}{8}$, we can by more careful analysis show that PARALLEL is $O(\frac{\log\log N}{(\log\log\log N)^2})$-competitive.) $\square$

The next lemma deals with the large jobs. The key observation is that the set of large jobs has only a few job classes, and therefore the performance of the algorithms PARALLEL and SERIAL is better on this restricted set of jobs.

Lemma 7.8 *There is an* $O(\log\log\log N)$*-competitive online scheduling algorithm for* $\mathcal{J} - \mathcal{J}'$.

Proof: Let $\mathcal{J}'' = \bigcup_{l=0}^{\lceil \log\log\log N \rceil} \mathcal{J}^{(l)}$. Applying SERIAL to $\mathcal{J}''$ constructs a $O(\log\log\log N)$-competitive schedule S_1' for $\mathcal{J}''$, because $order(\mathcal{J}'') = O(\log\log\log N)$. Notice that the order of $\mathcal{J} - \mathcal{J}' - \mathcal{J}''$ is at most $\log\log N$. This and the definition of J'' guarantees that when we apply applying PARALLEL to $\mathcal{J} - \mathcal{J}' - \mathcal{J}''$, all jobs are small enough to fit into the appropriate partition. By Lemma 7.7, PARALLEL generates a schedule S_2' for $\mathcal{J} - \mathcal{J}' - \mathcal{J}''$ with competitive ratio $O(\log order(\text{J-J'-J''})) = O(\log\log\log N)$. Using Lemma 7.3, we see that $S_1' \odot S_2'$ is a $O(\log\log\log N)$-competitive schedule for $\mathcal{J} - \mathcal{J}'$. $\square$

Theorem 7.9 *There is an* $O(\log\log N)$*-competitive online scheduling algorithm for the two-dimensional mesh of* N *processors.*

Proof: Serially compose the schedule for $\mathcal{J}'$ generated by PARALLEL (see Lemma 7.7) and the schedule for $\mathcal{J} - \mathcal{J}'$ from Lemma 7.8. $\square$

7.3 An Optimal Algorithm

In this section we construct an $O(\sqrt{\log\log N})$-competitive online scheduling algorithm for the two-dimensional mesh. In the next section, we prove a matching lower bound.

The main ingredient of the algorithm is to make an optimal trade-off between the average efficiency of the schedule and the amount of time that the efficiency of the schedule is below average. Instead of using a whole mesh as in PARALLEL, we schedule new jobs in just a small part of it. When the efficiency is too low, we can use the next part immediately, instead of waiting for all running jobs to finish.

Following Lemma 7.8, we can process the large jobs separately by an $O(\log\log\log N)$-competitive algorithm. Therefore we assume that there are no large jobs, $\mathcal{J} = \mathcal{J}'$.

Let $k = \lceil\sqrt{\log order(\mathcal{J})}\rceil = O(\sqrt{\log\log N})$.

Let U_j, $1 \le j \le k$, be a $\left\lfloor\frac{n_1}{k}\right\rfloor \times n_2$ submesh (See Figure 2).

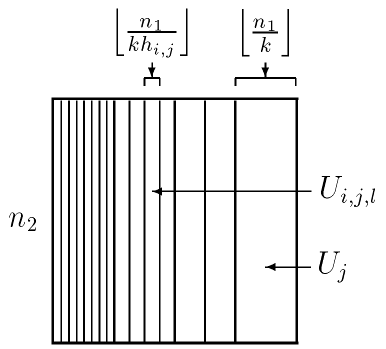

Figure 2: Mesh partition and the submeshes according to BALANCED PARALLEL

ALGORITHM BALANCED PARALLEL

let $i = 1$;

repeat

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

  1. **let** $\mathcal{J}_{i,j}$ be all jobs that have not been scheduled yet;
     let $h_{i,j} = order(\mathcal{J}_{i,j})$;
     assign to each nonempty job class $\mathcal{J}_{i,j}^{(l)}$ a $\left\lfloor\frac{n_1}{kh_{i,j}}\right\rfloor \times n_2$ submesh of U_j denoted by $U_{i,j,l}$ (see Figure 2);

2. apply CLASS to all nonempty sets $\mathcal{J}_{i,j}^{(l)}$ on $U_{i,j,l}$ until the first time when the efficiency of all currently running jobs with respect to U_j is less than $\frac{1}{8}$, then stop all instances of CLASS but allow the current jobs to be processed;
   if $j = k$ **then** wait until all jobs are finished (otherwise continue immediately);
3. **let** $j = j + 1$;

end

let $i = i + 1$;

until all jobs are scheduled

As in the proof of Claim 7.6 we can prove that $h_{i,j}$ decreases by a factor of 2 after each pass through the for loop. That means that during each phase (each pass through the repeat-until loop) it decreases by at least a factor of 2^k and therefore there can be at most $\lceil \log(order(\mathcal{J}))/k \rceil = O(\sqrt{\log\log N})$ phases. The same method as in the proof of Lemma 7.7 shows that BALANCED PARALLEL is $O(\sqrt{\log\log N})$-competitive for $\mathcal{J}'$. This together with Lemma 7.8 proves that there is a $O(\sqrt{\log\log N})$-competitive algorithm for two-dimensional meshes.

Theorem 7.10 *There is an $O(\sqrt{\log\log N})$-competitive online scheduling algorithm for the two-dimensional mesh of N processors.* □

8 Lower bound

We prove that no online scheduling on an $n \times n$ mesh of N processors can achieve a competitive ratio better than $\frac{1}{42}\sqrt{\log\log N}$. This proves that the algorithm in Section 7 is optimal up to a constant factor. (Note that it is sufficient to prove the lower bound for square meshes.)

The adversary tries to restrict the possibilities of the scheduler so that he has to act similar as the optimal algorithm we presented in Section 7.3. Lemma 8.1 is the key technical point. It shows that the adversary can restrict the actions of the scheduler substantially. More precisely, if the efficiency is high, the adversary is able to find a small subset of the running jobs which effectively block a large portion of the mesh. This implies that new jobs have to take new space, and eventually there is no more space available and the scheduler must wait until the running jobs are stopped. Of course, the adversary cannot go on forever. The price he pays is that after each such step the number of distinct sizes of available jobs is reduced. Nevertheless, it is possible to repeat this process sufficiently many times before all available job sizes are eliminated.

8.1 Notation

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

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

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

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

8.2 The scheduling problem

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

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

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

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

8.3 Adversary strategy

The adversary strategy is based on the following lemma which is proved in Section 8.4.

Lemma 8.1 *Suppose that we are scheduling the job system from Section 8.2. Then at any time of the schedule and for any interval* $I' = [a\,..\,b]$, $0 \le a \le b \le t$, *and* $\overline{I'} = [0\,..\,t] - I'$, *there exists a set* $\mathcal{D} = \mathcal{D}(I') \subseteq \mathcal{C}(\overline{I'})$ *such that* $\mathrm{eff}(\mathcal{D}) \le \frac{1}{8k}$ *and every normal* $(\frac{n}{4ks^b}, \frac{s^a}{4k})$*-rectangle intersected by* $\mathcal{C}(\overline{I'})$ *is also intersected by* $\mathcal{D}$.

The *adversary* maintains an *active interval* denoted by I. Initially $I = [0\,..\,t]$ and with time I gradually gets smaller. Let T be some fixed time. The adversary reacts to the schedulers actions according to the following steps.

SINGLE JOB: If the scheduler schedules some job on a submesh with an area smaller than $\frac{n}{k}$, then the adversary removes all other jobs (both running and waiting ones) and runs this single job for a sufficient amount of time.
DUMMY: If the scheduler starts a job that does not belong to a job class in the active interval (a job from $\mathcal{J}_j$, $j \notin I$), then the adversary removes it immediately.
CLEAN UP: If the time since the last CLEAN UP (or since the beginning) is equal to T and there was no SINGLE JOB step, then the adversary removes all running jobs, i.e., assigns the running times so that they are completed at this point.
DECREASE EFFICIENCY: If $\mathrm{eff}(\mathcal{C})$ exceeds $\frac{1}{k}$, the adversary does the following: He takes an interval $I' \subseteq I$ such that $|I'| = \lfloor \frac{|I|}{2} \rfloor$ and $\mathrm{eff}(\mathcal{C}(I')) \le \frac{\mathrm{eff}(\mathcal{C}(I))}{2}$ (such I' obviously exists - either the upper or the lower half of I, whichever has lower efficiency). Then he computes $\mathcal{D}(I')$ according to Lemma 8.1 and removes all jobs except those from $\mathcal{C}(I') \cup \mathcal{D}(I')$. He then sets the active interval to I'.

In the next two sections we prove that this strategy ensures that the competitive ratio is at most $\frac{1}{42}\sqrt{\log\log N}$ thus proving Theorem 8.7. In Section 8.4 the time

$T_S(\mathcal{J})$ of the online scheduler is compared to $T_{min}(\mathcal{J})$. Next in Section 8.5 we prove that T_{opt} and T_{min} differ only by a constant factor, thus finishing the proof.

8.4 Evaluation of the Adversary Strategy

In this section we prove that the adversary strategy from the previous section ensures that $T_S(\mathcal{J}) \geq kT_{min}(\mathcal{J})$.

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

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

The lower bound proof follows this outline. We first prove Lemma 8.1 which justifies the step DECREASE EFFICIENCY and then we show that each phase can have at most $6k$ of these steps. This implies that every schedule has to have at least k phases thus proving that $T_S(\mathcal{J}) \geq kT \geq kt_{max}(\mathcal{J})$. Also $T_S(\mathcal{J}) \geq kT_{\mathit{eff}}(\mathcal{J})$ because the efficiency is at most $\frac{1}{k}$ during the whole schedule.

The next claim is the key to the proof of Lemma 8.1. It states a purely geometrical fact which is true for an arbitrary set of submeshes $\mathcal{D}$, but we will only use it for $\mathcal{D}$ being a subset of running jobs.

Claim 8.2 *For some given y and v let $\mathcal{D}$ be a set of submeshes with height at most $\frac{y}{v}$. Then there exists a $\mathcal{D}' \subseteq \mathcal{D}$ such that* $\mathrm{eff}(\mathcal{D}') \leq \frac{2}{v}$ *and each normal $1 \times y$-rectangle intersected by $\mathcal{D}$ is also intersected by $\mathcal{D}'$.*

Proof: Let R be a normal $n \times y$-rectangle. We will define $\mathcal{D}_R \subseteq \mathcal{D}$ such that $\mathrm{eff}(\mathcal{D}_R) \leq \frac{2y}{vn}$ and it intersects all columns of R intersected by $\mathcal{D}$. It is then sufficient to set $\mathcal{D}' = \bigcup_R \mathcal{D}_R$, since every normal $1 \times y$-rectangle is a column of some normal $n \times y$-

rectangle. Because there are only $\lfloor \frac{n}{y} \rfloor$ normal $n \times y$-rectangles the efficiency of D' is $\mathrm{eff}(\mathcal{D}') \leq \lfloor \frac{n}{y} \rfloor \frac{2y}{vn} \leq \frac{2}{v}$.

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

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

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

Proof of Lemma 8.1:

Divide $\mathcal{C}(\overline{I'})$ into the following three classes:
$\mathcal{C}_1$ – all submeshes which heights are at most s^{a-1},
$\mathcal{C}_2$ – all submeshes which heights as well as width are at most $\frac{n}{s^{b+1}}$ and
$\mathcal{C}_3$ – all submeshes which widths are at most s^{a-1}.

All submeshes of $\mathcal{C}([0\,..\,a-1])$ are either in $\mathcal{C}_1$ or in $\mathcal{C}_3$ depending on their orientation and all submeshes of $\mathcal{C}([b+1\,..\,t])$ are in $\mathcal{C}_2$, hence $\mathcal{C}(\overline{I'}) = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$. The following four applications of Claim 8.2 to these sets give us:
$\mathcal{D}_1 \subseteq \mathcal{C}_1$, which intersects all normal $1 \times \frac{s^a}{4k}$-rectangles intersected by $\mathcal{C}_1$;
$\mathcal{D}_2 \subseteq \mathcal{C}_2 \cup \mathcal{C}_3$, which intersects all normal $\frac{n}{4ks^b} \times 1$-rectangles intersected by $\mathcal{C}_2 \cup \mathcal{C}_3$;
$\mathcal{D}_3 \subseteq \mathcal{C}_1 \cup \mathcal{C}_2$, which intersects all normal $1 \times \frac{n}{4ks^b}$-rectangles intersected by $\mathcal{C}_1 \cup \mathcal{C}_2$;
$\mathcal{D}_4 \subseteq \mathcal{C}_3$, which intersects all normal $\frac{s^a}{4k} \times 1$-rectangles intersected by $\mathcal{C}_3$;

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

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

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

Claim 8.3 *Let $j \in I$ and R be a normal $(\frac{n}{4ks^j}, \frac{s^j}{4k})$-rectangle that does not intersect $\mathcal{C}$ at this step of the schedule. Then during the current phase R has never intersected $\mathcal{C}$.*

Proof: Because our assumptions state that the DECREASE EFFICIENCY step is the only one removing jobs during a phase it is sufficient to prove that no previous DECREASE EFFICIENCY step removed all the jobs which intersected R.

Assume that the active interval before such a previous DECREASE EFFICIENCY step is $[a \mathinner{..} b]$ then $a \leq j \leq b$ (since the active interval never grows). Because the dimensions of R are integer multiples of the dimension of the normal rectangles from Lemma 8.1 R can be partitioned into a set $\mathcal{R}$ of such rectangles without any leftover. According to the assumption, the rectangle R, and hence some rectangle from $\mathcal{R}$, is intersected by $\mathcal{C}$ before the DECREASE EFFICIENCY step. But then (following the construction and Lemma 8.1) this rectangle, and hence also R, is also intersected by $\mathcal{C}$ after this step, which leads to a contradiction. □

Claim 8.4 *Each phase can have at most $6k$ DECREASE EFFICIENCY steps.*

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

The efficiency immediately before a DECREASE EFFICIENCY step is at most $\frac{1}{k} + \frac{1}{n}$ (since the threshold $\frac{1}{k}$ was reached by the last job started). By the construction and Lemma 8.1 the efficiency after the DECREASE EFFICIENCY step is at most $\frac{1}{2k} + \frac{1}{2n} + \frac{1}{8k} \leq \frac{2}{3k}$ for sufficiently large n.

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

Theorem 8.5 *The adversary strategy forces $T_S(\mathcal{J}) \geq \frac{1}{3}\sqrt{\log\log N}\, T_{min}(\mathcal{J})$.*

Proof: We first prove that after k phases the active interval I is still nonempty. Each

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

If j is in the active interval at the end of the kth phase then it was in the active interval during all k phases and the adversary could remove the jobs of $\mathcal{J}_j$ only in the CLEAN UP steps. During one CLEAN UP step he could remove at most kn of such jobs (because each occupies a minimal area), so some of them are not finished before the end of the kth phase. Hence $T_S(\mathcal{J}) \geq kT \geq kt_{max}(\mathcal{J})$.

During the whole schedule the efficiency was at most $\frac{1}{k}$, hence $T_S(\mathcal{J}) \geq kT_{\mathit{eff}}(\mathcal{J})$ and $T_S(\mathcal{J}) \geq k \times \max(T_{\mathit{eff}}(\mathcal{J}), t_{max}(\mathcal{J})) = kT_{min}(\mathcal{J})$. □

This together with the results of the next section proves Theorem 8.7.

8.5 Offline scheduling

For the proof of the lower bound we still need to prove that an offline scheduler for the two-dimensional mesh can schedule the job system in time which is a constant multiple of T_{min}. The problem is essentially a special case of three-dimensional bin packing problem.

Let $t = t_{max}(\mathcal{J})$. Divide $\mathcal{J}$ into the classes $\mathcal{J}_{l,j} = \{J_i \in \mathcal{J} \mid \frac{n}{2^{l+1}} < a_i \leq \frac{n}{2^l}, \frac{n}{2^{j+1}} < b_i \leq \frac{n}{2^j}\}$, $0 \leq l \leq j \leq \lfloor\log n\rfloor$. Now partition each $\mathcal{J}_{l,j}$ into $\mathcal{D}'_{l,j}$ and $\mathcal{D}_{l,j,u}$, $u = 1,\ldots,m_{l,j}$ so that $\frac{t}{2} \leq \sum_{J_i\in\mathcal{D}_{l,j,u}} t_i \leq t$ and $\sum_{J_i\in\mathcal{D}'_{l,j}} t_i \leq t$. This can be done using the first fit bin packing algorithm (see e.g. [6]) for each l and j, so that u indexes the bins $\mathcal{D}_{l,j,u}$ that are full and $\mathcal{D}'_{l,j}$ is the leftover bin for each l and j.

It is easy to see that for each $\mathcal{D}_{l,j,u}$ ($\mathcal{D}'_{l,j}$) there exists a schedule $S_{l,j,u}$ ($S'_{l,j}$) to schedule the jobs on a $\lfloor\frac{n}{2^l}\rfloor\times\lfloor\frac{n}{2^j}\rfloor$-mesh in time t which has average efficiency at least $\frac{1}{8}$. We now treat each of these schedules as a single job. Because of the special sizes and times of these jobs they are easy to schedule.

First consider all $S_{l,j,u}$'s for some fixed l. Sorting them by increasing j and scheduling them in time t using first fit bin packing on an $\lfloor\frac{n}{2^l}\rfloor\times n$-mesh creates the schedules $S_{l,v}$, $v = 1,\ldots,v_l$ and S'_l. Notice that because of the special sizes, all $S_{l,v}$ have average efficiency of at least $\frac{1}{8}$ (even if n is not a power of 2).

Similarly create schedules S_w on $n \times n$-mesh in time t for all $S_{l,v}$'s with average efficiency of at least $\frac{1}{8}$ and a leftover S'.

Then create the final schedule $S(\mathcal{J})$ as follows:

ALGORITHM: OFFLINE

1. Apply the schedules S_w one by one.
2. Apply S'.
3. Apply S'_0.
4. Apply simultaneously all $S'_l, 1 \leq l \leq \lfloor \log n \rfloor$, place S'_l at $(\left\lfloor \frac{n}{2^l} \right\rfloor, 0)$.
5. Apply $S'_{0,0}$.
6. Apply simultaneously all $S'_{0,j}, 1 \leq j \leq \lfloor \log n \rfloor$ at $(0, \lfloor \frac{n}{2^j} \rfloor)$.
7. Apply simultaneously all $S'_{l,j}, 1 \leq l \leq \lfloor \log n \rfloor$ at $(\lfloor \frac{n}{2^l} \rfloor, \lfloor \frac{n}{2^j} \rfloor)$.

Theorem 8.6 *For all job systems $\mathcal{J}$ and $S(\mathcal{J})$ as above, $T_S(\mathcal{J}) \leq 14 T_{min}(\mathcal{J})$.*

Proof: The phase 1 runs with average efficiency of at least $\frac{1}{8}$ and the rest of the schedule takes at most $6t$. Hence $T_S(\mathcal{J}) \leq 8 T_{eff}(\mathcal{J}) + 6 t_{max}(\mathcal{J}) \leq 14 T_{min}(\mathcal{J})$. □

This together with Theorem 8.5 gives the following theorem.

Theorem 8.7 *The competitive ratio of any online scheduling algorithm for a mesh of N processors is at least $\Omega(\sqrt{\log \log N})$.* □

9 The d-dimensional mesh

The online scheduling algorithm for d-dimensional meshes ($d > 2$) emulates the two-dimensional one. We assume without loss of generality that each job $J = (a_1, \ldots, a_d)$ satisfies $a_1 \leq a_2 \leq \ldots \leq a_d$.

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

A d-dimensional version of the algorithms in question is obtained by partitioning the jobs in job classes such that all jobs in each job class differ from each other by a factor of at most 2 in every dimension, i.e., $J_1 = (a_1, a_2, \ldots, a_d)$ and $J_2 = (b_1, b_2, \ldots, b_d)$ are in the same job class if for all i, $\lfloor \log(\frac{n}{a_i}) \rfloor = \lfloor \log(\frac{n}{b_i}) \rfloor$. Then the first dimension is used to schedule the jobs according to the one-dimensional algorithm, while the process of dividing the mesh and allocating the submeshes to the job classes stays the same. The loss in efficiency is $O(2^d)$.

Whereas so far large jobs have caused us little problems now they are a major obstacle. For all small jobs, i.e., jobs with $a_1 \le \frac{n}{(\log n)^d \log\log n}$, we just use the generalized BALANCED PARALLEL algorithm and achieve a competitive factor of $2^d\sqrt{d}\sqrt{\log\log N}$.

The problem with the large jobs is that the number of job classes is $(\log\log n)^d$, and not only $\log\log n$ as before. So just applying PARALLEL once and then SERIAL leads to a d^d factor in the competitive ratio. We can partially circumvent this by applying PARALLEL recursively. In every step of PARALLEL the number of job classes is reduced by an application of log. This implies that after $\log^* n$ iterations the number of job classes stabilizes to $(d \log d)^d$. After this we finish the schedule using SERIAL for these job classes. This results in a $O(2^d d \log d\sqrt{\log\log N} + 2^d(d\log d)^d)$-competitive algorithm.

Theorem 9.1 *There is an online scheduling algorithm for d-dimensional meshes of N processors that is $O(2^d d \log d\sqrt{\log\log N} + 2^d(d\log d)^d)$-competitive. Moreover, when d is a constant, the algorithm is optimal up to a constant factor.* □

10 Concluding remarks

We have made a contribution towards solving the problem of dynamic scheduling on parallel machines by introducing a model which captures some important aspects of the practical scheduling problem. The novelty of this model is that it includes the network topology of the parallel machine. We prove that the geometric structure of the network topology cannot be neglected. It requires careful consideration and eventually an abandonment of the natural greedy-like scheduling approach.

The techniques of this paper can be extended to some more network topologies. ORDERED is $(2 - \frac{1}{N})$-competitive for complete binary trees and our algorithm for two-dimensional meshes can also be applied to two-dimensional meshes of trees to obtain a $\Theta(\sqrt{\log\log n})$-competitive algorithm.

In this paper we have made the assumption that only the running times are given dynamically and that there are no dependencies among jobs. In this case the simulation assumption does not seem to matter. However, it is crucial for the work of [4] which introduces dependencies among jobs to the model.

The questions which remains open are, how can the model be extended to capture the practical scheduling even better and if the competitive ratio is the right performance measurement. Also of interest is whether randomization can help to improve the performance of the scheduling algorithm.

Acknowledgements:

We would like to thank Steven Rudich, Danny Sleator, Ming-Yang Kao, David Peleg and an anonymous referee for helpful discussions and comments.

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: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing* (ACM, 1988) 192–204.

[2] E. Davis and J.M. Jaffe, Algorithm for scheduling tasks in unrelated processors, *JACM* (ACM, 1981) 712–736.

[3] F.G. Feitelson and L. Rudolph, Wasted resources in gang scheduling, in: *Proceedings of the 5th Jerusalem Conference on Information Technology* (1990).

[4] A. Feldmann, M.-Y. Kao, J. Sgall, and S.-H. Teng, Dynamic scheduling on parallel machines with dependencies, in: *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, (ACM, 1993) 642–651.

[5] A. Feldmann, J. Sgall, and S.-H. Teng, Dynamic scheduling on parallel machines, in: *32st Annual Symposium on Foundations of Computer Science*, (IEEE, Oct. 1991), 111–120.

[6] M.R. Garey and D.S. Johnson, *Computers 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* (1966) 1563–1581.

[8] R.M. Karp and V. Ramachandran, A survey of parallel algorithms for shared memory machines, Technical Report UCB/CSD 88-408, UC. Berkeley, 1988.

[9] S.R. Kosaraju and M.J. Atallah, Optimal simulation between mesh-connected arrays of processors, in: *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, (ACM, 1986) 264–272.

[10] H.T. Kung, Computational models for parallel computers, Technical Report CMU-CS-88-164, CMU, 1987.

[11] F.T. Leighton, *Complexity Issues in VLSI* (Foundations of Computing, MIT Press, Cambridge, MA, 1983).

[12] J.K. Lenstra, D.B. Shmoys, and E. Tardos, Approximation algorithm for scheduling unrelated parallel machines. *Mathematical Programming*, **46** (1990) 259–271.

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

[14] A. Ranade. How to emulate shared memory, in: *28th Annual Symposium on Foundations of Computer Science*, (IEEE, Oct. 1987) 185–194.

[15] D.B. Shmoys, J. Wein, and D.P. Williamson, Scheduling parallel machines online, in: *32st Annual Symposium on Foundations of Computer Science*, (IEEE, Oct. 1991), 131–140.

[16] D.D. Sleator and R.E. Tarjan, Amortized efficiency of list update and paging rules, *Communications of the ACM*, Vol. 28, Num. 2 (ACM 1985) 202–208.