Randomized On-Line Scheduling of Parallel Jobs

Jiří Sgall*
Mathematical Institute
Academy of Sciences of the Czech Republic
Žitná 25
11567 Praha 1
Czech Republic

E-mail: `sgallj@earn.cvut.cz`

Abstract

We study randomized on-line scheduling on mesh machines. We show that for scheduling independent jobs randomized algorithms can achieve a significantly better performance than deterministic ones; on the other hand with dependencies randomization does not help.

*this research was done at Carnegie-Mellon University, Pittsburgh, PA, U.S.A.

1 Introduction

In this paper we study the power of randomization for on-line scheduling of parallel jobs in a model which was introduced and studied in [4, 3] for deterministic scheduling. We give a randomized on-line algorithm for scheduling independent jobs on mesh machines which is significantly better than the optimal deterministic algorithm from [4]. On the other hand, for scheduling of jobs with dependencies on one-dimensional meshes, we show that randomization does not help—no randomized algorithm has asymptotically better performance on the worst-case input than the optimal deterministic algorithm from [3].

These results are particularly interesting in the view of the fact that both the lower bound for deterministic algorithms for scheduling independent jobs on two-dimensional meshes in [4] and the lower bound for deterministic algorithms for scheduling jobs with dependencies on one-dimensional meshes in [3] use a very similar technique. Yet the second lower bound generalizes to randomized algorithms but the first one does not.

The influence of randomness on the performance of on-line algorithms was studied for many problems. For paging problem (k-server problem on a uniform space) it was shown that the performance of randomized algorithms is dramatically better than the performance of deterministic algorithms [5, 9], see also [8] for results on 2-server problem and a more general framework. However, for on-line scheduling, besides our randomized algorithm for scheduling independent jobs on meshes, we know of only one other result that shows that randomization provably improves the performance—namely the result of [2] on randomized scheduling of sequential jobs on two processors (in a slightly different model). However, in that result the improvement is only by a small constant factor, whereas in our case the improvement is much larger; also our technical tools are different and much more sophisticated.

In our model, each parallel job requests some number of processors arranged in a mesh of given dimensions and will run on such a mesh for some fixed time, called its running time, regardless of when we execute it. It can be scheduled on any such submesh, or on a submesh of smaller dimensions using virtualization; this means that each processor simulates several virtual processors and the running time increases proportionally. Scheduling may also be constrained by dependencies between different jobs; this means that it may be the case that a job cannot be started before some other jobs are finished. Once a job is started, it has to be completed on the same processors without stopping, it cannot be moved to other processors or stopped and restarted later on the same or different processors. We can schedule more than one job at once, as long as we can satisfy the requirements of all scheduled jobs simultaneously. Our task is to schedule a given set of jobs so that the total time is as small as possible. In the on-line problem we do not know the running times in advance; the only way to determine the running time of a job is to execute it. We also might not know about the existence of a job until all jobs on which it depends are finished. An on-line algorithm is evaluated by a competitive ratio. A randomized scheduling algorithm is σ-competitive if for every input the expected length of the produced schedule is at most σ times longer than the optimal schedule.

In the presence of dependencies we cannot use randomization efficiently. We prove a lower bound showing that no randomized algorithm for scheduling on one-dimensional meshes with virtualization has a better competitive ratio than $\Theta(\frac{\log N}{\log\log N})$, where N in the number of processors. Since there exists a deterministic algorithm which achieves a competitive ratio $\Theta(\frac{\log N}{\log\log N})$, see [3], randomization does not help in this case.

On the other hand, if no dependencies are allowed, we obtain a randomized on-line algorithm for scheduling on d-dimensional meshes with competitive ratio bounded by $O(4^d)$, a bound which does not depend of the number of processors. Moreover, under some small restrictions on the sizes of the jobs we obtain a much stronger result, namely the competitive ratio of our algorithm does not even depend on the dimension of the mesh. This is a significant improvement over the deterministic case studied in [4]—there it is proved that if the dimension of the mesh is constant, the optimal competitive ratio for deterministic algorithm is $\Theta(\sqrt{\log\log N})$, where N is the number of processors [4], while our randomized algorithm achieves a constant competitive ratio; also the dependency of the competitive ratio on d is much worse for the best known deterministic algorithm, namely $O((2d\log d)^d)$. (We have no lower bounds in terms of d for either deterministic or randomized case. Note that in practice d is very small, typically a constant, as arbitrarily large meshes can be built without changing d. Hence the competitive ratio is not that large even if the dependency on d is exponential.)

To achieve such a strong result, we estimate for each job size the total running time of all jobs of that size based on a random sample. However, it is not clear how this can lead to a constant competitive ratio, since the number of different job sizes depends on the number of processors, and in particular it is exponential in d. We use Chernoff-Hoeffding bounds in a powerful way to prove that with some constant probability all of the many different instances of sampling give a good approximation of the work. It is also interesting that randomization is used in our algorithm only to randomly permute the jobs at the beginning for the purpose of sampling. If we assume that the usage pattern of a parallel machine does not change very fast, we can estimate the work of jobs of different sizes based on the previous usage of the machine, and then schedule them very efficiently even without randomization. This is actually used in practice, since scheduling is sometimes done manually based on the previous experience and data. Our result explains to some extent why it might be very useful to consider the estimates based on previous experience. If these estimates are good, it saves us the sampling which is a relatively expensive part of our algorithm.

Section 2 introduces our model for scheduling of parallel jobs and some notation. In Section 3 we give the algorithm for scheduling independent jobs. In Section 5 we prove the lower bound for scheduling with dependencies. This paper extends the conference version [11]; the results were also presented in the dissertation [10].

2 A model for scheduling

Here we describe a simplified version of the model as it applies to mesh machines. For a complete description, discussion and practical considerations see [10].

Parallel machines, parallel jobs and virtualization

A *parallel machine* whose underlying topology is a *d-dimensional mesh* is represented by a grid graph of dimension d.

Each *parallel job* requests a mesh of processors with certain dimensions. It can be scheduled on any submesh of the machine with the same dimensions and will run on it for its *running time*. We assume that all the processors run at the same speed, thus the running time is independent of the choice of the submesh. The number of processors of the requested submesh is called the *size* of the job; the product of the size and the running time of the job is its *work*.

An important fact is that any parallel job can be scheduled on a smaller mesh of processors than it requests using a technique called *virtualization*. Each of the processors then simulates several processors requested by the job, and the running time is proportionally longer. We assume that the work of the job is preserved, since any mesh can be mapped on a smaller mesh so that the network topology is preserved and hence the simulation is efficient. (This is a common assumption, somewhat simplifying in case that the dimensions of the larger mesh are not divisible by the dimensions of the smaller one. However, this is irrelevant in our case, since we are only interested in the asymptotic behaviour.)

A *one-dimensional mesh* machine consists of N processors arranged on a line, where each of them is connected to its neighbors. If a job requests a submesh of p processors, it can be scheduled on any contiguous segment of the machine with $p' \leq p$ processors and the running time increases by a simulation factor of p/p'. A *two-dimensional* $n_1 \times n_2$ mesh machine is represented by a rectangular grid with width n_1 and height n_2. A job which requests an $a \times b$ mesh can be scheduled on an $a' \times b'$ or $b' \times a'$ mesh for $a' \leq a$ and $b' \leq b$ and the running time then increases by a factor of $ab/a'b'$. In contrast to one-dimensional meshes, the requested mesh is not determined by the size of a job, since we distinguish a 10×10 mesh from a 5×20 mesh.

Parallel job systems, schedules and scheduling algorithms

A *parallel job system* is a collection of jobs with the dependencies given by a directed acyclic graph called the *dependency graph*. The nodes represent the jobs and edges represent the dependencies between them; a job can be scheduled when all its predecessors in the dependency graph are finished.

A *schedule* (for a parallel job system) is an assignment of a submesh of processors and a time interval to each job such that all requirements are satisfied—for each job the assigned submesh is consistent with the requested one, the length of the assigned time interval is its running time scaled appropriately, and if there is a dependency between two jobs, then the time interval assigned to the first job ends before the interval assigned to the second job starts. A processor can be assigned to at most one job at any time. Once a job is started on some processor or a set of processors, it has to run on them until completion—we do not allow a job to be preempted, i.e., to be moved to a different set of processors where it would continue to run; we also do not allow a job to be stopped and restarted later on the same or different processors.

A *scheduling algorithm* takes as an input a parallel job system and produces a schedule for it; inputs might be restricted for example to all job systems without dependencies (with no edges in the dependency graph). A scheduling algorithm is *off-line* if it receives the complete information as its input, i.e., all jobs, their dependencies and running times. An *on-line* scheduling algorithm can only determine the running time of any job by scheduling and completing it, and at any given moment the algorithm only knows the resource requirements of the available jobs but has no information about the future jobs and their dependencies.

The performance measure

We measure the performance by the *length of a schedule* (makespan). The length of schedule S, denoted by $T(S)$, is the time when the last job finishes. An *optimal* schedule for a job system $\mathcal{J}$ is a schedule with the minimal length, denoted by $T_{\rm opt}(\mathcal{J})$. An optimal schedule can be computed off-line, given full information, and with unlimited computational power.

An on-line algorithm is evaluated by the *competitive ratio*, which compares the performance of an on-line algorithm to the optimal schedule. In the case of a randomized scheduling algorithm, the competitive ratio for a given input is the ratio of the expected length of a schedule produced by the on-line algorithm to the length of an optimal schedule; the expectation is taken over the random bits of the scheduling algorithm. The competitive ratio of an on-line algorithm is the supremum of the competitive ratio over all inputs. Equivalently, we say that a scheduling algorithm is *σ-competitive* if for every job system $\mathcal{J}$ the expected length of the schedule S generated by the algorithm satisfies $\mathrm{E}[T(S)] \leq \sigma T_{\rm opt}(\mathcal{J})$.

To analyze scheduling algorithms, the concept of efficiency is very important. If a schedule has high efficiency except for a short period of time, then the schedule cannot be much longer than the optimal one and hence the competitive ratio is small. The *efficiency of a schedule* at time t is the number of processors busy at time t divided by N, the total number of processors of the machine. Efficiency of an algorithm refers to the efficiency of the schedule generated by that algorithm (for a particular job system); if this algorithm is only used to schedule jobs on a submesh of the machine, it refers to the efficiency with respect to that submesh.

3 Scheduling independent jobs on meshes

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

The basic idea of the algorithm is following. We partition the jobs according to their size. (Since the jobs are independent, we know the sizes of all of them, and the only thing we do not know is their running time.) For each size we schedule a random sample of jobs and estimate the total work of the jobs of that size. Then we partition the mesh so that each job size is assigned an area proportional to the estimated work of jobs of that size and schedule all jobs of given size in the assigned area.

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

Even if we have a correct bound on the longest running time, sampling is not trivial. We have to guarantee that our sample is sufficiently good while keeping the time required for sampling small. Sampling a fixed number or fixed fraction of jobs does not work—for some size we can have many jobs with small running time and a few very long jobs; in that case we are not likely to see any long job and then we have no useful information about the total work. Instead, we sample until we see jobs with total running time exceeding some bound. Intuitively, if there are only two long jobs in the first quarter of the jobs, it is likely that the number of long jobs is close to eight; if there are two long jobs among the first four, probably about a half of the jobs are long. Even though we have a much larger sample in the first case, Chernoff-Hoeffding bounds guarantee that in both cases the probability of a wrong estimate is about the same. Note that while sampling in this way, we may schedule most or all jobs before the bound is reached.

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

The algorithm

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

A job requiring a $a_i \times b_i$ mesh with running time t_i is represented as (a_i, b_i, t_i) (of course, on-line algorithms do not know t_i); without loss of generality we assume that for each job $a_i \geq b_i$. The instruction "**wait**" in our pseudocode means that the algorithm waits until all currently running jobs are finished.

First we present a deterministic algorithm CLASS from [4, 10] which is used to schedule jobs requiring submeshes of the same width after the appropriate estimates are obtained. It is easy to see that if all the dimensions are powers of two, and all the jobs have equal width, the efficiency of CLASS is 1 as long as there is a job available, see [4, 10].

Algorithm CLASS

Let a be the maximal width of a job, $a = \max\{a_i | (a_i, b_i, t_i) \in \mathcal{J}\}$. Partition the mesh into $\lfloor n_1/a \rfloor$ strips of size $a \times n_2$;

Order the jobs according to their second dimension, so that $b_1 \geq b_2 \geq \ldots \geq b_m$;

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

  while no $a \times b_i$ submesh is available **do** nothing;

  schedule the job $J_i = (a_i, b_i, t_i)$ on the first available $a \times b_i$ submesh (with the smallest y-coordinate) aligned with the left edge of the strip;

  end;

wait.

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

In step (1) of the algorithm we schedule the large jobs (the three job classes $\mathcal{J}^{(0)}$, $\mathcal{J}^{(1)}$, and $\mathcal{J}^{(2)}$). Steps (2) and (3) implement the doubling strategy for estimating the longest running time. At any point the estimate is $2^I\tau$, the loop in step (1) guarantees $\tau > 0$. Step (4) implements the sampling. It uses a fixed partition of the mesh illustrated by Figure 1. To

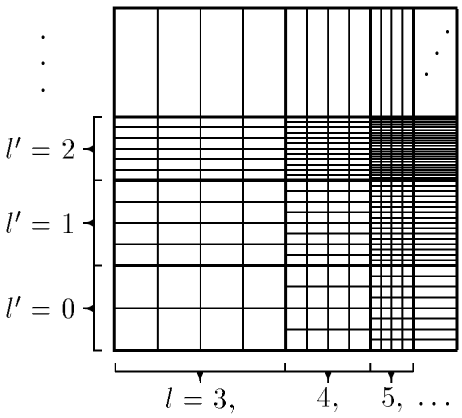

Figure 1: *The partition used for sampling in step (4) of the algorithm SAMPLE. (The thick lines show the submeshes $G_{l,l'}$, the thin lines the submeshes $G_{l,l',j}$.)*

each subclass $\mathcal{J}^{(l,l')}$, $l \geq 3$, $l' \geq 0$, we assign a submesh $G_{l,l'}$ of size $(4m_l)\times((l'+1)m_{l'}/4)$. Each $G_{l,l'}$ is divided into $(l'+1)2^l$ submeshes $G_{l,l',j}$ of size $m_l \times m_{l+l'}$. Hence $(l'+1)2^l$ jobs of each subclass $\mathcal{J}^{(l,l')}$ can be scheduled in parallel. We need to verify that these submeshes can be placed onto an $n \times n$ mesh so that they are pairwise disjoint. For a fixed l, the sum of the heights of submeshes $G_{l,l'}$ is bounded by $\sum_{l' \geq 0}(l'+1)n/2^{l'+2} = n$, therefore they all fit into a submesh of size $4m_l \times n$. The total width of these meshes for $l \geq 3$ is bounded by n, hence all these meshes fit into one $n \times n$ mesh. Step (5) computes the estimates based on previous sampling. For each l and l', $w_{l,l'}$ estimates the total running time of all jobs in $\mathcal{J}^{(l,l')}$ not scheduled before step (4). After rescaling, this also estimates the work, as the size of all jobs in $\mathcal{J}^{(l,l')}$ is the same. w_l estimates the total work of jobs in $\mathcal{J}^{(l)}$, and w estimates the total work of all unscheduled jobs. p_l is a number of processors proportional to w_l, z_l is the number of submeshes of size $m_l \times m_l$ actually assigned to the class $\mathcal{J}^{(l)}$ (the rounding guarantees that each job fits into the assigned mesh). Steps (6) and (7) schedule the jobs in areas approximately proportional to the estimated work in each class. The condition in (7) and the step (8) guarantee that we sample again if one of the estimates is wrong.

Algorithm SAMPLE

(1) Schedule the classes $\mathcal{J}^{(0)}$, $\mathcal{J}^{(1)}$ and $\mathcal{J}^{(2)}$ (one by one) using CLASS;

  while no job of nonzero time was scheduled **do**

    schedule one job;

  wait;

  let τ be the maximal running time of the jobs scheduled so far;

  let $I := 1$;

(2) During steps (3) to (7),

  if the running time of any job exceeds $2^I\tau$,

    then begin let $I := I + 1$; **goto** (3) **end**;

(3) **wait**;

(4) For every l, l', **let** $r_{l,l'}$ be the number of unscheduled jobs in $\mathcal{J}^{(l,l')}$;

  while the time elapsed in this step is $< 2 \cdot 2^I\tau$ **do**

    for all l, l', j **do**

      if $G_{l,l',j}$ is empty and there is an unscheduled job in $\mathcal{J}^{(l,l')}$

        then schedule a random unscheduled job in $\mathcal{J}^{(l,l')}$ onto $G_{l,l',j}$;

  wait;

(5) for every l, l', **if** there are unscheduled jobs in $\mathcal{J}^{(l,l')}$

    then begin

      let $k_{l,l'}$ be the smallest k such that the sum of the running times of the first k jobs from $\mathcal{J}^{(l,l')}$ scheduled during the preceding step (4) is at least $2(l'+1)2^l2^I\tau$;

      let $w_{l,l'} := 4r_{l,l'}(l'+1)2^l2^I\tau/k_{l,l'}$;

      end;

    else let $w_{l,l'}$ be the total running time of jobs from $\mathcal{J}^{(l,l')}$ scheduled during the preceding step (4);

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

  let $w := \sum_l w_l$;

  for every l, **let** $p_l := \frac{47}{48}n^2 w_l/w$; **let** $z_l := \lceil p_l/m_l^2 \rceil$;

(6) Partition the machine into a set of square submeshes $H_{l,j}$, $l > 0$, $1 \le j \le z_l$; $H_{l,j}$ has size $m_l \times m_l$ (see below for details);

(7) in parallel for each l schedule $\mathcal{J}^{(l)}$ on the collection of meshes $\{H_{l,1}, \ldots, H_{l,z_l}\}$ using CLASS, provided that

  if ((for some l all jobs from $\mathcal{J}^{(l)}$ are finished **and** the total work of the jobs from $\mathcal{J}^{(l)}$

scheduled during this step is less than $w_l/4$)

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

then interrupt all instances of CLASS (allowing running jobs to finish);

(8) **wait**;

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

We need to justify step (6). Since the meshes are squares whose sizes are powers of two, we can place them greedily starting with the largest mesh, so that the coordinates of each mesh are divisible by its size. See Figure 2 for an example. This process will place all the meshes

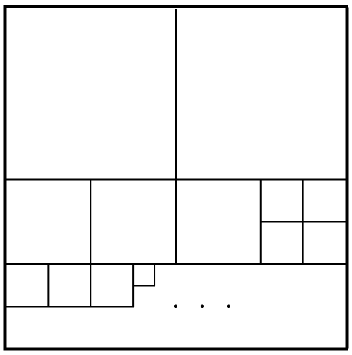

Figure 2: *An example of the partition of the mesh used in step (6) of the algorithm SAMPLE.*

as long as the number of processors is not too large. This is true in our case since the area is bounded by

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

In the rest of this section we analyze the performance of this algorithm and prove the following result.

Theorem 3.1 *The randomized algorithm SAMPLE is 28-competitive.*

Probability estimates

As our basic tool, we use Chernoff-Hoeffding bounds, which bound the probability that a sum of random variables differs significantly from its mean [7, 6, 1]. Many sources state them only for a random variable $\mathbf{S}$ which is a sum of independent 0-1 variables $\mathbf{X}_i$. However, the original Hoeffding paper [7] proves that the same bounds are true even for more general variables $\mathbf{X}_i$. First, it is sufficient if the values of each $\mathbf{X}_i$ are from the real interval $[0,1]$, not necessarily integers. Second, the variables can be produced by sampling without replacement from some universe, in which case the variables are not independent; independent variables correspond to sampling with replacement. Note that in an extreme case of sampling without replacement we obtain all elements of the universe, in which case we know the sum exactly.

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

$$\begin{aligned} \mathrm{Prob}[\mathbf{S} \le (1-\varepsilon)\mathrm{E}[\mathbf{S}]] &\le e^{-\varepsilon^2 \mathrm{E}[\mathbf{S}]/2} && \text{(A)} \\ \mathrm{Prob}[\mathbf{S} \ge (1+\varepsilon)\mathrm{E}[\mathbf{S}]] &\le e^{-\varepsilon^2 \mathrm{E}[\mathbf{S}]/3}. && \text{(B)} \end{aligned}$$

An elegant derivation for independent 0-1 variables can be found in [6]. To modify the proof for the more general $\mathbf{S}$ it is necessary to use convexity of the function $e^{h\mathbf{X}_i}$ and Jensen's inequality during the proof, see [7].

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

Lemma 3.2 *Let U be a set of r real numbers from $[0,1]$ with sum W. Let $\mathbf{X}_1, \ldots, \mathbf{X}_r$ be a sequence of random variables obtained by sampling without replacement out of the set U. Let $\mathbf{S}_i$ be the sum of the first i of them. Let $5 \le \lambda < W$ be given. Let $\mathbf{k}$ be the unique integer such that $\mathbf{S_{k-1}} < \lambda \le \mathbf{S_k}$ (note that $\mathbf{k}$ is a random variable). Then*

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

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

If $W < r\lambda/2\mathbf{k}$ then $\mathbf{k} < \alpha/2$ by the definition of α. From the definition of $\mathbf{k}$ it follows that for $\beta = \lfloor \alpha/2 \rfloor$, $\mathbf{S}_\beta \ge \lambda = \frac{\alpha}{\beta}\mathrm{E}[\mathbf{S}_\beta]$. Using (B) we get

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

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

If $W > 2r\lambda/\mathbf{k}$ then $\mathbf{k} > 2\alpha$. We put $\beta = \lfloor 2\alpha \rfloor$. Note that $\frac{\beta}{\alpha} \ge 2 - \frac{1}{\alpha} \ge 2 - \frac{1}{\lambda} \ge \frac{9}{5}$. By the definition of $\mathbf{k}$, $\mathbf{S}_\beta \le \lambda = \frac{\alpha}{\beta}\mathrm{E}[\mathbf{S}_\beta]$. Using (A), we get

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

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

Expected time analysis

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

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

Claim 3.3 *If no running time is larger than $2^I\tau$, then after step (6),* $\mathrm{Prob}[(\forall l,l')w_{l,l'}/4 \leq W_{l,l'} \leq w_{l,l'}] \geq 5/6$, *and therefore* $\mathrm{Prob}[(\forall l)w_l/4 \leq W_l \leq w_l] \geq 5/6$.

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

$$\mathrm{Prob}[\neg(w_{l,l'}/4 \leq W_{l,l'} \leq w_{l,l'})] \leq 2e^{-(l'+1)2^l/3}. \tag{C}$$

If $W_{l,l'} \leq 2(l'+1)2^l2^I\tau$ then $w_{l,l'} = W_{l,l'}$ by the definition of $w_{l,l'}$. Otherwise we normalize the running times of scheduled jobs and $W = W_{l,l'}$ by dividing them by $2^I\tau$, set $r = r_{l,l'}$, $\lambda = 2(l'+1)2^l$, and $\mathbf{k} = k_{l,l'}$, and we get exactly the situation described in the assumptions of Lemma 3.2. The statement of (C) now follows from Lemma 3.2 by definition of $w_{l,l'} = 2r_{l,l'}\lambda 2^I\tau/k_{l,l'}$.

Now, we set $\alpha = e^{-8/3} \approx 0.0695$ and sum the bounds over all l and l',

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

The statement for w_l is a trivial consequence. $\square$

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

Since different classes can finish at different times (within a factor of 4), the exact statement is somewhat tedious. One technical issue is that our estimates include the jobs that have already been scheduled during the sampling step. However, since the length of the sampling step is bounded by a constant multiple of the longest running time, we can "credit" this work to step (7). Formally, instead of proving that the time when the efficiency is low is short, we prove that the length of the schedule is bounded by a constant multiple of the work divided by the number of processors, plus an additive term bounded by a constant multiple of the longest running time. The next claim analyzes one pass through steps (4) to (8) in this way and also proves that if the estimates from the sampling are good, step (7) schedules all jobs or ends by finding a job longer than $2^I\tau$.

Claim 3.4 *(i) The total time T of a pass through steps (4) to (8) is bounded by $T \leq (4+\frac{4}{47})W'/n^2 + 4\cdot 2^I\tau$.*

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

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

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

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

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

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

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

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

Now it is easy to prove Theorem 3.1. We only need to take care of the fact that the sampling step can be repeated several times and of the doubling of the estimate on the longest running time. One subtle observation is that if the running time is larger than the current estimate, then all the previous claims still work until we actually see a long job. This is justified by the following mental experiment: replace all the long jobs by jobs whose length is equal to the current estimate; until the moment we see a job running longer than the estimate, the algorithm behaves exactly the same way on both instances, and hence all the bounds must be true as well.

Proof of Theorem 3.1. First we bound the total length T_i of steps (3) to (8) during which $I = i$. Let W_i'' denote the total work done during that part of schedule. Step (3) takes time at most $2^I\tau$. According to Claim 3.3, if no job longer than $2^I\tau$ is found, the probability that steps (4) to (8) are repeated is less than $1/6$. Therefore the expected number of passes through steps (4) to (8) is $6/5$ and hence using Claim 3.4 we get

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

Let W'' be the total work of all jobs. Since the optimal schedule has to perform all the work, we have $W''/n^2 \leq T_{\text{opt}}(\mathcal{J})$. Let i' be the maximal value of I during the algorithm. Obviously the longest running time of a job is at least $2^{i'}\tau/2$, and hence $2^{i'}\tau \leq 2T_{\text{opt}}(\mathcal{J})$, since the optimal algorithm has to schedule the longest job as well.

To bound the length of step (1), we use the fact that during step (1) the efficiency is less than 1 for at most 3τ. Hence, combining this with the bounds on T_i, the total length of the schedule $T(\mathcal{J})$ is bounded by

$$\begin{aligned} \mathrm{E}[T(\mathcal{J})] &\leq \left(4 + \frac{4}{47}\right)\frac{W''}{n^2} + 3\tau + \left(6 - \frac{1}{5}\right)\sum_{i=1}^{i'} 2^i\tau \\ &\leq \left(4 + \frac{4}{47}\right)T_{\text{opt}}(\mathcal{J}) + \left(6 - \frac{1}{5}\right)\sum_{i=0}^{i'} 2^i\tau \\ &\leq \left(4 + \frac{4}{47}\right)T_{\text{opt}}(\mathcal{J}) + \left(6 - \frac{1}{5}\right)2^{i'+1}\tau \\ &\leq \left(4 + \frac{4}{47}\right)T_{\text{opt}}(\mathcal{J}) + 4\left(6 - \frac{1}{5}\right)T_{\text{opt}}(\mathcal{J}) \leq 28T_{\text{opt}}(\mathcal{J}). \end{aligned}$$

$\square$

4 Higher-dimensional meshes

The algorithm SAMPLE can be generalized to higher-dimensional meshes. The ideas behind the algorithm do not change. However, to prove that they still work requires some tedious calculations.

The main result is that the competitive ratio of the generalized randomized algorithm SAMPLE is a constant that does not depend even on the dimension d, if the dimensions of all jobs and of the machine are are powers of two, and there are no large jobs (i.e., no dimension of any job is more than half of the corresponding dimension of the mesh). Even with these restrictions, this result seems to be surprisingly good, especially when compared to the best known deterministic algorithm, where the dependence on d is $O((2d \log d)^d)$

We can weaken these assumptions but only with some loss in the competitive factor. If the large jobs are allowed (but the dimensions are still powers of two) and the machine is a cube, i.e., all the dimensions of the mesh are equal, we can process the large jobs with a loss of only a factor d as follows. We partition them according to the number i of dimensions that are same as the corresponding dimension of the mesh. For each i we process them by as $d-i$ dimensional jobs, disregarding the first i dimensions. If the machine is not a cube, the same process loses a factor up to 2^d. In the most general case when also the dimensions of jobs and of the mesh are not powers of two, we have (in addition to the modifications above) to round up the dimensions of jobs and the total increase of the competitive ratio is by a factor of 4^d.

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

We assume that the dimensions of each job are ordered in descending order. Two jobs are defined to be in the same class if they differ only in the last dimension and in the same subclass if all their dimensions are equal. For $1 \leq l_1 \leq \ldots \leq l_d$, the subclass $\mathcal{J}^{(l_1,\ldots,l_d)}$ contains all jobs of size $m_{l_1} \times \ldots \times m_{l_d}$, where $m_l = n/2^l$. Let $l = l_1 + l_2 + \cdots + l_{d-1}$, $l'_1 = l_1$ and $l'_i = 1 + l_i - l_{i-1}$ for $i > 1$. Note that $l'_i \geq 1$ for all i, and $l \geq d-1$ since we do not allow large jobs.

We have to generalize two parts of the algorithm SAMPLE. First the sampling step (4), in which we now need to obtain estimate for each subclass. Second, the steps (5) to (7) where the mesh is partitioned proportionally to the obtained estimates of work for each class. The rest of the algorithm and proofs is the same as for two-dimensional meshes.

Obtaining many samples at once

First we show how to obtain the estimates for all $O((\log n)^d)$ subclasses at once so that with a constant probability all of them are good.

To a subclass $\mathcal{J}^{(l_1,\dots,l_d)}$ we assign a submesh of size $m_{l'_1}\times\cdots\times m_{l'_{d-1}}\times l'_d m_{l'_d}$. The submesh assigned to $\mathcal{J}^{(l_1,\dots,l_d)}$ is divided into $l'_d 2^{l+1-d}$ of submeshes of size $m_{l_1}\times\cdots\times m_{l_d}$, which means that we schedule $l'_d 2^{l+1-d}$ jobs from $\mathcal{J}^{(l_1,\dots,l_d)}$ in parallel.

It is easy to verify by induction that for every i and $l_1,\dots,l_i$, the submeshes for all $l_{i+1},\dots,l_d$ fit into two meshes of size $l_1\times\cdots\times l_i\times n\times\cdots\times n$, and hence all of the submeshes fit on two copies of the machine. We sample first in parallel for all submeshes on one copy of the machine, then in parallel for all submeshes on the other copy of the machine.

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

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

where α is a small positive constant which we choose later. Similarly to the proof of Claim 3.3, the total error for each class is bounded by

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

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

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

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

Packing the meshes

Now we generalize the steps (5) to (7) of the algorithm SAMPLE, where we schedule the jobs according to the previous estimates of the work of each job class. As before, we assign to each class a volume proportional to its work. This time we partition the whole volume, and prove that even after all rounding it can fit into a constant number of copies of the machine. This is sufficient, since instead of scheduling all jobs in parallel, we can arrange the parts of schedule performed on different copies of the machine sequentially. We can do this even in the case of the on-line randomized algorithm, with no change of the proofs.

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

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

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

Now we derive the estimates on the total volume. Let

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

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

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

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

Hence

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

$$
\begin{aligned}
&= e^{2^{1-j}} F(i-1, j+1) \\
&\leq e^{2^{1-j}} e^{2^{1-j}} 2^{1-i-j} = e^{2^{2-j}} 2^{1-i-j}
\end{aligned}
$$

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

5 Scheduling jobs with dependencies

In this section we prove the following theorem.

Theorem 5.1 *No randomized on-line scheduling algorithm for scheduling of jobs with dependencies on one-dimensional mesh can achieve a better competitive ratio than* $\Omega(\frac{\log N}{\log \log N})$.

As a consequence, the deterministic algorithm for one-dimensional meshes from [3] is within a constant factor of the optimal competitive ratio and even randomization cannot improve it. This is in contrast to scheduling without dependencies on two-dimensional meshes studied in Section 3, where we have shown that randomization can help significantly.

When proving lower bounds on the competitive ratio of a deterministic algorithm, we can predict its actions exactly and take advantage of this in the design of the input for which the competitive ratio is high. In particular, we can set the running times as a response to the actions of the algorithm. However, it is considerably more difficult to prove a lower bound for randomized algorithms. We have to specify the running times, or at least their distribution, in advance, since we cannot predict the random bits of the algorithm.

We design this distribution so that if the algorithm schedules many jobs at once, with large probability the jobs with long running times use only a few processors but make a large fraction of processors unusable, and thus the schedule has to be inefficient. This key technical fact is stated exactly in Lemma 5.2.

An additional technical difficulty is that the on-line algorithm can use virtualization. In the deterministic case we could argue that if any job is scheduled on a small number of processors, we just assign it a long running time. This no longer works, since we have to commit to the distribution of running times beforehand. Thus arguing about a single job is not sufficient. We have to argue that if the algorithm schedules many jobs using virtualization, with high probability one of them has long running time under the distribution of running times that we choose. To make this argument work, we have to set the parameters in our proof very carefully.

The job system and the off-line schedule

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

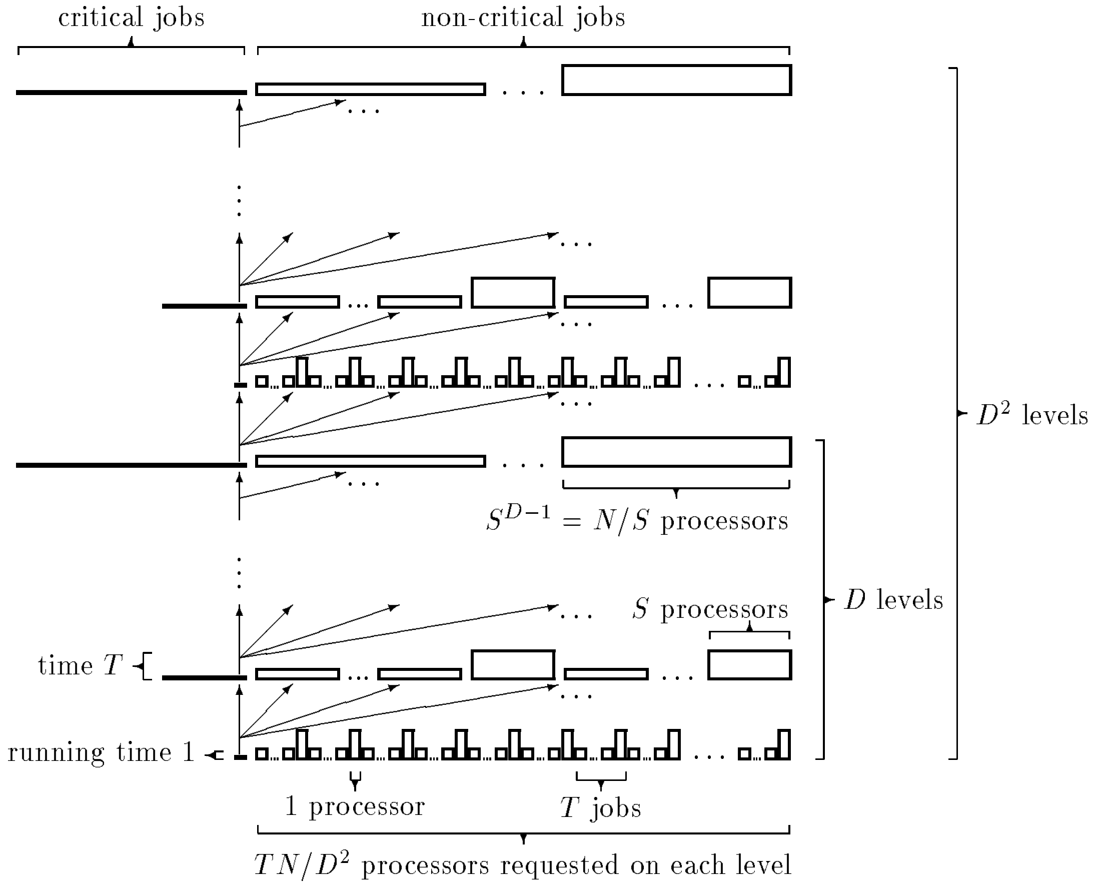

Figure 3: *A typical instance of a job system used in the proof of the lower bound for randomized scheduling on one-dimensional meshes. (The boxes denote the jobs, the vertical dimension is their running time and the horizontal dimension is their size.)*

The job system used in the proof is illustrated on Figure 3. It is a tree of depth D^2. All jobs on level $iD + j$ of the tree for $0 \leq i, j < D$ request S^j processors; there are $TN/D^2S^j + 1$ of them. We assign the running times at random, thus, strictly speaking, we have not a single instance of the problem but a distribution on some subset of instances. For each level the running time is 0 for a single randomly chosen job, and for the other jobs it is T with probability $1/T$ and 1 otherwise. All jobs on a given level depend on the job with running time 0 from the previous level, there are no other dependencies. We call the jobs with running time 0 *critical.*

For each level, the total number of processors requested by non-critical jobs is TN/D^2. Thus the work of the jobs on each level is at least TN/D^2 and the expected work on each level is less than $2TN/D^2$. The optimal schedule first schedules the chain of critical jobs; this takes no time as their running time is 0. The remaining jobs are independent of each other, their total expected work is less than $2TN$ and the longest running time is T. Using the results on scheduling without dependencies from [4] it follows that the expected length of the optimal schedule is $O(T)$.

An overview of the proof

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

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

The on-line algorithm has to schedule the critical job on the given level before it can schedule any jobs on the next level. If the non-critical jobs scheduled before the critical job on the given level are scheduled on a contiguous segment of the mesh, we expect that on most segments T times larger than the size of a job there is at least one job with running time T. In that case the segment cannot be used for time T for scheduling jobs at least S times larger. Lemma 5.2 proves a similar statement for a general case when the jobs are not necessarily scheduled on one contiguous segment of the mesh.

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

These arguments get more complicated if the algorithm uses virtualization. We no longer can completely avoid scheduling jobs in a space which is intuitively unusable, since with virtualization it is always possible schedule a large job on an arbitrarily small segment. However, if this happens too often, with large probability one of the jobs that use virtualization has long running time. Our parameters are carefully chosen so that this argument is sufficient; in fact, this is the main reason why S is as large as D^{12}, without virtualization a smaller power would be sufficient.

Simplifying assumptions about the on-line algorithm

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

We divide the schedule into phases and subphases as follows. The *subphase* j of the *phase* i, $0 \leq i, j < D$, is the time interval during which the first level with an unscheduled critical job is level $iD + j$ (for convenience we number the levels, phases and subphases from 0). We use the phrase *just before the current subphase* to refer to the beginning of the subphase if it is the subphase 0 of any phase, or to the end of the previous subphase otherwise.

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

Since the jobs on each level are ordered randomly with uniform distribution, no matter what the actions of the randomized algorithm are, the jobs from each level are scheduled in a random order. Since we average over all instances of the job system, it is equivalent to assume that the running times of the jobs are assigned as follows. At the beginning of each subphase we decide randomly the position in which the critical job on that level will be scheduled. Then whenever a job is scheduled, if it has the correct position, it has running time 0. Otherwise we randomly assign its running time to be T with probability $1/T$ or 1 otherwise. We make the random decision about the running time of a non-critical job only at the moment when that job would finish if its running time were 1 or at the end of the subphase. (Note that the actual running time can be increased by virtualization.) Since the algorithm is on-line, it does not know the running times of jobs until they finish, and thus this delay of the decision does not change its actions. All random choices that we make are independent. Of course, the algorithm can make some additional random choices. We call a non-critical job *undecided* if we did not yet assign its running time.

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

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

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

The measure of progress

We say that a processor is *used* if it is assigned to a job with running time T scheduled during the current phase. During the subphase j of any phase we say that a processor is *blocked* if the length of the largest segment of unused processors containing it is at most D^2TS^j (a used processor is considered blocked). The *blocked space* is defined to be the number of blocked processors.

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

Since the algorithm is randomized, we cannot ensure that some fixed amount of space is always blocked, as opposed to the deterministic proof in [3]. Instead, we measure the progress by the expected sum of the blocked space and the length of the schedule. This measure is very much like a potential function in amortized analysis. From the overview of the proof it follows that $1/D$ fraction of the space should have approximately the same weight as a time interval of length T/D.

Formally we define the *waste* at a given time to be the sum of the blocked space divided by N/D plus the current length of the schedule divided by $T/100D$. We measure the waste in *units*; one unit of waste corresponds to N/D of the blocked space or to time $T/100D$ in the length of the schedule. The *increase of the waste* is the difference between the current waste and the waste just before the current subphase. Note that the waste at the beginning of a subphase and at the end of the previous subphase can be different, since blocked space is defined in the context of the current subphase. The waste can decrease only at the beginning of any phase or after time T of any phase.

Estimating the progress

Lemma 5.2 *Suppose that the time since the beginning of the current phase is less than T. If at least $23N/D$ processors are assigned to undecided jobs, then the expected increase of the waste at the end of the current subphase is at least least 2 units.*

Proof. It is sufficient to prove that the expected number of processors that were not blocked just before the current subphase but are blocked after the running times of the undecided jobs are assigned is at least $2N/D$.

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

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

Now we show that there are many marked segments. It then follows that a constant fraction of them lies between two marked segments that are sufficiently close, and hence on average a constant fraction of marked segments will be blocked.

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

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

Proof of Theorem 5.1. First we prove that the expected increase of the waste at the end of each subphase is at least 2 units as long as the length of the current phase is at most T. If at any time during a given subphase the number of processors assigned to undecided jobs is more than $23N/D$, the expected increase of the waste is high by Lemma 5.2. If this is not true at any point of the subphase, we prove that the expected length of the subphase is at least $T/50D$.

At any time at most $23DN/S^j$ undecided jobs are scheduled on any $23N/D$ processors: by the second assumption about virtualization no T undecided jobs are scheduled on $o(1/D)$ fraction of the number of processors they request, hence those $23DN/S^j$ undecided jobs would have to be scheduled on at least $\Omega((23DN/S^j - T)S^j/D) = \Omega(N)$ processors. The expected number of non-critical jobs scheduled before the critical one is $TN/2D^2S^j$. By the previous argument only $23DN/S^j$ of them can be assigned running time at the end of the subphase. Thus the expected total work done by all jobs on the given level while they are undecided is at least $TN/2D^2 - 23DN \geq (1 - o(1))TN/2D^2$. Since undecided jobs are scheduled on at most $23N/D$ processors, for a sufficiently large N it takes expected time $(1 - o(1))T/46D \geq T/50D$ to perform the required work, which concludes the proof that the expected increase of the waste during the subphase is at least 2 units.

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

This expectation is taken not only over the random bits of the algorithm, but also over the random instances of the job system; therefore we still have to argue that this proves that the competitive ratio is at least $\Omega(D)$. Suppose that the competitive ratio is $o(D)$. Then for each instance of the job system the expected length of the on-line schedule is at most $o(D)$ time the length of the optimal schedule. Averaging over all instances we get that the expected length of the on-line schedule is $o(DT)$, since the average length of the optimal schedule is $O(T)$. This contradiction finishes the proof. $\square$

Acknowledgement

I am grateful to Anja Feldmann for many valuable comments on the early versions of these results.

References

[1] N. Alon, J. H. Spencer, and P. Erdős. *The Probabilistic Method.* Wiley, 1992.

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

[3] A. Feldmann, M.-Y. Kao, J. Sgall, and S.-H. Teng. Optimal online scheduling of parallel jobs with dependencies. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 642–651. ACM, 1993. Also appeared as technical report CMU-CS-92-189, Carnegie-Mellon University, 1992.

[4] A. Feldmann, J. Sgall, and S.-H. Teng. Dynamic scheduling on parallel machines. *Theoretical Computer Science*, 130(1):49–72, 1994.

[5] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, 1991.

[6] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33:305–308, 1990.

[7] W. Hoeffding. Probability inequalities for sums of bounded random variables. *American Statistical Association Journal*, pages 13–29, Mar. 1963.

[8] C. Lund and N. Reingold. Linear programs for randomized on-line algorithms. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 328–391. ACM-SIAM, 1994.

[9] L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.

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

[11] J. Sgall. Randomized on-line scheduling of parallel jobs. In *Proceedings of the 3rd Israel Symposium on Theory of Computing and Systems*, pages 241–250. IEEE, 1995.