Resource scheduling for parallel database and scientific applications

Soumen Chakrabarti[*] S. Muthukrishnan[†]

Abstract

We initiate a study of resource scheduling problems in parallel database and scientific applications. Based on this study we formulate a problem. In our formulation, jobs specify their running times and amounts of a fixed number of other resources (like memory, IO) they need. The resource-time trade-off may be fundamentally different for different resource types. The processor resource is *malleable*, meaning we can trade processors for time gracefully. Other resources may not be malleable. One way to model them is to assume *no* malleability: the entire requirement of those resources has to be reserved for a job to begin execution, and no smaller quantity is acceptable. The jobs also have precedences amongst them; in our applications, the precedence structure may be restricted to being a collection of trees or series-parallel graphs.

Not much is known about considering precedence and non-malleable resource constraints together. For many other problems, it has been possible to find schedules whose length match to a constant factor the sum of two obvious lower bounds: the total *resource-time product* of jobs, denoted V, and the *critical path* in the precedence graph, denoted Π. We show that there are instances when the optimal makespan is $\Omega(V + \Pi \log T)$ in our model. Here T is the ratio between longest and shortest job execution times, where typically $T \ll n$, the number of jobs.

We then give a polynomial time makespan algorithm that produces a schedule of length $O(V + \Pi \log T)$, which is therefore an $O(\log T)$ approximation. This contrasts with most existing solutions for this problem, which are greedy, list-based strategies. These fail under heavy load and that is provably unavoidable since theoretical results have established various adversaries for them that force $\Omega(T)$ or $\Omega(n)$ approximations. The makespan algorithm can be extended to minimize the weighted average completion time over all the jobs to the same approximation factor of $O(\log T)$.

1 Introduction

Judicious job scheduling is crucial to obtaining fast response and effective utilization. Consequently, algorithms for scheduling have been extensively researched since the 60's both in theory and in practice. In the past decade, there has been increasing interest in scheduling specifically for parallel systems. Owing to the vastly different nature and applicability of sequential and parallel computing systems, the scheduling problems of practical interest are rather different in the two settings. For example, scheduling general-purpose jobs is a reality for a uniprocessor operating systems while it is hardly an issue yet for existing parallel systems; similarly, scheduling threads for massive game tree searches is a reality for parallel systems while that is hardly an issue for uniprocessor traversal.

In this paper, our concern lies in scheduling problems that are critical to, and prevalent in, practical parallel computing. We first need to model parallel computing scenarios and then identify relevant problems. For modeling, we focus on applications where parallelism has been effectively exploited. We isolate parallel databases and scientific computing as two such areas. In both these areas the computation is well-structured so regular subproblems can be solved in parallel, and the resource requirements of jobs can be estimated well. For these reasons, these remain the two areas where parallel computing has proven highly successful. Here, we carefully model the resource scheduling problems in these applications and study them.

In the model we abstract from those applications, jobs arrive over time to a central manager; they have precedence constraints and resource requirements. Resources are of various different types. Some resources, like processors, can be traded for time gracefully. Others, like memory, are not flexible in such a continuous fashion. A survey of practical systems reveals that they typically use simple greedy approaches that may perform poorly under heavy load. This is not surprising since theoretical results have produced appropriate adversaries for many such situations. On the other hand, no algorithmic result to our knowledge completely addresses our real-life setting, in particular, the combined complication of precedence *and* non-malleable resource constraints. However, reported experience with parallel databases suggests that this is an important problem.

We initiate a study of such resource scheduling models. Our main technical contribution is the first known logarithmic approximation algorithm for this scheduling problem under two metrics, namely, the classical makespan, and the weighted average completion time (WACT), which has seen more attention recently. We suggest a number of directions for extending our preliminary results.

1.1 Model and problem statement

We describe the modeling scenarios in parallel databases and scientific computation later on (§2). The model and problems we abstract from there are as follows.

Model. Our model consists of m identical processors and s types of other resources not including the processors. The unit of computation is a job: job j specifies a number of parameters: $\vec{r}_j = (r_{j1}, \dots, r_{jk}, \dots, r_{js})$, of the fractions r_{jk} of resource of type k demanded by job j, $k = 1, \dots, s$; the maximum number of processors m_j that may be allotted to job j; $t_j \geq 1$, the running time for job j on m_j processors provided all the resource demands are met; and w_j, the weight of j which reflects its priority. In addition, there is a precedence relation $\prec$ amongst the jobs, that is, for each

[*]Computer Science Division, U. C. Berkeley, CA 94720. Partly supported by ARPA/DOD (DABT63-92-C-0026), DOE (DE-FG03-94ER25206), and NSF (CCR-9210260, CDA-8722788 and CDA-9401156). Email: soumen@cs.berkeley.edu.

[†]University of Warwick, UK. Supported in part by ALCOM-IT Email: muthu@dcs.warwick.ac.uk.

job j, there exists a possibly empty subset of jobs that must have completed before job j begins its execution. This can be modeled as usual by a directed acyclic graph (DAG). Processing resource is assumed to be perfectly *malleable*, that is, if job j is allotted μ_j processors, $1 \leq \mu_j \leq m_j$, (together with all the other resources it requires) then it takes time $m_j t_j / \mu_j$ (i.e , j gives linear speedup up to m_j processors). Other resources are assumed to be *non-malleable*, that is, when job j is scheduled to run, r_{jk} fraction of resource k must exclusively be set aside for it. For notational simplicity we scale the total available resource to be 1 unit for each type.

Problem. A *scheduler* is an algorithm that specifies for each processor and time slot, a portion of at most one job to be executed, so that all jobs get executed subject to all the constraints above. Consider a set of n jobs. Let a scheduler complete job j at time C_j. The two performance metrics that we seek to minimize are the *makespan* $\max_j C_j$ and the *weighted average completion time* (WACT) $\frac{1}{n}\sum_j w_j C_j$.

The applications we model impose further constraints on the schedulers. First, *preemption* is not allowed. Time-slicing and preemption of space-shared resources is very expensive because (1) the state has to be evicted to slower layers of the memory hierarchy, (2) processes have to synchronize and switch across protection domains, and (3) in-flight messages have to be flushed and reinjected [6, page 6]. Anecdotal evidence suggests that many practitioners switch off time sharing for production runs on parallel machines that do not permit creating dedicated space partitions. To recover from pathological cases, some machines (like the Cray T3D) have an expensive roll-in/out mechanism, but it is important to minimize its deployment. Second, the number of *types* of non-malleable resources, s, can be assumed to be a small constant. Since we target symmetric shared memory multiprocessors, resources are not private to processors; they are equally accessible by all jobs. Typically the set of resources will be memory, disk and network bandwidth (i.e., $s = 3$). Third, it suffices for our scheduler to be a sequential algorithm since in both databases and OS's these decisions are typically made by a front-end processor connected to clients. Fourth, the precedence graph may be thought of as a collection of trees or series parallel graphs only; these are the most prevalent instances in applications (see §2). Finally, the scheduler can collect jobs over some time window but cannot wait for all jobs to be submitted, as jobs arrive on-line over time. Moreover, each arriving job specifies only its predecessors at the time of its arrival, but not, e.g., successors.

1.2 Known approaches

Many simplified variants of our problem are strongly $\mathcal{NP}$-hard, even for makespan. Thus the goal is to find approximations for the worst case and heuristics in practical settings.

All known practical solutions use some variant of greedy list- or queue-type scheduling [6, 9, 18]. Jobs on arrival are placed in a list ordered by some heuristic (often FIFO). The scheduler dispatches the first ready job on the list when enough resources become available List scheduling and its variants are appealingly simple to implement; however they can be notoriously bad. In fact, theoretical results show that if precedence is combined with non-malleable resource constraints, there exist sequences of jobs such that the scheduler can be arbitrarily compromised. No deterministic scheduler that does not know t_j until job j has finished[1] has WACT performance ratio bounded away from T [20, Theorem 5.4]. Even if t_j is known upon arrival, any greedy schedule has worst case makespan performance ratio at least $nT/(n+T)$, which is roughly T when $T \ll n$ and n when $T \gg n$, both of which are achieved trivially [8, Theorem 1]. On the other hand, all positive algorithmic results we tried to adapt fall short of our goal in some respect or another, unless we abandon some requirement. In what follows we show examples of this.

[1]Such a scheduler is called "non-clairvoyant"

No precedence. If the precedence graph were empty, (i.e., independent jobs) then a number of approaches are known to get approximation bounds, even with additional restrictions like sub-linear speedup [8, 17, 26, 25, 24]. In the database scenario, it is possible to collapse each query (consisting of several jobs with a precedence relation amongst them) into one job (allocating maximum resources over all the jobs in the query) and then apply the results for independent jobs [27]. This has a serious drawback in that some obvious, critical co-scheduling may be lost. For example, a CPU-bound job from one query and the IO-bound job of another can be co-scheduled and it is highly desirable to do so [14]; this cannot be done after the collapse.

Only malleable resource. If there were no non-malleable resources, (i.e., there is only a malleable resource), then precedence can be handled (in the sense of approximating makespan) even by a scheduler that does not know t_j before a job finishes [7]. But we must also schedule non-malleable resources.

Small resource demand. Another possible restriction is to allow non-malleable resources, but require each job to reserve no more than λ fraction of the non-malleable resources, where λ is small. That is, the maximum fraction requested by an job is at most λ. In that case naive approaches will work well. The approximation ratio for greedy scheduling can be easily shown to be $1 + \frac{1}{1-\lambda}$. This is however useless even if one, or a *few* jobs need a large fraction, or equivalently, if λ approaches 1. In fact, $\lambda = 1 - O(1/m)$ suffices to render the greedy schedule useless. Intuitively, a few resource-intensive jobs can delay a convoy of tiny jobs.

Therefore, using known results off-the-shelf from the theory literature for our formulation means we have to compromise heavily. See Table 1 for a succinct comparison with some settings and results close to ours.

1.3 Our contributions

Our contribution is twofold. First, we initiate a study of resource scheduling problems in parallel databases and scientific computing, and abstract a problem formulation. While the abstraction does ignore some details, we show that it is already different from the body of known theory results. On the other hand, known practical solutions have poor worst-case behavior. Our second contribution is that we design new scheduling algorithms with good guaranteed worst case performances. We describe the latter results now.

Essentially all known scheduling results with precedence depend on two bulk parameters of the input instance: the *volume* and the *critical path*. The volume V is the total resource-time product over all jobs; the critical path Π is the earliest possible completion time assuming unlimited resources. Clearly $\Omega(V + \Pi)$ is then a lower bound to makespan, and in most existing settings this can be achieved to a constant factor. We show that our formulation is very different in flavor, in that some parameter other than V and Π is at work. Specifically, we give instances of our prob-

Reference	# procs	Job type	Mall	Non-Mall	Prec $\prec$	On-line/ off-line	Makespan / WACT
Garey *et al* [8]	N/A	N/A	$\times$	$\surd$	$\emptyset$	on	makespan
Feldmann *et al* [7]	m	par	$\surd$	$\times$	any	on	makespan
Hall *et al* [13]	$1, m$	seq	$\times$	$\times$	any	off	both
	$1, m$	seq	$\times$	$\times$	$\emptyset$	on	both
Chakrabarti *et al* [3]	m	par	$\surd$	$\times$	any	on	both
	m	par	$\times$	$\surd$	$\emptyset$	on	both
This paper	m	par	$\surd$	$\surd$	any	on	makespan
	m	par	$\surd$	$\surd$	any	off	WACT
	m	par	$\surd$	$\surd$	forest/SPG	on	both

Table 1: Comparison of results. N/A=not applicable, seq=sequential, par=parallel, Mall=malleable resource, Non-Mall=non-malleable resource, SPG=series-parallel graph.

lem where no schedule can achieve a makespan smaller than $\Omega(V + \Pi \log T)$, where T is the ratio $\max_j t_j / \min_j t_j$.

For this choice of parameters (V, Π and T), we give a simple approximation algorithm that matches the $O(V + \Pi \log T)$ makespan bound. The makespan problem can be extended in many different ways (§4). In particular we use it as a subroutine to give the first polynomial-time scheduler that, for all instances of our problem, determines a schedule with WACT at most a $O(\log T)$ factor away from optimal in the worst case. (Recall that in our model jobs need $s = O(1)$ distinct types of resource, and the precedence between them are forests or series-parallel graphs). The approach we use to minimize WACT requires another subroutine for solving a generalized packing problem, where the main complication is precedence constraints between items to be packed. This problem appears to be of independent interest. Throughout we have made an effort to keep our algorithms simple. For instance, our algorithms do not use powerful primitives such as linear programming algorithms, and indeed we could not improve the quality of our solution using them. Finally we remark that although our problem is different from existing theoretical settings, our solution borrows from various existing techniques [8, 22, 3, 13].

Outline. In §2 we study database and scientific computing scenarios to justify our model. In §3 and §4 we give the makespan lower and upper bounds. In §5 and §6 we show how to extend the makespan algorithm to a WACT algorithm. In §7 we pose unresolved problems.

2 Model

Databases. Query scheduling in parallel databases is a topic of active research [2, 18, 27, 12, 9]. Queries arrive from many users to a front-end manager process. A query is an in-tree where the internal vertices are operations like sort, merge, select, join etc., which we call jobs. (We think of pipelines as collapsed into single vertices.) The leaves are relations stored on storage devices. Edges represent data transfer; the source vertex sends output to disk, which the target vertex reads later. Queries may have a priority associated with them, e.g., an interactive transaction has high priority and statistics collection has low priority.

Databases keep certain access statistics along with the relations that are used to predict the sizes of results of joins and selects, and how many CPU instructions will be required to compute these results. The tools are standard in database literature [21]. For parallel databases, one can also estimate for each operation up to how many processors can be employed at near-linear speedup [14]. Thus t_j and m_j can be estimated when a job arrives. Estimates of sizes of intermediate results can be used to estimate the memory and disk bandwidth resource vector $\vec{r}_j$.

Unlike for processors, the running time of a job is not roughly inversely proportional to memory. For example, a hash-join between two relations R_1 and R_2 with, say, R_1 being smaller, takes time roughly proportional to $\lceil \log_r |R_1| \rceil$, where r is the memory allocated; typically the query planner picks $r = |R_1|$ or $r = |R_1|^{1/2}$, independent of other queries [18]. Once processor and memory allocation are fixed, the disk bandwidth needed can be estimated from the total IO volume and job running time.

This model is best suited to shared memory databases running on symmetric bus-based multiprocessors (SMP) with shared access to disk [14]. They currently scale to 30–40 processors. There is consensus that SMP's and scalable multiprocessors will converge to networked clusters of SMP nodes [12]. Since communication costs across clusters is much more expensive than shared access within a cluster, the expectation is that most queries will be parallelized within an SMP node.

Scientific applications. Multiprocessor installations are shared by many users submitting programs to manager processes running on the front-end that schedules them. Examples of front-end schedulers are DJM (distributed job manager) on CM5, NQS (network queuing system) on Paragon, POE (parallel operating environment) on SP2. In some machines, the parallel job scheduler is closely integrated with the OS, such as IRIX. Users may submit a script to the manager. A script file has a sequence of invocations of executables, each line typically specifying a priority, the number of processors, estimated memory and running time. Notice that although there may be some flexibility in the amount of memory needed, the user typically makes a choice before specifying an amount to the scheduler, which has to regard it as inflexible. To improve utilization, system support has been designed to express jobs at a finer level inside an application and convey the information to the resource manager by annotating the parallel executable [11, 5, 19].

Note that the common precedence graphs are chains for scripts, series-parallel graphs for structured programs, forests for database queries, and for divide-and-conquer and branch-and-bound.

Fidelity. The most general representation of a job is its running time as a function of the resource allocated to it. It is difficult to find this function [9], and unclear if it is simple enough to be used by the optimizer. We propose grouping the resources into two types: malleable and non-malleable. It may be of interest to evaluate more elaborate alternatives. Assuming knowledge of t_j is reasonable for our applications, but for more unpredictable applications, adaptive scheduling is needed.

We have avoided both unwarranted simplifications and generality. For example, we have not assumed for simplicity that all resources are malleable. Memory is clearly not mal-

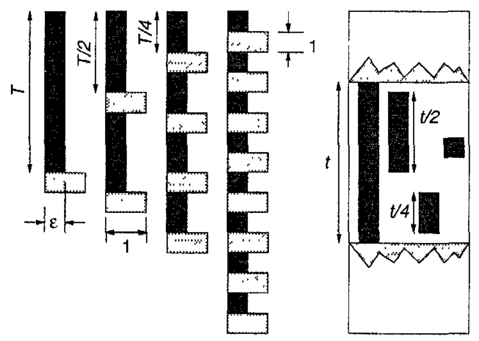

Figure 1: Illustration of the lower bound.

leable, even for programs that are adaptive in limited ways to the amount of allocated memory. Similarly, we have not modeled the details of the inter-processor network.

3 Makespan lower bound

Many results in DAG scheduling rely on two bulk parameters of the workload. One, which we call the *volume* parameter, is the sum of resource-time products over jobs in the graph, denoted V For example, with only one resource type, $V = \sum_j t_j r_j$. Note that for malleable jobs $r_j = m_j/m$ and for Graham's list-scheduling type situation $r_j = 1/m$ for all j. The other parameter is the critical path Π of the graph. $\Pi = \max_j \Pi_j$, where Π_j is the minimum time at which j can complete even if infinite resources were available.

$\Omega(V + \Pi)$ is a lower bound to makespan. In all occasions without non-malleable resources, this lower bound can be achieved to a constant factor: $\Pi + \frac{1}{m}\sum_j t_j = \Pi + V$ for sequential jobs [10], and $\Pi + (\frac{\sqrt{5}-1}{3-\sqrt{5}})\frac{1}{m}\sum_j t_j m_j = \Pi + O(V)$ for malleable jobs [7]. These depend on arguments of the form: "if a critical path is being ignored, most of the resources are being utilized."

In this section, we show that $O(V + \Pi)$ makespan is not always possible with both non-malleable resources and precedence constraints. Thus this differentiates our problem from all list-type approaches [10, 8, 13] and malleable job scheduling [7]. Our formulation is very different in flavor because jobs may have to wait even when resource utilization is very low because their specific requirement of the non-malleable resources are not met. In contrast, jobs can proceed with a smaller amount of a malleable resource with proportionate slowdown. Thus some parameter other than V and Π is at work.

Claim 1 *With precedence and non-malleable resources, the optimal makespan can be as large as $\Omega(V + \Pi \log T)$, where $T = t_{\max}/t_{\min}$.*

Proof. Consider the following instance with one resource dimension. There are $\Theta(\log T)$ independent job chains. Chain i, $i = 0, 1, \ldots$ has 2^i sequential compositions of the following two-job chain: the *tall* predecessor has $t_j = T/2^i$ and $r_j = \epsilon$, and the *fat* successor has $t_j = r_j = 1$. Then it can be verified that $\Pi = O(T)$, and that $V = \sum_i 2^i(\epsilon T/2^i + 1) \le \epsilon T \log T + 2T = O(T)$ when we choose $\epsilon \le 1/\log T$.

Note that only tall jobs can run concurrently; fat jobs cannot run concurrently with each other or with tall jobs. Also note that the total length of tall jobs is $\Theta(T \log T)$.

We will account for the makespan of an optimal schedule by iteratively picking the currently largest job j in the schedule, say of length $t_j = T/2^x$. No larger jobs are present at this point. Thus this job only overlaps with smaller jobs, but at most one of each distinct length. Remove j and all these jobs from consideration. This reduces the makespan by $T/2^x$ and the total remaining tall job length by at most $T(1/2^x + 1/2^{x+1} + \cdots + 1) \le 2T/2^x$. Note that the fat jobs need to be only over $1 - 1/\log T$ wide in the resource dimension. ∎

This implies that a recent elegant constant factor WACT approximation technique due to Chekuri *et al*, which converts uniprocessor schedules to multiprocessor schedules [4], will not generalize to handle non-malleable resources. Their technique is to start with a uniprocessor schedule where job j completes at time C_j^1 and derive an m-machine schedule with $C_j^m = O(C_j^1/m + \Pi_j)$. In their model all jobs are sequential and the jobs need no other resources. If we additionally add jobs that request non-malleable resources, their conversion does not go through, and with good reasons: if it does, a schedule with $O(\Pi)$ additive term, rather than the $\Omega(\Pi \log T)$ term that we showed above, will hold, thus violating the lower bound.

4 Makespan upper bound

In this section we will give an algorithm Makespan that is polynomial in s, T and n that will achieve a makespan of $O(Vs + \Pi \log T)$ for an input set of jobs J with the bulk parameters V, Π and T defined before, where there is one malleable resource and s types of non-malleable resources. Denote $\vec{v}_j = t_j \vec{r}_j$, and let $V = \max\{\frac{1}{m}\sum_j m_j t_j, \|\sum_j \vec{v}_j\|_\infty\}$. Also let Π_j be the critical path length from a root through job j, and $\Pi = \max_j\{\Pi_j\}$.

The makespan algorithm first invokes a malleable list-type scheduling algorithm [7] to allocate processors to jobs in J, and to assign (infeasible) preliminary execution intervals to these jobs. Then we partition the jobs into a sequence of *layers* with jobs within a layer being independent of each other. Finally we schedule these layers one by one using bin-packing.

Step 1. Let $\frac{1}{2} < \gamma < 1$ be a free parameter to be set later. Compute a greedy schedule for J ignoring all non-malleable resource requirements, as follows. Whenever there are more than γm free processors, schedule any job j in J (whose predecessors have all completed) on the minimum of m_j and the number of free processors. This step is similar to [7].

Denote by μ_j the number of processors allocated to job j. After processor allocation let the modified job times be $t'_j = t_j m_j/\mu_j$, and modified critical path lengths be Π'_j. At this stage the non-malleable resource requirements are still violated in general.

Step 2. Round all job times to powers of two: first scale up the time axis by a factor of two, then round down each job to a power of two. Don't shift jobs so all precedences are still satisfied. We will show that after this step, for all j, $t'_j = O(t_j)$ and $\Pi'_j = O(\Pi_j)$. For notational convenience we will continue to refer to the modified quantities as t_j and Π_j, and assume that the modified time t'_j is a power of two, with $\min_j t'_j = 1$ and $\max_j t'_j = T$, all this affecting only constants.

Step 3. Let $B_\tau = \{j : \tau T \le \Pi_j - t_j < (\tau+1)T\}$, i.e., divide the earliest possible start times of jobs into blocks of length T. Each block of length T will be expanded to a sequence of layers of total length $O(T \log T)$. Specifically, remove from

B_τ all jobs of length T and schedule them in a layer of length T. This is the last layer. Divide $[\tau T, (\tau+1)T)$ into $[\tau T, (\tau+\frac{1}{2})T)$ and $[(\tau+\frac{1}{2})T, (\tau+1)T)$ and recurse, placing the generated layers in pre-order [22]. The total length of the invalid schedule will be $O(\Pi \log T)$.

Step 4. Schedule each layer separately in time order. Consider each layer to be an instance of a generalized s-dimensional bin-packing problem[2] [8]. A first-fit (FF) bin packing of each layer, considering job j in the layer to be an item of "size" $\vec{r}_j$ and bins of "size" $\vec{1}$, suffices for our purpose.

Lemma 2 *After Step 2, the modified times and critical paths obey $t'_j = O(t_j)$ and $\Pi'_j = O(\Pi_j)$. Furthermore the length of the schedule is $O(\frac{1}{m}\sum_j m_j t_j + \Pi)$.*

Proof. (Sketch) The former claim follows because each job j is assigned either m_j machines, in which case $t'_j = t_j$, or at least γm machines, in which case $t'_j \le t_j/\gamma$. Picking γ such that $(1-\gamma)(1+\frac{1}{\gamma}) = 1$, the length of the (invalid) schedule is at most $\Pi + \frac{1}{m\gamma}\sum_j m_j t_j$, similar to [7]. ■

Lemma 3 *Jobs assigned to a particular layer I are independent; the layer ordering is consistent with job precedence $\prec$ and in any layer I, $\sum_{j\in I}\mu_j \le m$.*

Proof. At every stage in the recursion, consider the jobs of length t removed from the block of length t (and put in a separate layer of length t). Any pair of such jobs must have overlapping "execution" intervals after the first step. They must therefore have been independent, and the sum of processors assigned to them was at most m in the first step. A job of length t starting in a block of length t can only have (smaller than t) predecessors starting in the same block and no successors starting in the same block. When the block is bisected, these predecessors are all completed before any of the t-long jobs in this block starts. ■

Note moreover that every job j placed in a layer I of length $t(I)$ has $t_j = \Omega(t(I))$. We will need the following observation which follows from a pigeon-hole argument.

Lemma 4 *For any set $\{\vec{v}\}$ of s-dimensional vectors, $\|\Sigma\vec{v}\|_\infty \ge \frac{1}{s}\Sigma\|\vec{v}\|_\infty$.*

Theorem 5 *For s resource dimensions the above algorithm obtains a makespan of $O(Vs + \Pi\log T)$.*

Proof. Consider a layer I that is $t(I)$-long in time, and generates $f(I)+1$ bins using FF. Then there is at most one bin that is less than half-full in all s dimensions. Each of the other $f(I)$ bins are at least half-full in at least one dimension. Call these bins $F_{1/2}(I)$. For a bin b define $\vec{v}_b = \sum_{j\in b}\vec{v}_j$. Then for all $b \in F_{1/2}(I)$, $\|\vec{v}_b\|_\infty \ge \frac{1}{4}t(I)$. Hence we obtain $V \ge \left\|\sum_b \vec{v}_b\right\|_\infty \ge \frac{1}{s}\sum_b\|\vec{v}_b\|_\infty \ge \frac{1}{s}\sum_I\sum_{b\in F_{1/2}(I)}\|\vec{v}_b\|_\infty \ge \frac{1}{4s}\sum_I f(I)t(I)$. The total length of the schedule is thus at most $\sum_I t(I)(f(I)+1) \le 4Vs + O(\Pi\log T)$. ■

[2]However, this problem is not one where items are solid blocks with volume and the bin is a hollow unit cube

Observe that the makespan routine is polynomial in n, T, and s, and works for any precedence. The above method also gives an alternative algorithm and much simpler analysis (weaker only in a constant) for the $(s+1)$-approximate resource constrained scheduling result of [8]. Their $(s+1)$-approximation is for $\prec = \emptyset$. In this case $\Pi = T$, and we allocate $\log T$ layers with $t(I) \in \{1,2,4,\ldots,T\}$. Then $\sum_I t(I) < 2T$, giving an $O(s)$ approximation.

Corollary 6 *If $\prec = \emptyset$, the above analysis (with a trivial Step 3) gives a schedule of length $O(Vs + T)$.*

5 Weighted average completion time

In this section we will describe how to extend the makespan algorithm developed earlier to the WACT metric, by applying techniques similar to [13, 3]. First we divide time into geometrically increasing intervals. That is, define $\tau_0 = 1$, $\tau_\ell = 2^{\ell-1}$, $\ell = 1,\ldots$. In what follows, consider the ℓth interval in time, namely, $(\tau_{\ell-1}, \tau_\ell]$; other intervals are processed similarly.

Step 1. Let J_ℓ be the set of jobs that have arrived within time $[1, \tau_\ell]$ but have not already been scheduled. We remove from consideration any job which cannot be scheduled within the ℓ-th interval because of critical path constraints. Compute the earliest possible finish time Π_j of each job j based on critical path (assuming unlimited resource and processors). Any job j for which $\Pi_j > \tau_\ell$ is removed from J_ℓ and deferred to a later interval.

Step 2. In this step, we choose a suitably significant subset of J_ℓ to be scheduled, carrying a possibly empty remainder forward to the subsequent intervals.

The subset of J_ℓ chosen for scheduling is described as follows. From J_ℓ we have to pick a subset J^*_ℓ such that $\sum_{j\in J^*_\ell} t_j m_j \le m\tau_\ell$, $\sum_{j\in J^*_\ell} t_j\vec{r}_j \le \tau_\ell\vec{1}$, $j_1 \prec j_2$ and $j_2 \in J^*_\ell$ implies $j_1 \in J^*_\ell$, and the objective $w(J^*_\ell) = \sum_{j\in J^*_\ell} w_j$ is maximized. Suppose the optimal value of the objective above is W^*_ℓ. Since the above problem is $\mathcal{NP}$-hard, we will instead obtain a subset J'_ℓ with value at least W^*_ℓ closed under $\prec$, and satisfying $\sum_{j\in J'_\ell} t_j m_j = O(m\tau_\ell)$, and $\sum_{j\in J'_\ell} t_j\vec{r}_j \le O(\tau_\ell)\vec{1}$. We will describe this procedure, called DualPack, later. We postpone the jobs in $J_\ell \setminus J'_\ell$ to a later interval.

Step 3. In this step we schedule the jobs in J'_ℓ using the Makespan subroutine in §4 which will find a schedule of length $O(\tau_\ell \log T)$. We schedule the output of Makespan in the interval $[\kappa\tau_\ell\log T, \kappa\tau_{\ell+1}\log T)$ for some fixed constant κ determined by the O in the makespan algorithm.

That completes the description of the processing for the ℓth interval. The steps at the high level are standard, for example, see [13, 3]. The technical crux is the design of the two subroutines DualPack and Makespan.

Theorem 7 *The above algorithm is polynomial in T and n and, for $s = O(1)$, is an $O(\log T)$ approximation for both makespan and WACT with on-line job arrival.*

Proof. (Sketch) Consider J_ℓ and J'_ℓ as in the algorithm. Let $w(J'_\ell)$ be weight of the jobs in J'_ℓ. Say a optimal schedule completes within time τ_L, finishing job j at time C^*_j, and let the total weight of jobs completing in $(\tau_{\ell-1}, \tau_\ell]$ be W^*_ℓ in that optimal schedule. First, we establish the following dominance property for all $\ell = 1, \ldots, L$, $\sum_{i=1}^{\ell} w(J'_i) \geq \sum_{i=1}^{\ell} W^*_i$. This follows from the design of DualPack.

Now observe that because of the dominance property, the above algorithm too finishes within round L, and also that $\sum_\ell w(J'_\ell) = \sum_\ell W^*_\ell$ Finally, the cost of the the schedule it determines is at most $O(\log T)\sum_{\ell=1}^{L} \tau_{\ell+1} w(J'_\ell) = O(\log T)\sum_{\ell=1}^{L} \tau_{\ell-1} w(J'_\ell) \leq O(\log T)\sum_{\ell=1}^{L} \tau_{\ell-1} W^*_\ell$, which is at most $O(\log T)\sum_j w_j C^*_j$. That completes the argument. ■

Recall that throughout we have assumed $s = O(1)$. In general, the algorithm above can be proved to be an $O(s + \log T)$ approximation. Also, note that if we had an κ approximation algorithm for the Makespan routine, we would have an $O(\kappa)$ approximation for minimizing WACT.

If we are only interested in off-line schedules, we can compute J'_ℓ via rounding an integer program similar to [13], obviating the need for the DualPack routine. We omit the details of the following claim.

Theorem 8 *There is an off-line algorithm, polynomial in s, T and n, that approximates makespan and WACT to an $O(s + \log T)$ multiplicative factor.*

6 Packing jobs with large weight

We describe the DualPack routine used earlier. We are given a set J of items (jobs) j with profit $p(j)$ and cost $c(j)$, where $c()$ and $p()$ extend to sets of items additively. There is a precedence $\prec$ amongst the items. A subset $J' \subseteq J$ is called *closed under* $\prec$ if $j_1 \prec j_2$ and $j_2 \in J'$ imply $j_1 \in J'$.

We consider the following *primal* problem: given a cost bound C, determine a subset J^* closed under $\prec$ such that $c(J^*) \leq C$ and $p(J^*)$ is maximized. Let this maximum profit be $P^*(C)$. This is similar to the classical knapsack problem except that there are additional constraints based on the precedence. In this section we design a subroutine for the following related problem $\mathcal{X}$: given C, determine a subset J' closed under $\prec$ with $p(J') \geq P^*(C)$ and $c(J') = O(C)$. We claim that suffices for the DualPack routine.

Consider the *dual* of the primal problem above: given a target profit P, determine a subset J^* closed under $\prec$ such that $p(J^*) \geq P$ and $c(J^*)$ is minimized. Let this minimum cost be $C^*(P)$.

Our algorithm for problem $\mathcal{X}$ performs a binary search for the unknown value of $P^*(C)$. With each such binary search query, it solves an approximate version of the dual problem above to "correct" its guess. The whole routine is as follows (κ to be fixed later):

```
Initialize $P_{\rm lb} = 0$; $P_{\rm ub} = 1 + \sum_j p(j)$.
While $P_{\rm ub} > P_{\rm lb} + 1$ do
    $P = \lfloor (P_{\rm lb} + P_{\rm ub})/2 \rfloor$
    Call the dual approximation with argument $P$,
        suppose it returns $J'$
    If $c(J') > \kappa C$ then $P_{\rm ub} = P$ else $P_{\rm lb} = P$
Return $J'$.
```

It is clear that if there was an algorithm for the dual problem which returned a subset closed under $\prec$ with cost $O(C^*(P))$, then we could have an algorithm for problem $\mathcal{X}$. It is this dual approximation that we describe below. Note that choosing κ in the program above no less than the constant in the O of the dual approximation will be sufficient.

In describing the dual approximation below, we assume that the resource units are discrete, as is the case in reality. In the preceding sections we had represented the resource requirements as fractions for notational convenience only. For simplicity we only describe the case $s = 1$, i.e., the case of a single resource.

Series-parallel graphs. A series-parallel graph (SPG) can be represented hierarchically using the grammar: a SPG G is a vertex j (which is both the source and the sink), two SPG's $G_{\rm up}$ and $G_{\rm down}$ in series, or $G_{\rm left}$ and $G_{\rm right}$ in parallel.

A *solution* for G is a subgraph of G closed under $\prec$. We set up a dynamic programming table Y, where $Y[G, c]$ is the maximum profit of a solution for G with cost at most c. Thus for our given target profit P the minimum cost is

$$C^* = \min\{c \cdot Y[\text{root}, c] \geq P\}$$

We fill in the array $Y[]$ as follows. At a single vertex $G = \{j\}$, we set

$$Y[G,c] = \begin{cases} p(j), & \text{if } c(j) \leq c \\ 0, & \text{else} \end{cases}$$

At a series node $G = (G_{\rm up}, G_{\rm down})$, we set

$$Y[G,c] = \begin{cases} Y[G_{\rm up}, c], & \text{if } c(G_{\rm up}) > c \\ p(G_{\rm up}) + Y[G_{\rm down}, c - c(G_{\rm up})], & \text{else} \end{cases}$$

At a parallel node $G = (G_{\rm left} || G_{\rm right})$, we set

$$Y[G,c] = \max_{0 \leq c' \leq c} \{Y[G_{\rm left}, c'] + Y[G_{\rm right}, c - c']\}$$

These dynamic programs give pseudo-polynomial time algorithms, and it is routine to then convert them to fully polynomial time approximation schemes (fptas). Furthermore, the solution sets can be retrieved along with the cost/profit values using some additional book-keeping. The details are very similar to designing fptas for standard knapsack [15] and are omitted. We finally note that at the expense of running time growing exponentially in s, all the above can be extended to s resource dimensions.

Forests. These are handled similar to the standard knapsack on trees [15]; we omit the details Loosely speaking hierarchical graphs can be all handled in a similar way, but arbitrary graphs appear harder (§7).

7 Extensions

Finally, we raise several questions regarding extensions of the model and algorithms in this paper.

Tighter bounds. V, Π and T are not the best characterization of an instance, since the optimal makespan is $\Omega(\Pi \log T)$ for some but not all instances. A better characterization of the lower bound and improved algorithms are needed It is not clear how such a characterization can improve the WACT algorithm.

Imperfect malleability. In reality the processor resource is not perfectly malleable, neither are other resources perfectly non-malleable. How important is it to model and optimize for complicated intermediate forms of malleability? For imperfect malleability in the processor dimension [3] gives an WACT algorithm.

Persistent resources. A job may allocate memory mid-way through execution. We cannot model this by a chain of two jobs, since the memory the job was already holding is not released. How can jobs with such persistent resource needs be scheduled?

Non-clairvoyance and preemption. For the motivating applications, reasonable estimates of job running time are possible. More general purpose schedulers must be non-clairvoyant, i.e., work without knowledge of t_j before j completes [23, 20]. To handle this, recourse to job preemption or cancellation is needed, whose large cost has to be factored into the algorithm.

Arbitrary DAGs. While we have handled hierarchical job graphs such as forests or series-parallel graphs, the general DAG case is open. It is known that precedence-constrained knapsack with general precedence is strongly $\mathcal{NP}$-hard [15], unlike forests. We show that settling the approximability issue will be challenging [1]. This shows that the framework of [13] may need modification to handle DAG's, not necessarily that the scheduling problem is difficult.

Claim 9 *There is an approximation preserving reduction from* Expansion, *the problem of estimating the vertex expansion of a bipartite graph, to* P.O.K , *the partial order knapsack problem, even when all item costs and profits are restricted to* $\{0, 1\}$.

Proof. Given $G = (L, R, E)$, suppose we need the vertex expansion of R. Construct a two layer partial order $\prec$: the upper layer contains a job j for every $u \in L$, with $c(j) = 1$, $p(j) = 0$. The lower layer contains a job j for every $v \in R$, with $c(j) = 0$, $p(j) = 1$. E is directed from L to R. Run the routine for P.O.K n times, with target profits $P \in \{1, \ldots, n\}$. Return $\min_P\{C(P)/P\}$, where $C(P)$ is the cost returned for target profit P. ∎

Flowtime. In this paper we consider $\max_j C_j$ and $\sum_j w_j C_j$; some more ambitious objective functions are $\sum_j (C_j - a_j)$ and $\sum_j w_j (C_j - a_j)$, commonly called *flowtime.* Unfortunately, even with a single machine, off-line problem instance, and no resource or precedence constraints, it is $\mathcal{NP}$-hard to approximate non-preemptive flowtime better than about a factor of $\Omega(\sqrt{n})$ [16]. In a practical implementation of our algorithm, one might artificially increase w_j for jobs waiting for a long time.

Acknowledgments. Thanks to Joel Wein for introducing us to the area.

References

[1] N. Alon. Eigenvalues and expanders. *Combinatorica*, 6(2):83–96, 1986.

[2] K. Brown et al. Resource allocation and scheduling for mixed database workloads. Technical Report 1095, University of Wisconsin at Madison, July 1992.

[3] S. Chakrabarti, C. Phillips, A. Schulz, D. Shmoys, C. Stein, and J. Wein. Improved scheduling algorithms for minsum criteria. In *Automata, Languages and Programming (ICALP)*, LNCS, Paderborn, Germany, July 1996. Springer-Verlag.

[4] C. Chekuri, R. Motwani, and B. Natarajan. Scheduling to minimize weighted completion time. Manuscript, 1995.

[5] K. Ekanadham, J. E. Moreira, and V. K. Naik. Application oriented resource management on large scale parallel systems. Technical Report RC 20151, IBM Research, Yorktown Heights, Aug. 1995.

[6] D. G. Feitelson and L. Rudolph, editors. *Job Scheduling Strategies for Parallel Processing*, number 949 in LNCS. Springer-Verlag, Apr. 1995. Workshop at International Parallel Processing Symposium.

[7] A. Feldmann, M.-Y. Kao, J. Sgall, and S.-H. Teng. Optimal online scheduling of parallel jobs with dependencies. In *Symposium on the Theory of Computing (STOC)*, pages 642–651. ACM, 1993.

[8] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, June 1975.

[9] M. Garofalakis and Y. Ioannidis. Multi-dimensional resource scheduling for parallel queries In *SIGMOD conference* ACM, May 1996. To appear.

[10] R. L. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, March 1969.

[11] S. L. Graham, S. Lucco, and O. Sharp. Orchestrating interactions among parallel computations. In *Programming Language Design and Implementation (PLDI)*, pages 100–111, Albuquerque, NM, June 1993. ACM.

[12] J. Gray. A survey of parallel database techniques and systems, September 1995. Tutorial at *VLDB*.

[13] L. Hall, D. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line algorithms. In *Symposium on Discrete Algorithms (SODA)*. ACM-SIAM, 1996.

[14] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, Jan. 1993. Also see *Parallel Query Processing using Shared Memory Multiprocessing and Disk Arrays* by W. Hong, PhD thesis, UCB/ERL M93-28.

[15] D. S. Johnson and K. A. Niemi. On knapsacks, partitions, and a new dynamic programming technique for trees. *Mathematics of Operations Research*, 8(1):1–14, February 1983.

[16] H. Kellerer, T. Tautenhahn, and G. J. Woeginger. Approximability and non-approximability results for minimizing total flow time on a single machine. In *Symposium on the Theory of Computing (STOC)*. ACM, May 1996. To appear.

[17] W. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *Symposium on Discrete Algorithms (SODA)*, pages 167–176. ACM-SIAM, 1994.

[18] M. Mehta and D. Dewitt. Dynamic memory allocation for multiple-query workloads. In *VLDB*, pages 354–367, 1993.

[19] J. E. Moreira, V. K. Naik, and R. B. Konuru. A system for dynamic resource allocation and data distribution. Technical Report RC 20257, IBM Research, Yorktown Heights, Oct. 1995.

[20] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130:17–47, August 1994. Preliminary version in SODA 1993, pp 422–431.

[21] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD conference*, pages 23–34, 1979.

[22] D. Shmoys, C. Stein, and J. Wein. Improved approximation algorithms for shop scheduling problems. *SIAM Journal on Computing*, 23:617–632, 1994. Preliminary version in SODA 1991.

[23] D. B. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines online. In *Foundations of Computer Science (FOCS)*, pages 131–140, 1991.

[24] J. Turek, W. Ludwig, J. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schweigelshohn, and P. S. Yu. Scheduling parallelizable tasks to minimize average response time. In *Symposium on Parallel Algorithms and Architectures (SPAA)*. ACM, 1994.

[25] J. Turek, U. Schwiegelshohn, J. Wolf, and P. Yu. Scheduling parallel tasks to minimize average response time. In *Symposium on Discrete Algorithms (SODA)*, pages 112–121. ACM-SIAM, 1994.

[26] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 323–332, 1992

[27] J. Wolf, J. Turek, M. Chen, and P. Yu. The optimal scheduling of multiple queries in a parallel database machine. Technical Report RC 18595 (81362) 12/17/92, IBM, 1992.