Task Scheduling in Networks*

Cynthia Phillips[†]
Sandia National Labs
Albuquerque, NM

Clifford Stein[‡]
Department of CS
Dartmouth College
Hanover, NH

Joel Wein [§]
Department of CS
Polytechnic University
Brooklyn, NY

May 8, 1995

Abstract

Scheduling a set of tasks on a set of machines so as to yield an efficient schedule is a basic problem in computer science and operations research. Most of the research on this problem incorporates the potentially unrealistic assumption that communication between the different machines is instantaneous. In this paper we remove this assumption and study the problem of *network scheduling*, where each job originates at some node of a network, and in order to be processed at another node must take the time to travel through the network to that node.

Our main contribution is to give approximation algorithms and hardness proofs for fully general forms of the fundamental problems in network scheduling. We consider two basic scheduling objectives: minimizing the makespan, and minimizing the average completion time. For the makespan we prove small constant factor hardness-to-approximate and approximation results. For the average completion time, we give a log-squared approximation algorithm for the most general form of the problem. The techniques used in this approximation are fairly general and have several other applications. For example, we give the first non-trivial approximation algorithm to minimize the average weighted completion time of a set of jobs on related or unrelated machines, with or without a network.

Keywords: Scheduling, Approximation Algorithm, NP-completeness, Networks

AMS(MOS) subject classifications: 68M10, 90B12, 68Q10, 68Q22, 68Q25, 90B35, 68M20

*Much of this work was done while the second and third authors were visiting Sandia National Labs, the third author was visiting DIMACS, and the second author was visiting Polytechnic University.

[†] `caphill@cs.sandia.gov`.

This work was performed under U.S. Department of Energy contract number DE-AC04-76DP00789.

[‡] `cliff@cs.dartmouth.edu`.

Research partly supported by NSF Award CCR-9308701, a Walter Burke Research Initiation Award and a Dartmouth College Research Initiation Award.

[§] `wein@mem.poly.edu`.

Research partially supported by NSF Research Initiation Award CCR-9211494 and a grant from the New York State Science and Technology Foundation, through its Center for Advanced Technology in Telecommunications.

1 Introduction

Scheduling a set of tasks on a set of machines so as to yield an efficient schedule is a basic problem in computer science and operations research. It is also a difficult problem and hence, much of the research in this area has incorporated a number of potentially unrealistic assumptions. One such assumption is that communication between the different machines is instantaneous. In many application domains, however, such as a network of computers or a set of geographically-scattered repair shops, decisions about when and where to move the tasks are a critical part of achieving efficient resource allocation. In this paper we remove the assumption of instantaneous communication from the traditional parallel machine models and study the problem of *network scheduling*, in which each job originates at some node of a network, and in order to be processed at another node must take the time to travel through the network to that node.

Until this work, network scheduling problems had either loose [2, 4] or no approximation algorithms. Our main contribution is to give approximation algorithms and hardness proofs for fully general forms of the fundamental problems in network scheduling. Our upper bounds are robust, as they depend on general characteristics of the jobs and the underlying network. In particular, our algorithmic techniques to optimize average completion time yield other results, such as the first nontrivial approximation algorithms for a combinatorial scheduling question: minimization of average *weighted* completion time on unrelated machines, and the first approximation algorithm for a problem motivated by satellite communication systems. (To differentiate our network scheduling models from the traditional parallel machine models, we will refer to the latter as *combinatorial* scheduling models.)

Our results not only yield insight into the network scheduling problem, but also demonstrate contrasts between the complexity of certain combinatorial scheduling problems and their network variants, shedding light on their relative difficulty.

An instance $\mathcal{N} = (G, \ell, \mathcal{J})$ of the network scheduling problem consists of a network $G = (V, E)$, $|V| = m$, with non-negative edge lengths ℓ; we define $\ell_{\max}$ to be the maximum edge length. At each vertex v_i in the network is a machine M_i. We are also given a set of n jobs, $J_1, \ldots, J_n$. Each job J_j originates, at time 0, on a particular *origin machine* M_{o_j} and has a processing requirement p_j; we define $p_{\max}$ to be $\max_{1\leq j\leq n} p_j$. Each job must be processed on one machine without interruption. Job J_j is not available to be processed on a machine M' until time $d(M_{o_j}, M')$, where $d(M_i, M_k)$ is the length of the shortest path in G between M_i and M_k. We assume that the M_i are either identical (J_j takes time p_j on every machine) or that they are *unrelated* (J_j takes time p_{ij} on M_i, and the p_{ij} may all be different). In the unrelated machines setting we define $p_{\max} = \max_{1\leq i\leq m, 1\leq j\leq n} p_{ij}$. The identical and unrelated machine models are fundamental in traditional parallel machine scheduling and are relatively well understood [3, 10, 11, 12, 15, 17, 25]. Unless otherwise specified, in this paper the machines in the network are assumed to be identical.

An alternative view of the network scheduling model is that each job J_j has a *release date*, a time before which it is unavailable for processing. In previous work on traditional scheduling models a job's release date was defined to be the same on all machines. The network model can be characterized by allowing a job J_j's release date to be different on different machines; J_j's release date on M_k is $d(M_{o_j}, M_k)$. One can generalize further and consider problems in which a job's release date can be chosen arbitrarily for all m machines, and need not reflect any network structure. Almost all of our upper bounds apply in this more general setting, whereas our lower bounds all apply when the release dates have network structure.

We study algorithms to minimize the two most basic objective functions. One is the *makespan* or *maximum completion time* of the schedule; that is, we would like all jobs to finish by the earliest time possible. The second is the *average completion time*. We define an α-approximation algorithm to be a polynomial-time algorithm that gives a solution of cost no more than α times optimal.

1.1 Previous Work

The problem of network scheduling has received some attention, mostly in the distributed setting. Deng et. al. [4] considered a number of variants of the problem. In the special case in which each edge in the network is of unit length, all job processing times are the same, and the machines are identical, they showed that the off-line problem is in $\mathcal{P}$. It is not hard to see that the problem is $\mathcal{NP}$-Complete when jobs are allowed to be of different sizes; they give an off-line $O(\log(m\ell_{\max}))$-approximation algorithm for this. They also give a number of results on the distributed problem when the network topology is completely connected, a ring or a tree.

Awerbuch, Kutten and Peleg [2] considered the distributed version of the problem under a novel notion of on-line performance, which subsumes the minimization of both average and maximum completion time. They give distributed algorithms with polylogarithmic performance guarantees in general networks. They also characterize the performance of feedback-based approaches. In addition they derived off-line approximation results similar to those of Deng et. al [2, 20]. Alon et. al. [1] proved an $\Omega(\log m)$ lower bound on the performance of any distributed scheduler that is trying to minimize schedule length. Fizzano et. al. [5] give a distributed 4.3-approximation algorithm for schedule length in the special case in which the network is a ring.

Our work differs from these papers by focusing on the centralized off-line problem and by giving approximations of higher quality. In addition, our approximation algorithms work in a more general setting, that of unrelated machines.

1.2 Summary of Results

We first focus on the objective of minimizing the makespan, and give a 2-approximation algorithm for scheduling jobs on networks of unrelated machines; the algorithm gives the same performance guarantee for identical machines as a special case. The 2-approximation algorithm matches the best known approximation algorithm for scheduling unrelated machines with no underlying network [17]. Thus it is natural to ask whether the addition of a network to a combinatorial scheduling problem actually makes the problem any harder. We resolve this question by proving that the introduction of the network to the problem of scheduling identical machines yields a qualitatively harder problem. We show that for the network scheduling problem, no polynomial-time algorithm can do better than a factor of $\frac{4}{3}$ times optimal unless $\mathcal{P} = \mathcal{NP}$, even in a network in which all edges have length one. Comparing this with the polynomial approximation scheme of Hochbaum and Shmoys [10] for parallel machine scheduling, we see that the addition of a network does indeed make the problem harder.

Although the 2-approximation algorithm runs in polynomial time, it may be rather slow [21]. We thus explore whether a simpler strategy might also yield good approximations. A natural approach to minimizing the makespan is to construct schedules with no unforced idle time. Such strategies provide schedules of length a small constant factor times optimal, at minimal computational cost, for a variety of scheduling problems [6, 7, 15, 24]. We call such schedules *busy schedules*, and show that for the network scheduling problem their quality degrades significantly; they can be as much as an $\Omega\left(\sqrt{\frac{\log m}{\log\log m}}\right)$ factor longer than the optimal schedule.

This is in striking contrast to the combinatorial model (for which Graham showed that a busy strategy yields a 2-approximation algorithm [6]). In fact, even when release dates are introduced into the identical machine scheduling problem, if each job's release date is the same on all machines, busy strategies still give a $(2-\frac{1}{m})$-approximation guarantee [8, 9]. Our result shows that when the

	Combinatorial	Network
min. makespan, identical machines	$\alpha < (1+\epsilon)$[10]	$4/3 < \alpha \leq 2$
min. makespan, identical machines, Busy schedules	$\alpha = 2 - \frac{1}{m}$[6]	$O\left(\frac{\log m}{\log\log m}\right), \Omega\left(\sqrt{\frac{\log m}{\log\log m}}\right)$
min. makespan, unrelated machines	$3/2 < \alpha \leq 2$ [17]	$3/2 < \alpha \leq 2$
min. avg. completion time unrelated machines	$\alpha = 1$ [12]	$1 < \alpha \leq O(\log^2 n)$
min. avg. wtd. completion time unrelated machines, release dates	$1 < \alpha$[16] $\alpha \leq O(\log^2 n)$	$1 < \alpha \leq O(\log^2 n)$

Figure 1: Summary of main algorithms and hardness results. The notation $x < \alpha \leq y$ means that we can approximate the problem within a factor of y, but unless $\mathcal{P} = \mathcal{NP}$ we cannot approximate the problem within a factor of x. Unreferenced results are new results found in this paper.

release dates of the jobs are allowed to be different on different machines busy scheduling degrades significantly as a scheduling strategy. This provides further evidence that the introduction of a network makes scheduling problems qualitatively harder. However, busy schedules are of some quality; we show that they are of length a factor of $O\left(\frac{\log m}{\log\log m}\right)$ longer than optimal. This analysis gives a better bound than the $(O(\log m\ell_{\max}))$ bound of previously known approximation algorithms for identical machines in a network [2, 4, 20].

We then turn to the $\mathcal{NP}$-hard problem of the minimization of average completion time. Our major result for this optimality criterion is a $O(\log^2 n)$-approximation algorithm in the general setting of unrelated machines. It formulates the problem as a hypergraph matching integer program and then approximately solves a relaxed version of the integer program. We can then find an integral solution to this relaxation, employing as a subroutine the techniques of Plotkin, Shmoys and Tardos [21]. In combinatorial scheduling, a schedule with minimum average completion time can be found in polynomial time, even if the machines are unrelated.

The techniques for the average completion time algorithm are fairly general, and yield an $O(\log^2 n)$-approximation for minimizing the average *weighted* completion time. A special case of this result is an $O(\log^2 n)$-approximation algorithm for the $\mathcal{NP}$-hard problem of minimizing average weighted completion time for unrelated machines with no network; no previous approximation algorithms were known, even in the special case for which the machines are just of different speeds [3, 15]. Another special case is the first $O(\log^2 n)$-approximation algorithm for minimizing the average completion time of jobs with release dates on unrelated machines. No previous approximation algorithms were known, even for the special case of *just one machine* [15]. The technique can also be used to give an approximation algorithm for a problem motivated by satellite communication systems [18, 26].

We also give a number of other results, including polynomial-time algorithms for several special cases of the above-mentioned problems and a $\frac{5}{2}$-approximation for a variant of network scheduling in which each job has not only an origin, but also a destination.

A summary of some of these upper bounds and hardness results appears in Figure 1.

A line of research which is quite different from ours, yet still has some similarity in spirit, was started by Papadimitriou and Yannakakis [19]. They modeled communication issues in parallel machine scheduling by abstracting away from particular networks and rather describing the communication time between *any* two processors by one network-dependent constant. They considered the scheduling of precedence-constrained jobs on an infinite number of identical machines in this model; the issues involved and the sorts of theorems proved are quite different from our results.

Although all of our algorithms are polynomial-time algorithms, they tend to be rather inefficient.

Most rely on the work of [21] as a subroutine. As a result we will not discuss running times explicitly for the rest of the paper.

2 Makespan

In this section we study the problem of minimizing the makespan for the network scheduling problem. We first give an algorithm that comes within a factor of 2 of optimal. We then show that this is nearly the best we can hope for, as it is $\mathcal{NP}$-hard to approximate the minimum makespan within a factor of better than $\frac{4}{3}$ for identical machines in a network. This hardness result contrasts sharply with the combinatorial scenario, in which there is a polynomial approximation scheme [10]. The 2-approximation algorithm is computationally intensive, so we consider simple strategies that typically work well in parallel machine scheduling. In another sharp contrast to parallel machine scheduling, we show that the performance of such strategies degrades significantly in the network setting; we prove an $\Omega\left(\sqrt{\frac{\log m}{\log\log m}}\right)$ lower bound on the performance of any such algorithm. We also show that greedy algorithms do have some performance guarantee, namely $O(\frac{\log m}{\log\log m})$. Finally we consider a variant of the problem in which each job has not only an origin, but also a destination, and give a $\frac{5}{2}$-approximation algorithm.

2.1 A 2-Approximation Algorithm For Makespan

In this section we describe a 2-approximation algorithm to minimize the makespan of a set of jobs scheduled on a network of unrelated machines; the same bound for identical machines follows as a special case. Let $\mathcal{U}' = (G, \ell, \mathcal{J}')$ be an instance of the unrelated network scheduling problem with optimal schedule length D. Assuming that we know D, we will show how to construct a schedule of length at most $2D$. This can be converted, via binary search, into a 2-approximation algorithm for the problem in which we are not given D [10].

In the optimal schedule of length D, we know that the sum of the time each job spends travelling and being processed is bounded by D. Thus, a characterization of when job J_j is able to run on machine M_i in the optimal schedule is that:

$$d(M_{o_j}, M_i) + p_{ij} \leq D. \tag{1}$$

In other words, the length of an optimal schedule is not altered if we allow job J_j to run only on the machines for which (1) is satisfied. Formally, for a given job J_j, we will denote by $Q(J_j)$ the set of machines that satisfy (1). If we restrict each J_j to only run on the machines in $Q(J_j)$, the length of the optimal schedule remains unchanged.

Form combinatorial unrelated machines scheduling problem $(\mathcal{Z})$ as follows:

$$p'_{ij} = \begin{cases} p_{ij} & \text{if } M_i \in Q(J_j) \\ \infty & \text{otherwise} \end{cases} \tag{2}$$

If the optimal schedule for the unrelated network scheduling problem has length D, then the optimal solution to the unrelated parallel machine scheduling problem (2) is at most D. We will use the 2-approximation algorithm of Lenstra, Shmoys and Tardos [17] to assign jobs to machines. The following theorem is easily inferred from [17].

Theorem 2.1 (Lenstra, Shmoys, Tardos [17]) *Let $\mathcal{Z}$ be an unrelated parallel machine scheduling problem with optimal schedule of length D. Then there exists a polynomial-time algorithm that finds a schedule $\mathcal{S}$ of length $2D$. Further, $\mathcal{S}$ has the property that no job starts after time D.*

Theorem 2.2 *There exists a polynomial-time 2-approximation algorithm to minimize makespan in the unrelated network scheduling problem.*

Proof: Given an instance of the unrelated network scheduling problem, with shortest schedule of length D, form the unrelated parallel machine scheduling problem $\mathcal{Z}$ defined by (2) and use the algorithm of [17] to produce a schedule $\mathcal{S}$ of length $2D$. This schedule does not immediately correspond to a network schedule because some jobs may have been scheduled to run before their release dates. We now show how to convert this schedule into a schedule for the network problem.

In schedule $\mathcal{S}$, each machine M_i is assigned a set of jobs S_i. Let $|S_i|$ be the sum of the processing times of the jobs in S_i and let $S_i^{\max}$ be the job in S_i with largest processing time on machine i; call its processing time $p_i^{\max}$. By Theorem 2.1 and the fact that the last job run on machine i is no longer than the longest job run, we know that $|S_i| - p_i^{\max} \leq D$. Let $S_i^{'}$ denote the set of jobs $S_i - S_i^{\max}$. We form the schedule for each machine i by running job $S_i^{\max}$ at time $D - p_i^{\max}$, followed by the jobs in $S_i^{'}$.

In this schedule the jobs assigned to any machine clearly finish by time $2D$; it remains to be shown that all jobs can be routed to the proper machines by the time they need to run there. Job $S_i^{\max}$ must start at time $D - p_i^{\max}$; conditions (1) and (2) guarantee that it arrives in time. The remaining jobs need only arrive by time D; conditions (1) and (2) guarantee this as well. Thus we have produced a valid schedule of length $2D$. ■

Observe that this approach is fairly general and can be applied to any problem that can be characterized by a condition such as (2). Consider, for example the following very general problem, which we call *generalized network scheduling with costs.* In addition to the usual unrelated network scheduling problem, the time that it takes for job J_j to travel over an edge is dependent not only on the endpoints of the edge, but also on the job. Further, there is a cost c_{ij} associated with processing job J_j on machine M_i. Given a schedule in which job J_j actually runs on machine $M_{\pi(j)}$, the cost of a schedule is $\sum_j c_{\pi(j),j}$. Given any target cost C, we define $s(C)$ to be the minimum length schedule of cost at most C.

Theorem 2.3 *Given a target cost C, we can, in polynomial time, find a schedule for the generalized network scheduling problem with makespan at most $2s(C)$ and of cost C if a a schedule of cost C exists.*

Proof: We use similar techniques to those used for Theorem 2.2. We first modify equation 1 so that $d(\cdot,\cdot)$ depends on the job as well. We then use a generalization of the algorithm of Lenstra, Shmoys and Tardos for unrelated machine scheduling, due to Shmoys and Tardos [25] which, given a target cost C finds a schedule of cost C and length at most twice that of the shortest schedule of cost C. The schedule returned also has the property that no job starts after time D, so the proof of Theorem 2.2 goes through if we use this algorithm in place of the algorithm of [17]. ■

2.2 Nonapproximability

Theorem 2.4 *It is $\mathcal{NP}$-complete to determine if an instance of the network scheduling problem has a schedule of length 3, even in a network with $\ell_{\max} = 1$.*

Proof: See Appendix.

Corollary 2.5 *There does not exist an α-approximation algorithm for the network scheduling problem with $\alpha < 4/3$ unless $\mathcal{P} = \mathcal{NP}$, even in a network with $\ell_{\max} = 1$.*

Proof: Any algorithm with $\alpha < 4/3$ would have to give an exact answer for a problem with a schedule of length 3 since an approximation of 4 would have too high a relative error. ■

It is not hard to see, via matching techniques, that it is polynomial-time decidable whether there is a schedule of length 2. We can show that this is not the case when the machines in the network can be unrelated. Lenstra, Shmoys and Tardos proved that it is $\mathcal{NP}$-Complete to determine if there is a schedule of length 2 in the traditional combinatorial unrelated machine model [17]. If we allow multiple machines at one node, their proof proves Theorem 2.6. If no zero length edges are allowed, i.e. each machine is forced to be at a different network node, this proof does not work, but we can give a different proof of hardness, which we do not include in this paper.

Theorem 2.6 *There does not exist an α-approximation algorithm for the unrelated network scheduling problem with $\alpha < 3/2$ unless $\mathcal{P} = \mathcal{NP}$, even in a network with $\ell_{\max} = 1$.*

2.3 Naive Strategies

The algorithms in Section 2.1 give reasonably tight bounds on the approximation of the schedule length. Although these algorithms run in polynomial time, they may be rather slow [21]. We thus explore whether a simpler strategy might also yield good approximations.

A natural candidate is a *busy* strategy: construct a *busy schedule*, in which, at any time t there is no idle machine M_i and idle job J_j so that job J_j can be started on M_i at time t. Busy strategies and their variants have been analyzed in a large number of scheduling problems (see [15]) and have been quite effective in many of them. For combinatorial identical machine scheduling, Graham showed such strategies yield a $(2 - \frac{1}{m})$ approximation guarantee [6]. In this section we analyze the effectiveness of busy schedules for network scheduling. Part of the interest of this analysis lies in what it reveals about the relative hardness of scheduling with and without an underlying network; namely, the introduction of an underlying network can make simple strategies much less effective for the problem.

2.3.1 A Lower Bound

We construct a family of instances of the network scheduling problem, and demonstrate, for each instance, a busy schedule which is $\Omega(\sqrt{\frac{\log m}{\log\log m}})$ longer than the shortest schedule for that instance. The network $G = (V, E)$ consists of ℓ levels of nodes, with level $i, 1 \le i \le \ell$, containing ρ^{i-1} nodes. Each node in level $i, 1 \le i < \ell - 1$, is connected to every node in level $i + 1$ by an edge of length 1. Each machine in levels $1, \ldots, \ell - 1$ receives ρ jobs of size 1 at time 0. The machines in level ℓ initially receive no jobs. The optimal schedule length for this instance is 2 and is achieved by each machine in level $i, 2 \le i \le \ell$ taking exactly one job from level $i - 1$. We call this instance $\mathcal{I}$. See Figure 2.

The main idea of the lower bound is to construct a busy schedule in which machine M always processes a job which originated on M, if such a job is available. This greediness “prevents” the scheduler from making the much larger assignment of jobs to machines at time 2 in which each job is assigned to a machine one level away.

To construct a busy schedule S, we use algorithm $\mathcal{B}$, which in Step t constructs the subschedule of S at time t.

Step t:

Phase 1: Each machine M processes one job that originated at M, if any such jobs remain. We call such jobs *local* to machine M.

Phase 2: Consider the bipartite graph $G^* = (X, Y, A)$, where X has one vertex representing each job that is unprocessed after Phase 1 of time t, Y contains one vertex representing each machine which has not had a job assigned to it in Phase 1 of Step t, and $(x, y) \in A$ if and only if job x

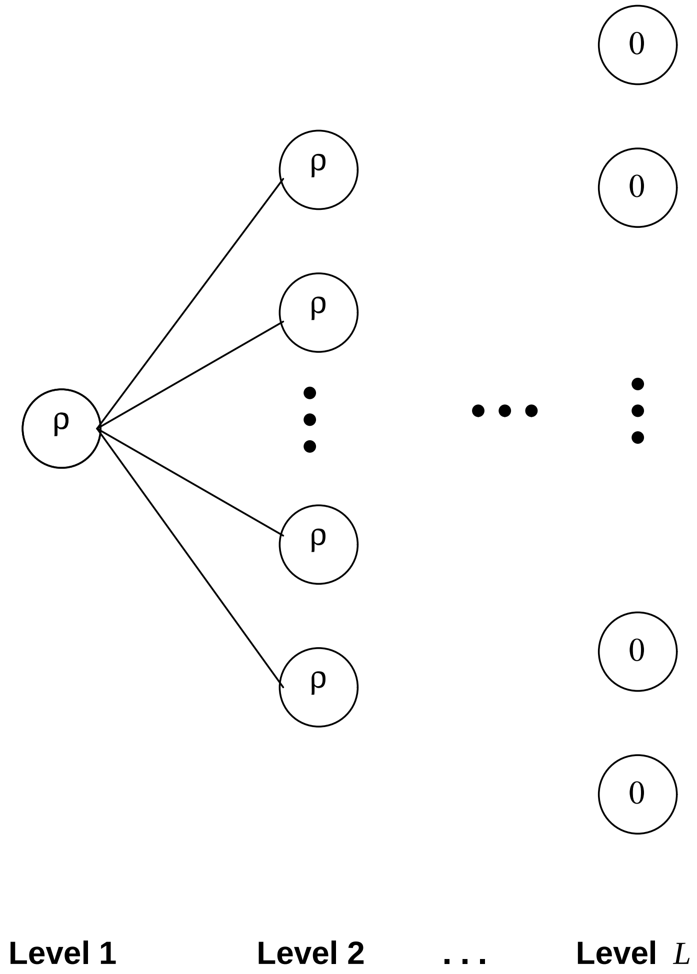

Figure 2: Lower Bound Instance for Theorem 2.8. Circles represent processors, and the numbers inside the circles are the number of jobs which originate at that processor at time 0. Levels i and $i+1$ are completely connected to each other. The optimal schedule is of length 2 and is achieved by shifting each job to a unique processor one level to its right.

originated a distance no more than $t-1$ from machine y. Complete the construction of S at time t by processing jobs on machines based on any maximum matching in G^*. It is clear that S is busy.

When we apply algorithm $\mathcal{B}$ to instance $\mathcal{I}$, the behavior follows a well-defined pattern. In Phase 2 of Step 2, all unprocessed jobs that originated in level $\ell-1$ are processed by distinct processors in level ℓ. During Phase 2 of Step 3, all unprocessed jobs that originated in levels $\ell-2$ and $\ell-3$ are processed by machines in levels $\ell-1$ and ℓ. This continues, so that at Step i an additional $(i-1)$ levels pass their jobs to higher levels and all these jobs are processed. This continues until either level 1 passes its jobs, or processes its own jobs. We characterize the behavior of the algorithm more formally in the following lemma.

Lemma 2.7 *Let $j(i,t)$ be the number of local jobs of processor i still unprocessed after Phase 2 of Step t and let $lev(i)$ be the level number of processor i. Then for all times $t \geq 2$, if $\rho \geq t$, then*

$$j(i,t) = \begin{cases} 0 & \text{if } lev(i) \geq \ell - t(t-1)/2 \\ j(i,t-1)-1 & \text{otherwise} \end{cases} \tag{3}$$

Proof: We prove the lemma by induction on t. During Phase 2 of Step 2, the only edges in the graph G^* connect levels ℓ and $\ell-1$. There are $\rho^{\ell-1}$ nodes in level ℓ and $\rho^{\ell-2}(\rho-1)$ remaining jobs local to machines in level $\ell-1$, so the matching assigns all the unprocessed jobs in level $\ell-1$ to level ℓ. Machines in level 1 to $\ell-1$ all process local jobs during Phase 1. As a result, all the neighbors of machines in levels 1 to $\ell-2$ are busy in Phase 1 and cannot process jobs local to these machines during Phase 2. The number of local jobs on these machines, therefore, decreases only by 1. Thus the base case holds.

Assume the lemma holds for all $t < t'$. Then $j(i,t'-1) = 0$ for levels greater than $b \equiv \ell-(t'-1)(t'-2)/2$, and $j(i,t') = 0$ for levels greater than b as well. We now show that $j(i,t') = 0$ if $lev(i) \geq \ell - t'(t'-1)/2$. For $1 \leq x \leq t'-1$, level $b+x$ has ρ^{b+x-1} processors. Level $b+x-(t'-1)$ has at most $\rho \cdot \rho^{b+x-(t'-1)-1} = \rho^{b+x-t'+1}$ local jobs remaining. If $t' \geq 2$ then there are enough machines on level $b+x$ to process all the remaining jobs local to level $b+x-(t'-1)$. Therefore another $t'-1$ of the highest-numbered levels have their local jobs completed during time t'. Thus at time t' we have $j(i,t') = 0$ if $lev(i) \geq \ell - t'(t'-1)/2$.

Since we assumed sufficiently large initial workloads on all processors on levels $1\ldots(\ell-1)$, then by the induction hypothesis, for all machines in levels less than $\ell - t'(t'-1)/2$, all machines within distance $t'-1$ of them have local jobs remaining after time $t'-1$ and will be assigned a local job during Phase 1 of Step t'. Therefore all machines i such that $lev(i) < \ell - t'(t'-1)/2$ cannot pass any jobs to higher levels and $j(i,t') = j(i,t'-1)-1$. ■

Depending on the relative values of ρ and ℓ, either the machine in level 1 processes all of the jobs which originated on it, or some of those jobs are processed by machines in higher-numbered levels. Balancing these two cases we get the following theorem:

Theorem 2.8 *For the family of instances of the network scheduling problem defined above, there exist busy schedules of length a factor $\Omega(\sqrt{\frac{\log m}{\log\log m}})$ longer than optimal.*

Proof: The first case in (3) will apply to level 1 when $1 \geq \ell - t(t-1)/2$. This inequality does not hold when $t = \sqrt{2\ell}$, but it does hold when $t = \sqrt{2\ell}+1$. Thus, if $\rho > \sqrt{2\ell}$ then the schedule length is $\sqrt{2\ell}$, while if $\rho < \sqrt{2\ell}$ then the jobs in level 1 will be totally processed in their level, which takes ρ time. Therefore the makespan of S is at most $\min(\sqrt{2\ell},\rho)$. Given that the total number of machines is $m = \theta(\rho^{\ell-1})$, a simple calculation reveals that $\min(c\sqrt{\ell},\rho)$ is maximized at $\ell = \theta(\frac{\log m}{\log\log m})$. Thus S is a busy schedule of length $\theta(\sqrt{\frac{\log m}{\log\log m}})$ longer than optimal. ■

Note that this example shows that several natural variants of busy strategies, such as scheduling a job on the machine on which it will finish first, or scheduling a job on the closest available processor, also perform poorly.

2.3.2 An Upper Bound

In contrast to the lower bound of the previous subsection, we can prove that busy schedules are of some quality. Given an instance $\mathcal{I}$ of the network scheduling problem, we define $C^*_{\max}(\mathcal{I})$ to be the length of a shortest schedule for $\mathcal{I}$ and $C^A_{\max}(\mathcal{I})$ to be the length of the schedule produced by algorithm A; when it causes no confusion we will drop the $\mathcal{I}$ and use the notation $C^*_{\max}$.

Definition 2.9 *Consider a busy schedule S for an instance $\mathcal{I}$ of the network scheduling problem. Let $p_j(t)$ be the number of units of job J_j remaining to be processed in schedule S at time t, and $W_t = \sum_{k=1}^{j} p_k(t)$ be the total work remaining to be processed in schedule S at time t.*

Lemma 2.10 $W_{iC^*_{\max}} \leq \frac{W_0}{2i!}$ *for* $i \geq 1$.

Proof: We partition schedule S into consecutive blocks $B_1, B_2, \ldots$ of length $C^*_{\max}(\mathcal{I})$ and compare what happens in each block of schedule S to an optimal schedule S^* of length $C^*_{\max}$ for instance $\mathcal{I}$.

Consider a job J_j that was not started by time $C^*_{\max}$ in schedule S, and let M_j be the machine on which job J_j is processed in schedule S^*. This means that in block B_1 machine M_j is busy for p_j units of time during job J_j's *slot* in schedule S^* – the period of time during which job J_j was processed on machine M_j in schedule S^*. Hence for every job J_j that is not started in block B_1 there is an equal amount of unique work which we can identify that *is* processed in block B_1, implying that $W_{C^*_{\max}} \leq W_0/2$. Successive applications of this argument yields $W_{iC^*_{\max}} \leq W_0/2^i$ for $i \geq 1$, which proves the lemma for $i = 1, 2$.

To obtain the stronger bound $W_{iC^*_{\max}} \leq \frac{1}{2}(W_0/i!)$, we increase the amount of processed work which we identify with each unstarted job. Choose $i \geq 3$ and consider a job J_j which is unstarted in schedule S at the start of block B_{i+1}, namely at time $iC^*_{\max}$. Assume for the sake of simplicity that in every block B_k of schedule S, only one job is processed in job J_j's slot (the time during which job J_j would be processed if block B_k was schedule S^*). Assume also that this job is exactly of the same size as job J_j; if multiple jobs are processed the argument is essentially the same. Let J_r be the job that took job J_j's slot in block B_r, for $r \leq i-2$. We will show that J_j could have been processed in J_r's slot in block B_i for all $1 \leq r \leq i-2$.

Assume that job J_j originated on machine M_{o_j}, and that job J_r originated on machine M_{o_r}, and that job J_j was processed on machine M_j in schedule S^*. Then $d(M_{o_j}, M_j) \leq C^*_{\max}$ since job J_j was processed on machine M_j in schedule S^*, and $d(M_{o_r}, M_j) \leq rC^*_{\max}$ since job J_r was processed in job J_j's slot in block B_r. Thus $d(M_{o_j}, M_{o_r}) \leq (r+1)C^*_{\max}$ and consequently J_j could have run in job J_r's slot in any of blocks $B_{r+2}, \ldots, B_i$. We focus on block B_i. Since J_j was not processed in block B_i and schedule S is busy, some job must have been processed during job J_r's slot in block B_i for $1 \leq r \leq (i-2)$. We identify this work with job J_j ; note that no work is ever identified with more than one job.

When we consider the $(i-2)$ different jobs which were processed in J_j's slot in blocks $B_1, \ldots, B_{i-2}$, and consider the jobs that were processed in their slots in B_i , we see that with each job J_j unstarted at time $iC^*_{\max}$, we can uniquely identify $(i-2)p_j$ units of work that was processed in block B_i. If all these slots were not full in block B_i, then job J_j would have been started in one of them. Including the work processed during job J_j's slot in block B_i, we obtain

$$W_{iC^*_{\max}} \leq \frac{1}{i} W_{(i-1)C^*_{\max}}.$$

■

Corollary 2.11 *During time $iC^*_{\max}$ to $(i+1)C^*_{\max}$ at most $m/(2i!)$ machines are completely busy.*

Proof: We have $W_0 \leq mC^*_{\max}$. Therefore, by Lemma 2.10, we have $W_{iC^*_{\max}} \leq mC^*_{\max}/(2i!)$. A machine that is completely busy from time $iC^*_{\max}$ to time $(i+1)C^*_{\max}$ does $C^*_{\max}$ work during that time and therefore at most $m/(2i!)$ machines can be completely busy. ■

To get a stopping point for the recurrence, we require the following lemma:

Lemma 2.12 *In any busy schedule, if at time t all remaining unprocessed jobs originated on the same machine, the schedule is no longer than $t + 2C^*_{\max}$.*

Proof: Let M be the one machine with remaining local jobs. Let $W^*_{M_i}$ be the amount of work from machine M that is done by machine M_i in the optimal schedule. Clearly $\sum_i W^*_{M_i}$ equals the amount of work that originated on machine M. Because there is no work left that originated on machines other than M, each machine M_i can process at least $W^*_{M_i}$ work from machine M in the next $C^*_{\max}$ time steps. If after $C^*_{\max}$ steps, all the work originating on machine M is done, then we have finished. Otherwise, some machine M_i processed less than $W^*_{M_i}$ work during this time, which means there was no more work for it to take. Therefore after $C^*_{\max}$ steps all the jobs that originated on machine M have started. Because no job is longer than $C^*_{\max}$, another $C^*_{\max}$ time suffices to finish all the jobs that have started. ■

We are now ready to prove the upper bound:

Theorem 2.13 *Let A be any busy scheduling algorithm and $\mathcal{I}$ an instance of the network scheduling problem. Then $C^A_{\max}(\mathcal{I}) = O(\frac{\log m}{\log\log m} C^*_{\max}(\mathcal{I}))$.*

Proof: If a machine ever falls idle, all of its local work must be started. Otherwise it would process remaining local work. Thus by Corollary 2.11, in $O(\frac{\lg m}{\lg \lg m})C^*_{\max}$ time, the number of processors with local work remaining is reduced to 1. By Lemma 2.12, when the number of processors with remaining local work is down to one, a constant number of extra blocks suffice to finish. ■

2.4 Scheduling with Origins and Destinations

In this subsection we consider a variant of the network scheduling problem in which each job, after being processed, has a *destination machine* to which it must travel. Specifically, in addition to having an origin machine M_{o_j}, job J_j also has a terminating machine M_{t_j}. Job J_j begins at machine M_{o_j}, travels distance $d(M_{o_j}, M_{d_j})$ to machine M_{d_j}, the machine it gets processed on, and then proceeds to travel for $d(M_{d_j}, M_{t_j})$ units of time to machine M_{t_j}. We call this problem the *point-to-point scheduling problem.*

Theorem 2.14 *There exists a polynomial-time $\frac{5}{2}$-approximation algorithm to minimize makespan in the point-to-point scheduling problem.*

Proof: We construct an unrelated machines scheduling problem as in the proof of Theorem 2.2. In this setting the condition on when a job J_j can run on machine M_i depends on the time for J_j to get to M_i, the time to be processed there, and the time to proceed to the destination machine. Thus we can replace the condition (1) with

$$d(M_{o_j}, M_i) + p_{ij} + d(M_i, M_{tj}) \leq D. \tag{4}$$

This gives an assignment of jobs to machines. Pick any machine M_i and let $\mathcal{J}_i$ be the set of jobs assigned to machine M_i. By Theorem 2.1 we know that the sum of the processing times of all of the jobs in $\mathcal{J}_i$ except the longest is at most D. We partition the set of jobs $\mathcal{J}_i$ into three groups, and place each job into the lowest numbered group which is appropriate:

1. $\mathcal{J}_i^0$ contains the job in $\mathcal{J}_i$ with the longest processing time,

2. $\mathcal{J}_i^1$ contains jobs for which $d(M_{o_j}, M_i) \leq D/2$,

3. $\mathcal{J}_i^2$ contains jobs for which $d(M_{o_j}, M_i) \geq D/2$.

Let $p(\mathcal{J}_i^k)$ be the sum of the processing times of the jobs in group $\mathcal{J}_i^k$, $k = 1, 2$. As noted above, $p(\mathcal{J}_i^1) + p(\mathcal{J}_i^2) \leq D$. We will always schedule $\mathcal{J}_i^1 \cup \mathcal{J}_i^2$ in a block of D consecutive time steps, which we call B. The first $p(\mathcal{J}_i^1)$ time steps will be taken up by jobs in $\mathcal{J}_i^1$ while the last $p(\mathcal{J}_i^2)$ time steps will be taken up by jobs in $\mathcal{J}_i^2$. Note that there may be idle time in the interior of the block.

We consider two possible scheduling strategies based on the relative sizes of $p(\mathcal{J}_i^1)$ and $p(\mathcal{J}_i^2)$.

Case 1:($p(\mathcal{J}_i^1) \leq p(\mathcal{J}_i^2)$) In this case we first run the long job in $\mathcal{J}_i^0$; by condition (4) it finishes by time D. We then run block B from time D to $2D$. Since $p(\mathcal{J}_i^1) \leq D/2$, the jobs in $\mathcal{J}_i^1$ all finish by time $3D/2$ and by condition (4) reach their destinations by time $5D/2$. By the definition of $\mathcal{J}_i^2$, for any job $J_j \in \mathcal{J}_i^2, d(M_i, M_{t_j}) \leq D/2$. Since every $J_j \in \mathcal{J}_i^2$ is scheduled to complete processing by time $2D$, it will arrive at its destination by time $5D/2$.

Case 2: ($p(\mathcal{J}_i^1) \geq p(\mathcal{J}_i^2)$) We first run block B from time $D/2$ to $3D/2$. We then start the long job in $\mathcal{J}_i^0$ at time $3D/2$; by condition (4) it arrives at its destination by time $5D/2$. Since $p(\mathcal{J}_i^2) \leq D/2$, machine M_i need not start processing any job in $\mathcal{J}_i^2$ until time D and hence we are guaranteed that they have arrived at machine M_i by that time. By definition of $\mathcal{J}_i^1$ all of its jobs are available by time $D/2$; it is straightforward from condition (4) that all jobs arrive at their destinations by time $5D/2$. ■

3 Average Completion Time

3.1 Background

We turn now to the network scheduling problem in which the objective is to minimize the average completion time. Given a schedule S, let C_j^S be the time that job J_j finishes running in S. The average completion time of S is $\frac{1}{n}\sum_j C_j^S$, whose minimization is equivalent to the minimization of $\sum_j C_j^S$. Throughout this section we assume without loss of generality that $n \geq m$.

We have noted in Section 1 that our network scheduling model can be characterized by a set of n jobs J_j and a set of release dates r_{ij}, where J_j is not available on m_i until time r_{ij}. We noted that this is a generalization of the traditional notion of release dates, in which $r_{ij} = r_{i'j}\ \forall i, i'$. We will refer to the latter as *traditional* release dates; the unmodified phrase *release date* will refer to the general r_{ij}.

The minimization of average completion time when the jobs have no release dates is polynomial-time solvable [3, 12], even on unrelated machines. The solution is based on a bipartite matching formulation, in which one side of the bipartition has jobs and the other side (machine, position) pairs. Matching J_j to (m_i, k) corresponds to scheduling J_j in the *kth-from-last* position on m_i; this edge is weighted by kp_{ij}, which is J_j's contribution to the average completion time if J_j is kth from last.

When release dates are incorporated into the scheduling model, it seems difficult to generalize this formulation. Clearly it can not be generalized precisely for arbitrary release dates, since even the one machine version of the problem of minimizing average completion time of jobs with release dates is strongly $\mathcal{NP}$-hard [3]. Intuitively, even the approximate generalization of the formulation seems difficult, since if all jobs are not available at time 0, the ability of J_j to occupy position k on m_i is dependent on which jobs precede it on m_i and when. Release dates associated with a network structure do not contain traditional release dates as a subclass even for one machine, so the $\mathcal{NP}$-completeness of the network scheduling problem does not follow immediately from the combinatorial hardness results; however, not surprisingly, minimizing average completion time for a network scheduling problem is $\mathcal{NP}$-complete.

Theorem 3.1 *The network scheduling problem with the objective of minimum average completion time is $\mathcal{NP}$-complete even if all the machines are identical and all edge lengths are* 1.

Proof: See Appendix.

In what follows we will develop an approximation algorithm for the most general form of this problem. We will follow the basic idea of utilizing a bipartite matching formulation; however we will need to explicitly incorporate time into the formulation. In addition, for the rest of the section we will consider a more general optimality criterion: average *weighted* completion time. With each J_j we associate a weight w_j, and the goal is to minimize $\sum_{i=1}^{i=n} w_j C_j$. All of our algorithms handle this more general case; in addition they allow the nm release dates r_{ij} to be arbitrary and not necessarily derived from the network structure. We consider first the special case of unit-size jobs.

3.2 Unit-Size Jobs

Theorem 3.2 *There exists a polynomial-time algorithm to schedule unit-size jobs on a network of identical machines with the objective of minimizing the average weighted completion time.*

Proof: We reduce the problem to minimum-weight bipartite matching. One side of the bipartition will have a node for each job J_j, $1 \le j \le n$, and the other side will have a node $[m_i, t]$ for $1 \le i \le m$, $t \in T_i$ with T_i to be described below. An edge $(J_j, [m_i, t])$ of weight $w_j(t+1)$ is included if J_j is available on m_i at time t, and the inclusion of that edge in the matching will represent the scheduling of J_j on m_i from time t to $t+1$. Release dates are included in the model by excluding an edge $(J_j, [m_i, t])$ if J_j will not be available on m_i by time t.

To determine the necessary sets T_i, we observe that there is no advantage in unforced idle time. Since each job is only one unit long, there is no reason to make it wait for a job of higher weight that is about to be released. It is clear, therefore, that setting $T_i = \{t | r_{ij} \le t \le r_{ij} + n \ \forall j\}$ would suffice, since no job would need to be scheduled more than n time later than its release date. This gives $T_i = O(n^2)$; this can be reduced to $O(n)$, but we omit the details for the sake of space. ■.

By excluding edges which do not give job J_j enough time to travel between the machine on which J_j runs and the destination machine M_{d_j}, we can prove a similar theorem for the point-to-point scheduling problem.

Theorem 3.3 *There exists a polynomial-time algorithm to solve the point-to-point scheduling problem with the objective of minimizing the average weighted completion time of unit-size jobs.*

3.3 Polynomial-Size Jobs

We now turn to the more difficult setting of jobs of different sizes and unrelated machines. The minimization of average weighted completion time in this setting is strongly $\mathcal{NP}$-hard, as are many special cases. For example, the minimization of average completion time of jobs with release dates on one machine is strongly $\mathcal{NP}$-hard [16]; no approximation algorithms were known for this special case, to say nothing of parallel identical or unrelated machines, or weighted completion times. If there are no release dates, namely all jobs are available at time 0, then minimization of average weighted completion time is $\mathcal{NP}$-hard for parallel identical machines. A small constant factor approximation algorithm was known for this problem [14], but no approximation algorithms were known for the more general cases of machines of different speeds or unrelated machines. We introduce techniques which yield the first approximation algorithms for several other problems as well, which we discuss in Section 3.5.

Our approximation algorithm for minimum average completion time begins by formulating the scheduling problem as a *hypergraph matching problem.* The set of vertices will be the union of two sets, J and M, and the set of hyperedges will be denoted by F. J will contain n vertices J_j, one for each job, and M will contain mT vertices, where T is an upper bound on the number of time units that will be needed to schedule this instance. The time units will range over $\mathcal{T} = \{t | \exists r_{ij} \text{ with } r_{ij} \leq t \leq r_{ij} + np_{\max}\}$. M will have a node for each (machine, time) pair; we will denote the node that corresponds to machine M_i at time t as $[m_i, t]$. A hyperedge $e \in F$ represents scheduling a job J_j on machine M_i from time t_1 to t_2 by including nodes $J_j, [m_i, t_1], [m_i, t_1 + 1], \ldots, [m_i, t_2]$. The cost of an edge e, denoted by c_e, will be the weighted completion time of job J_j if it is scheduled in the manner represented by e. There will be one edge in the hypergraph for each feasible scheduling of a job on a machine; we exclude edges that would violate the release date constraints.

The problem of whether there exists a matching in the hypergraph of cost at most C which matches all of J can be phrased as the following integer program $\mathcal{I}$. We use decision variable $x_e \in \{0, 1\}$ to denote whether hyperedge e is in the matching.

$$
\begin{aligned}
\text{minimize } & \sum_e x_e c_e \\
\text{subject to } & \\
& \sum_{j \in e} x_e \quad = \quad 1 \qquad\qquad j = 1 \ldots, n \\
& \sum_{(i,t) \in e} x_e \quad \leq \quad 1 \qquad\qquad \forall (i,t) \in M \\
& x_e \quad \in \quad \{0, 1\}.
\end{aligned} \tag{5}
$$

Two considerations suggest that this formulation might not be useful. The formulation is not of polynomial size in the input size, and in addition the following theorem suggests that calculating approximate solutions for this integer program may be difficult.

Theorem 3.4 *Consider an integer program in the form $\mathcal{I}$ which is derived from an instance of the network scheduling problem with identical machines, with the c_e allowed to be arbitrary. Then there exists no polynomial-time algorithm $\mathcal{A}$ to approximate $\mathcal{I}$ within any factor unless $\mathcal{P} = \mathcal{NP}$.*

Proof: For an arbitrary instance of the network scheduling problem construct the hypergraph matching problem in which an edge has weight $W >> n$ if it corresponds to a job being completed later than time 3 and give all other edges weight 1. If there is a schedule of length 3 then the minimum weight hypergraph matching is of weight n; otherwise the weight is at least W; therefore an α-approximation algorithm with $\alpha < \frac{W}{n}$ would give a polynomial-time algorithm to decide if there was a schedule of length 3 for the network scheduling problem, which by Theorem 2.4 would imply $\mathcal{P} = \mathcal{NP}$.

■

In order to overcome this obstacle, we need to seek a different kind of approximation to the hypergraph matching problem. Typically, an approximate solution is a feasible solution, i.e. one that satisfies all the constraints, but whose objective value is not the best possible. We will look for a different type of solution, one that satisfies a *relaxed* set of constraints. We will then show how to turn a solution that satisfies the relaxed set of constraints into a schedule for the network scheduling problem, while only introducing a bounded amount of error into the quality of the approximation.

We will assume for now that $p_{\max} \le n^3$. This implies that the size of program $\mathcal{I}$ is polynomial in the input size. We will later show how to dispense with the assumption on the size of $p_{\max}$ via a number of rounding and scaling techniques.

We begin by constructing approximate solutions to the following integer linear program:

$$\begin{aligned}
\sum_{j \in e} x_e &= 1 & j = 1\ldots, n \qquad (6)\\
\sum_{(i,t) \in e} x_e &\le 1 & \forall (i,t) \in M \qquad (7)\\
\sum_{e} x_e c_e &\le C & \qquad (8)\\
x_e &\in \{0, 1\}.
\end{aligned}$$

This integer program is a packing integer program, and as has been show by Raghavan [22], Raghavan and Thompson [23] and Plotkin, Shmoys and Tardos [21], it is possible to find provably good approximate solutions in polynomial time. We briefly review the approach of [21], which yields the best running times.

Plotkin, Shmoys and Tardos [21] consider the following general problem.

The Packing Problem: $\exists ? x \in P$ such that $Ax \le b$, where A is an $m \times n$ nonnegative matrix, $b > 0$, and P is a convex set in the positive orthant of R^n.

They demonstrate fast algorithms that yield approximately optimal integral solutions to this linear program. All of their algorithms require a fast subroutine to solve the following problem.

The Separation Problem: Given an m-dimensional vector $y \ge 0$, find $\tilde{x} \in P$ such that $c\tilde{x} = \min(cx : x \in P)$, where $c = y^t A$.

The subroutine to solve this problem will be called the *separating subroutine.*

An approximate solution to the packing problem is found by considering the relaxed problem

$$\exists ? x \in P \text{ such that} Ax \le \lambda b$$

and approximating the minimum λ such that this is true. Here the value λ characterizes the "slack" in the inequality constraints, and the goal is to minimize this slack.

Our integer program can be easily put in the form of a packing problem; the equality constraints (6) define the polytope P and the inequality constraints (7,8) make up $Ax \le b$. The quality of the integral solutions obtained depends on the *width* of P relative to $Ax \le b$, which is defined by

$$\rho = \max_i \max_{x \in P} \frac{a_i x}{b_i}$$

It also depends on d, where d is the smallest integer such that any solution returned by the separating routine is guaranteed to be an integral multiple of $\frac{1}{d}$.

The width of the polytope P from equations (6) is at least n, which as we will see would yield integral solutions of poor quality. Polytope P, however, can be decomposed into n different polytopes: $P = P^1 \times P^2 \times \ldots P^n$, where P^j corresponds to those equality constraints which include only J_j. In order to keep the width of the P^i small, we also include into the definition of P^j the constraint $x_e = 0$ for each edge e which includes J_j and has $c_e > C$; this does not increase the optimal value of the integer program. We integrate each of these new constraints into the appropriate polytope P^j, and decompose $x = (x^1, x^2, \ldots x^n)$, where x^j consists of those components of x which represent edges that include J_j. In other words, P^l is defined by

$$\begin{array}{rcll} \sum_{J_l \in e} x_e & = & 1 & \\ x_e & = & 0 & \text{if } c_e > C \text{ and } J_l \in e. \end{array}$$

This yields the following relaxation $\mathcal{L}$:

$$\begin{array}{llcll} \text{minimize } \lambda & & & & \\ \text{subject to} & & & & \\ & x^l & \in & P^l & 1 \leq l \leq n \\ & \sum_{(i,t) \in e} x_e & \leq & \lambda & \forall (i,t) \in M \\ & \sum_e x_e c_e & \leq & \lambda C & \\ & x & = & (x^1, x^2, \ldots x^n) \in \{0,1\}^{|F|}. & \end{array} \tag{9}$$

Let λ^* be the optimum value of $\mathcal{L}$. The following theorem is a specialization of Theorem 2.11 in [21] to our problem, and describes the quality of integral solutions that can be obtained for such integer programs.

Theorem 3.5 *[21] Let ρ^l be the width of P^l and $\bar{\rho} = \max_l \rho^l$. Let $\gamma = |M|$, and $\lambda' = \max(\lambda^*, (\bar{\rho}/d) \log \gamma)$. Given a polynomial-time separating subroutine for each of the P^i, there exists a polynomial-time algorithm for $\mathcal{L}$ which gives an integral solution with $\lambda \leq \lambda^* + O(\sqrt{\lambda'(\bar{\rho}/d) \log(\gamma n d)})$.*

To apply this theorem we must (1) demonstrate a polynomial-time separating subroutine and (2) calculate $\bar{\rho}$, d and γ. The decomposition of P into n separate polytopes makes this task much easier. The separating subroutine must find $x^l \in P^l$ that minimizes cx^l; however, since the vector that is 1 in the eth component and 0 in all other components is in P^l for all e such that $J_l \in e$ and $c_e \leq C$, the separating routine reduces merely to finding the minimum component $c_{e'}$ of c and returning the vector with a 1 in position e' and 0 everywhere else. An immediate consequence of this is that $d = 1$. Recall as well that the assumption that $p_{\max} \leq n^3$ implies that γ is upper bounded by a polynomial in n.

It is not hard to see that $\bar{\rho}$ is 1; therefore

$$\begin{aligned} \lambda &\leq \lambda^* + O(\sqrt{(\bar{\rho}/d) \log \gamma (\bar{\rho}/d) \log(\gamma n d)}) \\ &\leq 1 + O(\log n) = O(\log n). \end{aligned}$$

By employing binary search over C and the knowledge that the optimal solution has $\lambda = 1$, we can obtain an invalid "schedule" in which as many as $O(\lambda)$ jobs are scheduled at one time. If $p_{\max}$ is polynomial in n and m then we have a polynomial-time algorithm; therefore we have proven the following lemma.

Lemma 3.6 *Let C^* be the solution to the integer program $\mathcal{I}$ and assume that $|M|$ is bounded by mn^4. There exists a polynomial-time algorithm that produces a solution x^* such that*

$$\begin{aligned} \sum_{j \in e} x_e^* &= 1 & j = 1 \ldots, n \\ \sum_{(i,t) \in e} x_e^* &= O(\log n) & \forall (i,t) \in M \\ \sum_e x_e^* c_e &= O(C^* \log n) \\ x_e^* &\in \{0,1\} \end{aligned} \tag{10}$$

This relaxed solution is not a valid schedule, since $O(\log n)$ jobs are scheduled at one time; however, it can be converted to a valid schedule by use of the following lemma.

Lemma 3.7 *Consider an invalid schedule S for a set of jobs with release dates on m unrelated parallel machines, in which at most λ jobs are assigned to each machine at any time. If W is the average weighted completion time of S, then there exists a schedule of average weighted completion time at most λW, in which at most one job is assigned to each machine at any time,*

Proof: Consider a job J_j scheduled in S; let its completion time be C_j^S. If we schedule the jobs on each machine in the order of their completion times in S, never starting one before it's release date, then in the resulting schedule

1. J_j is started no earlier than its release date,
2. J_j finishes by time at most λC_j^S.

Statement 1 is true by design of the algorithm. Statement 2 is true since at most $\lambda C_j^S - p_{ij}$ work from other jobs can complete no later than C_j^S in schedule S, and jobs run simultaneously in schedule S can run back-to-back with no intermediate idle time in our expanded schedule. Therefore, job J_j is started by time $\lambda C_j^S - p_{ij}$ and completed by time λC_j^S. ■

Combining the last two lemmas with the observation that $p_{\max} \leq n^3$ implies $|M| \leq mn^4$ yields the following theorem.

Theorem 3.8 *There is a polynomial-time $O(\log^2 n)$-approximation algorithm for the minimization of average weighted completion time of a set of jobs with machine-varying release dates on unrelated machines, under the assumption that the maximum job sizes are bounded by $p_{\max} \leq n^3$.*

3.4 Large Jobs

Since the p_{ij} are input in binary and in general need not be polynomial in n and m, the technique of the last section can not be applied directly to all instances, since it would yield superpolynomial-size formulations. Therefore we must find a way to handle very large jobs without impacting significantly on the quality of solution.

It is a standard technique in combinatorial scheduling to partition the jobs into a set of large jobs and a set of small jobs, schedule the large jobs, which are scaled to be in a polynomially-bounded range, and then schedule the small jobs arbitrarily and show that their net contribution is not significant, (see e.g. [24]). In the minimization of average weighted completion time, however, we must be more careful, since the small jobs may have large weights and can not be scheduled arbitrarily.

We employ several steps, each of which increases the average weighted completion time by a small constant factor. With more care we could reduce the constants introduced by each step; however since our overall bound is $O(\log^2 n)$ we dispense with this precision for the sake of clarity of exposition.

The basic idea is to characterize each job by the minimum value, taken over all machines, of its (release date + processing time) on that machine. We then group the jobs together based on the size of their minimum $r_{ij} + p_{ij}$. The jobs in each group can be scaled down to be of polynomial size and thus we can construct a schedule for the scaled down versions of each group. We then scale the schedules back up, correct for the rounding error, and show that this does not affect the quality of approximation by more than a constant factor. We then apply Lemma 3.9 (see below) to show that the makespan can be kept short simultaneously.

The resulting schedules will be scheduled consecutively. However, since we have kept the makespan from growing too much, we have an upper bound on the start time of each subsequent schedule and thus we can show that the the net disturbance of the initial schedules to the latter schedules will be minimal.

We now proceed in greater detail. Let $m(J_j) = \min_i(p_{ij} + r_{ij})$, and $\mathcal{J}^i = \{J_j | n^{i-1} \leq m(J_j) \leq n^i\}$. Note that there are at most n nonempty $\mathcal{J}^i$, one for each of the n jobs. We will employ the following lemma in order to keep the makespan from growing too large.

Lemma 3.9 *A schedule S for $\mathcal{J}^k$ can be converted, in polynomial time, to a schedule T of makespan at most $2n^{k+1}$ such that $C_j^T \leq 2C_j^S \; \forall j \in \mathcal{J}^k$.*

Proof: Remove all jobs from S that complete later than time n^{k+1}, and, starting at time n^{k+1}, schedule them arbitrarily on the machine on which they run most quickly. This will take at most n^{k+1} time, so therefore any rescheduled job J_j satisfies $C_j^T \leq 2n^{k+1} \leq 2C_j^S$. ■

We now turn to the problem of scheduling each $\mathcal{J}^l$ with a bounded guarantee on the average completion time.

Lemma 3.10 *There exists an $O(\log^2 n)$-approximation algorithm to schedule each $\mathcal{J}^l$. In addition the schedule for $\mathcal{J}^l$ has makespan at most $2n^{l+1}$.*

Proof: Let $\mathcal{A}$ be the algorithm referred to in Theorem 3.8. We will use $\mathcal{A}$ to find an approximately optimal solution S^l for each $\mathcal{J}^l$. $\mathcal{A}$ can not be applied directly to $\mathcal{J}^l$ since the sizes of the jobs involved may exceed n^3, so we apply $\mathcal{A}$ to a scaled version of $\mathcal{J}^l$.

For all j such that $J_j \in \mathcal{J}^l$ and for all i, set $p'_{ij} = \lfloor \frac{p_{ij}}{n^{l-2}} \rfloor$ and $r'_{ij} = \lfloor \frac{r_{ij}}{n^{l-2}} \rfloor$. Note that on at least one machine i , for each job J_j, $p'_{ij} \in [0, n^2]$ and $r'_{ij} \in [0, n^2]$.

We use $\mathcal{A}$ to obtain an approximate solution to the scaled version of $\mathcal{J}^l$ of average weighted completion time W. Although some of the p'_{ij} may still be large, Lemma 3.9 indicates that restricting the hypergraph formulation constructed by A to allow completion times no later than time $\lfloor \frac{2n^{l+1}}{n^{l-2}} \rfloor = 2n^3$ can only affect the quality of approximation by at most a factor of 2. Therefore $|M|$, the number of (machine, time) pairs, is $O(mn^3)$. Note that some of the p'_{ij} may be 0, but it is still important to include an edge in the hypergraph formulation for each job of size 0.

Now we argue that interpreting the solution of the scaled instance as a solution to the original instance $\mathcal{J}^l$ does not degrade the quality of approximation by more than a constant factor. The conversion from the scaled instance to the original instance is carried out by multiplying $p^*_{ij} = n^{l-2}p'_{ij}, r^*_{ij} = n^{l-2}r'_{ij}$ (which has no impact on quality of approximation) and then adding to each r^*_{ij} and p^*_{ij} the residual amount that was lost due to the floor operation.

The additional residual amounts of the release dates contribute at most a total of n^{l-1} time to the makespan of the schedule, since $|r_{ij} - r^*_{ij}| < n^{l-2}$ and therefore the entire contribution to the makespan is bounded above by $n \times n^{l-2} = n^{l-1}$. By a similar argument, the entire contribution of the residual amounts of the processing times to the makespan is bounded above by n^{l-1}.

So in the conversion from p^*_{ij}, r^*_{ij} to p_{ij}, r_{ij} we add at most $2n^{l-1}$ to the makespan of the schedule for $\mathcal{J}^l$. However, n^{l-1} is a lower bound on the completion time of any job in $\mathcal{J}^l$. Therefore, even if this additional time were added to the completion time of every job, the restoration of the residual amounts of the r_{ij} and p_{ij} degrades the quality of the approximation to average completion time by at most a constant factor. Finally, to satisfy the makespan constraint, we apply Lemma 3.9. ■

We now construct two schedules S^{o} and S^{e}. In S^{o} we consecutively schedule $S^1, S^3, S^5, \ldots$ and in S^{e} we consecutively schedule $S^2, S^4, S^6, \ldots$. For the sake of clarity our schedule will have time of length $2n^{i+1}$ dedicated to each S^i even if S^i has no jobs.

Lemma 3.11 *Let $\mathcal{J}^o$ be the set of jobs scheduled in S^o and $\mathcal{J}^e$ the set of jobs scheduled in S^e. The average weighted completion time of S^o is within a factor of $O(\log^2 n)$ of the best possible for $\mathcal{J}^o$, and similarly for S^e and $\mathcal{J}^e$.*

Proof: The subschedule for any set $\mathcal{J}^i$ scheduled in S^{o} or S^{e} begins by time $(2 + o(n))n^{i-1}$, since $\mathcal{J}^i$ is scheduled after $\mathcal{J}^{i-2}, \mathcal{J}^{i-4}, \ldots$ and the makespan of $\mathcal{J}^l$ is at most $2n^{l+1}$. Since n^{i-1} is a lower bound on the completion time of any job in $\mathcal{J}^i$, in the combined schedule S^{o} or S^{e}, each job completes within a small constant factor of its completion time in S^i. ■

We now combine S^{o} and S^{e} by superimposing them over the same time slots. This creates an infeasible schedule in which the sum of completion times is just the sum of the completions times in S^{o} and S^{e}, but in which there may be two jobs scheduled simultaneously. We then use Lemma 3.7 to combine S^{o} and S^{e} to obtain a schedule S^α for all the jobs, whose average weighted completion time is within a factor of $O(\log^2 n)$ of optimal.

Theorem 3.12 *There is a polynomial-time $O(\log^2 n)$-approximation algorithm for the minimization of average weighted completion time of a set of jobs with machine-varying release dates on unrelated machines.*

3.5 Scheduling with Periodic Connectivity

The hypergraph formulation of the scheduling problem can model time-varying connectivity between jobs and machines; e.g. a job can only be processed during certain times on each machine. In this section we show how to apply our techniques to scheduling problems of *periodic* connectivity under some modest assumptions on the length of the period and job sizes.

Definition 3.13 *The* periodic scheduling problem *is defined by n jobs, m unrelated machines, a period P, and for each time unit of P a specification of which jobs are allowed to run on which machines at that time.*

Theorem 3.14 *Let $\mathcal{I}$ be an instance of the periodic scheduling problem in which $p_{\max}$ is polynomial in n and m, and let the optimum makespan of $\mathcal{I}$ be $\mathcal{L}$. There exists a polynomial-time algorithm which delivers a schedule of makespan $O(\log n)(\mathcal{L} + P)$.*

Proof:

As above, we assume that $\mathcal{L}$ is known in advance, and then use binary search to complete the algorithm.

We construct the integer program

$$
\begin{aligned}
\sum_{j \in e} x_e &= 1 && j = 1 \ldots, n && (11) \\
\sum_{(i,t) \in e} x_e &\leq 1 && \forall (i,t) \in M && (12) \\
x_e &\in \{0,1\}.
\end{aligned}
$$

where $M = \{(i,t) | 1 \leq i \leq m, 1 \leq t \leq \mathcal{L}\}$. We include an edge in the formulation if and only if it is valid with respect to the connectivity conditions. We then use Theorem 3.8 to produce a relaxed solution that satisfies

$$
\begin{aligned}
\sum_{j \in e} x_e^* &= 1 && j = 1 \ldots, n \\
\sum_{(i,t) \in e} x_e^* &= O(\log n) && \forall (i,t) \in M \\
x_e^* &\in \{0,1\}
\end{aligned}
$$

Let the length of this relaxed schedule be L; $L \leq \mathcal{L}$. We construct a valid schedule of length $O(\log n)(L + P)$ by concatenating $O(\log n)$ blocks of length L. At the end of each block we will have to wait until the start of the next period to begin the next block; hence we obtain an overall bound of $O(\log n)(L + P)$. ■

Note that we are assuming that the entire connectivity pattern of P is input explicitly; if it is input in some compressed form then we must assume that P is polynomial in n and m.

One motivation for such problems is the domain of *satellite communication systems* [18, 26]. One is given a set of sites on Earth and a set of satellites(in Earth orbit). Each site generates a sequence of communication requests; each request is potentially of a different duration and may require communication with any one of the satellites. A site can only transmit to certain satellites at certain times, based on where the satellite is in its orbit. The connectivity pattern of communication opportunities is periodic, due to the orbiting nature of the satellites.

The goal is to satisfy all communication requests as quickly as possible. We can use our hypergraph formulation technique to give an $O(\log n)$-approximation algorithm for the problem under the assumption that the p_j are bounded by a polynomial, since the rounding techniques do not generalize to this setting.

Acknowledgments We are grateful to Phil Klein for several helpful discussions early in this research, to David Shmoys for several helpful discussions, especially about the upper bound for average completion time, to David Peleg and Baruch Awerbuch for explaining their off-line approximation algorithm to us, and to Perry Fizzano for reading an earlier draft of this paper.

References

[1] N. Alon, G. Kalai, M. Ricklin, and L. Stockmeyer. Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 334–343, 1992.

[2] B. Awerbuch, S. Kutten, and D.Peleg. Competetive distributed job scheduling. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 571–580, 1992.

[3] J.L. Bruno, E.G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17:382–387, 1974.

[4] X. Deng, H. Liu, J. Long, and B. Xiao. Deterministic load balancing in computer networks. In *Proceedings of 2nd IEEE Symposium on Parallel and Distributed Processing*, 1992.

[5] P. Fizzano, D. Karger, C. Stein, and J. Wein. Job scheduling in rings. In *Proceedings of the 1994 ACM Symposium on Parallel Algorithms and Architectures*, pages 210–219, 1994.

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

[7] R.L. Graham. Bounds on multiprocessing anomalies. *SIAM Journal of Applied Mathematics*, 17:263–269, 1969.

[8] D. Gusfield. Bounds for naive multiple machine scheduling with release times and deadlines. *Journal of Algorithms*, 5:1–6, 1984.

[9] L. Hall and D. B. Shmoys. Approximation schemes for constrained scheduling problems. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 134–141. IEEE, October 1989.

[10] D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.

[11] D.S. Hochbaum and D.B. Shmoys. A polynomial approximation scheme for machine scheduling on uniform processors: using the dual approximation approach. *SIAM Journal on Computing*, 17:539–551, 1988.

[12] W. Horn. Minimizing average flow time with parallel machines. *Operations Research*, 21:846–847, 1973.

[13] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[14] T. Kawaguchi and S. Kyan. Worst case boundof an lrf schedule for the mean weighted flow-time problem. *SIAM Journal on Computing*, 15:1119–1129, 1986.

[15] E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, and D.B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S.C. Graves, A.H.G. Rinnooy Kan, and P.H. Zipkin, editors, *Handbooks in Operations Research and Management Science, Vol 4., Logistics of Production and Inventory*, pages 445–522. North-Holland, 1993.

[16] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[17] J.K. Lenstra, D.B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[18] J. H. Lodge. Mobile satellite communication systems: Toward global personal communications. *IEEE Communications Magazine*, 30(24):24–31, November 1991.

[19] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 510–514, 1988.

[20] D. Peleg, 1992. Private communication.

[21] S. Plotkin, D. B. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. To appear in Math of Operations Research, 1995.

[22] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *Journal of Computer and System Sciences*, 37:130–143, 1988.

[23] P. Raghavan and C. D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365–374, 1987.

[24] D. B. Shmoys, C. Stein, and J. Wein. Improved approximation algorithms for shop scheduling problems. *SIAM Journal on Computing*, 23(3):617–632, June 1994.

[25] D. B. Shmoys and E. Tardos. Scheduling parallel machines with costs. In *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 448–455, January 1993.

[26] P. Wood. Mobile satellite services for travelers. *IEEE Communications Magazine*, 30(11):32–35, November 1991.

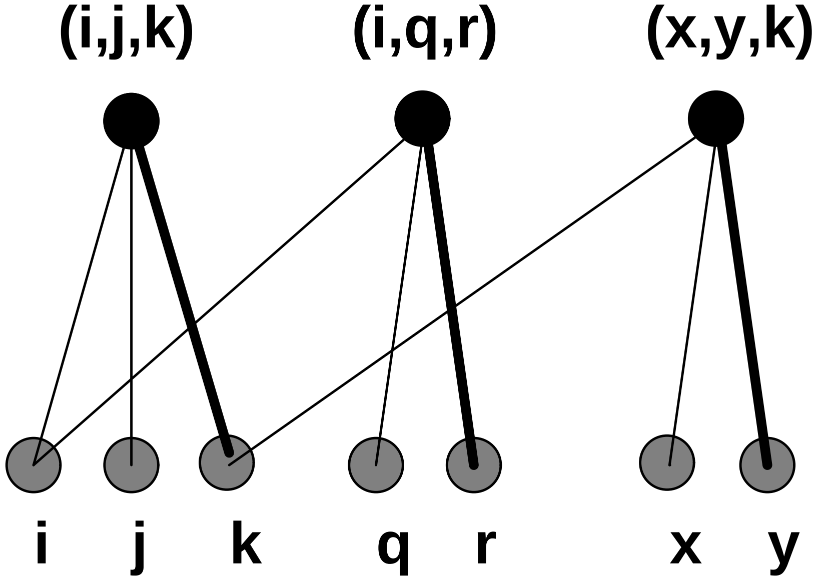

Figure 3: Subgraph of $\mathcal{N}$ corresponding to two triples (i, j, k) and (p, q, r).
Dark edges correspond to edges of length 2.

Appendix

Proof of Theorem 2.4 The reduction is similar to the techniques used by Lenstra, Shmoys, and Tardos [17] to show that no algorithm can approximate the optimal makespan for unrelated parallel machines by better than a factor of $\frac{3}{2}$ unless $\mathcal{P} = \mathcal{NP}$.

Let A, B, C be disjoint sets, each with n elements, and let T be a set of m *triples*, $T = \{(a_i, b_j, c_k) : a_i \in A, b_j \in B, \text{and } c_k \in C\}$. We say that triple (a_i, b_j, c_k) *covers* a_i, b_j and c_k, and define a *perfect matching* as a set of n triples that covers every element of A, B and C exactly once. The problem of determining whether there exists a perfect matching given A, B, C, T is known as 3-dimensional matching and is $\mathcal{NP}$-complete [13]. We will refer to this problem as 3DM.

We will convert an instance $M = (A, B, C, T)$ of 3DM to an instance $(G, \ell, \mathcal{J})$ of the network scheduling problem that has a schedule of length 3 if and only if instance M has a perfect matching. We construct $\mathcal{N} = (G, \ell, \mathcal{J})$ as follows. To construct $G = (V, E)$, we associate a machine with each triple $t \in T$ (the *triple machines*) and a machine with each element of sets A, B, and C (the *A machines, B machines*, and *C machines* respectively). Thus there are $3n + m$ machines. For each triple $t = (a_i, b_j, c_k)$, we create three edges: one from machine t to machine a_i of length 1, one from machine t to machine b_j of length 1, and one from machine t to machine c_k of length 2 (See Figure 12). This yields a network with $3m$ edges.

(In order to obtain a construction with only unit-length edges we introduce new nodes v_e, one for each edge of length 2, and replace each edge e from t to c_k by a path t to v_e to c_k. Each node v_e receives a job of size 3 at time 0. Clearly in a schedule of length 3 this functions exactly as an edge of length 2, so for ease of exposition we use edges of length 2.)

The initial job distribution $\mathcal{J}$ is defined as follows. For each element $a_i \in A$, let $t(a_i)$ be the number of triples which contain element a_i. On A machine a_i we place $t(a_i)$ jobs:

1. $t(a_i) - 1$ jobs J_j with $p_j = 2$, the *dummy jobs*,

2. 1 job J_j with $p_j = 3$

On each B machine, we place 2 jobs:

1. 1 job J_j with $p_j = 1$,
2. 1 job J_j with $p_j = 3$.

On each C machine, we place 2 jobs:

1. 1 job J_j with $p_j = 1$,
2. 1 jobs J_j with $p_j = 3$.

The basic idea behind the construction is that in any schedule of length 3, each machine corresponding to triple (a_i, b_j, c_k) runs one of only two possible schedules: either one dummy job from machine a_i (a *dummy schedule*), or the two unit-size jobs from machines b_j and c_k respectively (a *matching schedule*). Each machine a_i is adjacent to exactly one machine running the latter schedule, and therefore these machines correspond to a perfect matching. If there is no perfect matching, a schedule of length 3 cannot exist. We now proceed in more detail.

We first argue that if M has a perfect matching, then the corresponding network scheduling problem $\mathcal{N}$ has a schedule of length 3. Each A, B, and C machine runs its job of size 3 from time 0 to 3. The remaining $m - n$ jobs of size 2 from the A machines ($t(a_i) - 1$ from machine a_i), the n unit-size jobs from the B machines and the n unit-size jobs from the C machines are scheduled as follows. Let $T_p \subset T$ be the perfect matching. Each machine corresponding to $t \in T_p$ runs a matching schedule, specifically the unit-size job from machine b_j and the unit-size job from machine c_k. Since these jobs are available to their triple machines at times 1 and 2 respectively, this schedule is feasible, and all jobs starting on B or C machines have been scheduled. Because the matching T_p contains exactly one triple for each a_i, there are $t(a_i) - 1$ unutilized machines adjacent to machine a_i. Each such machine runs one of the size-2 jobs from machine a_i starting at time 1. Since any job starting on machine a_i can arrive at these machines at time 1, this schedule is feasible and all jobs originating on the A machines have been scheduled. Therefore we have scheduled every job validly in 3 units of time.

We now show that if instance $\mathcal{N}$ has a schedule of length 3, then 3DM instance M has a perfect matching. We argue that any schedule of length 3 must have the form described above where each triple machine runs either a matching schedule or a dummy schedule; the set of machines running matching schedules correspond to a perfect matching for instance M.

First observe that in the schedule created above, each machine started processing a job as early as possible, and then was busy until the schedule completed. We call this property the *non-idleness* property. Clearly in any schedule of length 3 each A, B, and C machine must run only the size 3 job that originates there. In addition, a simple counting argument shows that the triple machines are idle for one unit of time and must be busy the remainder of the time. Thus in any schedule of length 3, all the $m - n$ size-2 dummy jobs from the A machines, n unit-size jobs from the B machines and the n unit-size jobs from the C machines must be run by the triple machines.

We also observe that each job that is not run on its originating machine must run on an adjacent machine. We call this the *locality* property. The size 2 dummy jobs cannot travel more than 1 unit away in a length 3 schedule. Because all edges adjacent to each C machine have length 2, these unit-size jobs cannot travel more than one edge in a length-3 schedule. Finally, the unit-size jobs from the B machines must travel distance at least 3 to reach a nonadjacent triple machine, which is impossible in a schedule of length 3.

We now argue that each triple machine must run either a dummy schedule or a matching schedule. Each A machine a_i must send all of its $t(a_i) - 1$ size-2 jobs to the triple machines adjacent to it. In a length-3 schedule, no machine can process two size-2 jobs. Therefore, the $t(a_i) - 1$ jobs will be sent to $t(a_i) - 1$ distinct triple machines. They will all run from time 1 through 3, and hence no other jobs can run on those $t(a_i) - 1$ machines.

There are now exactly n triple machines not running dummy jobs and by construction of the network each has a different first element (the a_i's are all distinct). There are $2n$ unit-size jobs remaining to be scheduled, so by the nonidleness property, each such machine must run a unit-size job at time 1 and at time 2. Each edge adjacent to a C machine has length 2. Therefore, no job originating at a C machine can be processed elsewhere before time 2. Since there are n such jobs and n triple machines remaining to process them, each triple machine $t = (a_i, b_j, c_k)$ must run a job from a C machine at time 2. Furthermore, this C job must correspond to element c_k by the locality property. Therefore, each triple machine must process the job from the B machine adjacent to it at time slot 1. Therefore, the set of machines which run matching schedules cover all elements of sets A, B, and C. ■

Proof of Theorem 3.1 We show how to convert an instance $M = (A, B, C, T)$ of the 3-dimensional matching problem to an instance $\mathcal{N} = (G, \ell, \mathcal{J})$ of the network scheduling problem. Instance $\mathcal{N}$ will have an average completion time equal to a certain value if and only if instance M has a perfect matching.

We construct $\mathcal{N} = (G, \ell, \mathcal{J})$ as follows. To construct the graph $G = (V, E)$, we associate a machine with each triple $t \in T$ (the triple machines), and a machine with each element of sets A, B, and C. For each triple $t = (a_i, b_j, c_k)$, we create three paths: one from machine t to machine a_i of length 1, one from machine t to machine b_j of length 3, and one from machine t to machine c_k of length 1. On the intermediate nodes of the path of length 3 we place machines (called the path machines), thus yielding a network with $m + 5n$ nodes (machines), and $5m$ edges.

The initial job distribution $\mathcal{J}$ is defined as follows. For each element $x \in A \cup B \cup C$, let $t(x)$ be the number of triples which contain element x. Let $L = 10nm$.

1. On each A machine a_i, we place two jobs, one with processing time 2 and one with processing time L.

2. On each B machine b_j, we place two jobs, each with processing time L.

3. On each path machine, we place a job of length L.

4. On each C machine c_k we place $t(c_k)$ jobs, all of length L.

Let I_j be the total idle time experienced by J_j before being processed. A schedule of minimum average completion time will minimize $\beta = \sum_j I_j$, the sum of the idle times. Because the value of L is so large, an optimal schedule will minimize the number of times quantities with the value L contribute to β. In particular, to avoid any job experiencing an idle time of L, in an optimal schedule once a machine runs a job of size L, it doesn't run any jobs afterwards. Thus, at most one job of size L runs on any machine, and that job must be the last one. Since there are $m + 5n$ jobs of size L and $m + 5n$ machines, every machine must run exactly one size-L job.

We now compute a lower bound on β. First observe that all but one of the jobs that originate on a machine c_k must run on another machine, since no machine can run two jobs of size L. Thus, each of these jobs must travel at least one edge for a total of $m - n$ idle time. Next observe that of the two jobs that start on an A machine a_i, they either both run on a_i, with idle time at least 2, or one runs on another machine for an idle time of at least 1, and n overall. Now consider a B

machine and its associated path machines. The combined idle time of the jobs originating on these machines must be at least 3. Thus we have a lower bound on idle time β of $m-n+n+3n = m+3n$.

We now show that a schedule with total idle time of β can be achieved if and only if there is a perfect 3D matching. If there is a matching, then each of the n triple machines that correspond to a matched edge will run a size-2 job from the A machine at time 1 and one of the size L jobs from the B machine at time 3. The $m-n$ unmatched triple machines will run a job from the corresponding C machine. Since there is a perfect matching there are exactly $t(c_k)-1$ such machines. All other jobs run on their originating machines at time 0, thus giving us a schedule with $\beta = m+3n$.

Now we show that this is the only such schedule of this length and hence must imply that a perfect matching exists. By the above lower bound arguments, each C machine c_k must send out $t(c_k)-1$ jobs, thus contributing at least $t(c_k)-1$ to the idle time. If any of these machines contributes more to the idle time, the total idle time must exceed β. The only way this lower bound can be achieved is for each of these jobs to travel exactly 1 edge and run at time 1. Therefore in any schedule with idle time β, $m-n$ of the jobs of size L from C machines travel to adjacent triple machines and are run at time 1. These triple machines cannot run any other jobs. By construction of the network, the remaining set of triple machines T cover the set C.

Again by the above lower bound arguments, each A machine must contribute at most 1 to the idle time. Keeping both jobs incurs an idle time of 2 and therefore the global lower bound is exceeded. Thus in any schedule with $m+3n$ idle time, exactly one of the jobs from each A machine travels exactly 1 unit of time and is run at time 1. It must be the job of size 2, because each A machine must run a job of size L. Because the only adjacent machines are triple machines, all of the size-2 A jobs run on adjacent triple machines at time 1. Because there are exactly n machines in set T, each running exactly one A job, the set T covers set A.

Now consider a B machine and its associated path machines. The lower bound argument above shows that the combined idle time of the jobs originating on these machines must be at least 3. There are many ways to achieve this amount of idle time, each one places a job of size L on a triple machine at time x, where $x \in \{1,2,3\}$. But by the arguments above about the placement of the A jobs, we see that the size-L job that makes it from one of the B or path machines cannot run before time 3. It is straightforward to show that in an optimal schedule, a job arrives from a B or path machine at a triple machine at exactly time 3, and this job must run immediately upon arrival (otherwise the idle time would exceed 3). Therefore, each triple machine in T processes exactly one size-L job from a B machine, as this is the only job that can arrive at exactly time 3 without causing any additional idle time. This is only possible if set T covers set B. Thus the set of triples in T is a perfect matching. ■