Scheduling Tasks on Parallel Machines with Network-Based Restrictions

Dmitriy Drusvyatskiy, Robert W.H. Fisher, and Joel Wein*

September 26, 2007

Abstract

We consider the (NP-Complete) problem of *task scheduling with restrictions*, in which each job j has processing time p_j on a certain subset of a set of parallel machines, and may not be processed on the other machines; the subset of machines may differ for each job. A 2-approximation algorithm for this problem has been known for close to twenty years; no algorithm can achieve better than a $\frac{3}{2}$-approximation unless $P = NP$.

In the first part of this paper we consider two special cases of this problem, in which the restrictions can be modeled as a *permissibility graph*. When the permissibility graph is a leveled hierarchy, modelling machines or workers with hierarchical capabilities, we give a $\frac{4}{3}$-approximation algorithm. When the permissibility graph is a tree, we give a very simple and intuitive 2-approximation algorithm.

The task scheduling with restrictions problem has seen a variety of applications, notably as a key subroutine in algorithms for *task scheduling in networks*. In the second part of this paper we consider a natural special case of that problem, in which the network is modeled by one latency parameter, and give both centralized control and distributed approximation algorithms for task scheduling in that model.

*Department of Computer and Information Science, Polytechnic University, 5 Metrotech Center, Brooklyn, NY 11201. Contact author: wein@poly.edu. Partially supported by NSF Grant 0430444 and Polytechnic University's Othmer Institute for Interdisciplinary Studies. This work is part of the undergraduate honors research of Dmitriy Drusvyatskiy and Robert W.H. Fisher, as performed as part of the Othmer Institute Undergraduate Summer Research Program.

1 Introduction

One of the more notorious open problems in the theory of approximation algorithms for combinatorial scheduling is that of *scheduling jobs on parallel unrelated machines.* In this NP-complete problem we are given n jobs and m parallel machines, and job j's processing time p_{ij} may differ depending on the machine on which it is run. Close to twenty years ago Lenstra, Shmoys and Tardos gave a polynomial-time 2-approximation algorithm as well as a hardness result of $\frac{3}{2}$ [8], and these results have not been significantly improved in the interim.

An interesting special case of this problem that captures much of the difficulty is that of *task scheduling with restrictions*, in which each job j has processing time p_j on a certain subset of the parallel machines, and may not be processed on the other machines; the set of machines may differ for each job. In other words, a job j's processing time on machine i, p_{ij}, is either p_j or ∞. The best known asymptotic upper and lower bounds for this problem as well are 2 and $\frac{3}{2}$.

In the first part of this paper we consider two special cases of the problem of task scheduling with restrictions, in which the restrictions can be modeled as a *permissibility graph*. When the permissibility graph is a leveled hierarchy, modelling machines or workers with hierarchical capabilities, we give a $\frac{4}{3}$-approximation algorithm.

When the permissibility graph is a tree, we give a very simple 2-approximation algorithm that does not rely on linear programming.

The task scheduling with restrictions problem has been applied to the problem of *task scheduling in networks*. In this problem the parallel machines are at the nodes of a graph, and a job originates on one node of this graph. In order to be processed on a different machine, a job must travel through the network to the other node before being processed. Several groups of authors have considered both centralized control and distributed-control algorithms for this problem, e.g. [1, 5, 9, 10, 11]. In particular, Philips, Stein and Wein gave a 2-approximation algorithm for this problem that uses as a subroutine an instance of the task scheduling with restrictions problem [9].

In the second part of this paper we consider a naturally-motivated special case of task scheduling in networks in which the communications network that connects the parallel machines is modeled by one latency parameter which represents the time it takes for a message to travel between any pair of nodes in the system; this model is sometimes referred to as the "postal model" for modelling distributed memory systems, e.g. [2, 3, 4]. We give a centralized control $\frac{4}{3}$-approximation algorithm for the problem and a distributed control $\frac{7}{3}$-approximation algorithm.

1.1 Notation

We first introduce some notation. For all our scheduling problems, we will let J denote the set of jobs and M the the set of machines. The cardinality of these two sets is $|J| = n$ and $|M| = m$ respectively. The *makespan* of a schedule is defined as the latest completion time of any job in the schedule, and we denote this by C_{max}. For a particular instance, C^*_{max} represents the length of the smallest makespan that could be constructed for that instance. We will assume throughout the paper that for every job j its total processing time, p_j, is an integral value. We will denote the set of jobs assigned to be processed on some machine i as A_i. For some machine i, we will define the set of jobs originating on that machine as J_i.

1.2 Discussion of Results

We will begin by considering two special cases of task scheduling with restrictions. We introduce the notion of a *permissibility graph*, which gives a compact representation of the restrictions. The nodes in the graph will represent machines. Within the directed graph $G = (V, E)$, jobs from the set J originate on some machine in the set M. A job, j, will have a processing time of p_j on any machine to which there is a directed path in G from the job's origin machine. A job cannot be processed on the machines in the network to which there exists no directed path from its origin machine. For example, if the permissibility graph is a directed tree, jobs originating on the root machine of the graph can be processed on any machine, while jobs originating on leaf nodes can only be processed on their origin machine.

1.2.1 Scheduling Jobs with a Hierarchical Permissibility Graph

In our first result we consider scheduling jobs with restrictions that are described by a leveled, hierarchical permissibility graph. A hierarchical permissibility graph is structurally similar to a tree, in that it is broken into a series of levels. However, unlike the tree, we assume that every machine on some level l in the graph has a path to every other machine on that same level l. Additionally, every machine on some level, l, has outgoing, one-directional edges to every machine on level $l + 1$. Every job in J originates on some level in the graph. A job has a running time of p_j on the machines to which there is a path following the directed edges. For example, if a job originates on level l, it can be processed in p_j time on any machine on level l or any other level k, such that $k > l$. However, if there is a level k such that $k < l$, then the jobs originating

on level l have a processing time of ∞ on any machine on level k. An example of the network in question is shown in Figure 1.

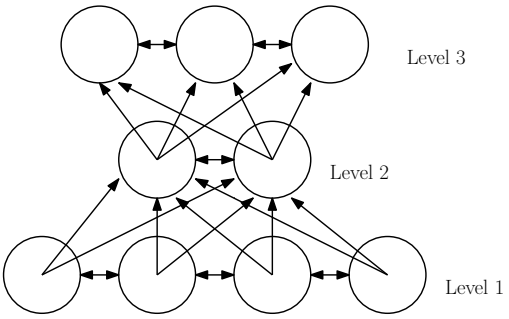

Figure 1: A Hierarchical Permissibility Graph

The jobs that originate on the lowest level, level 1, can be processed anywhere in the network. However a job that originates on the highest level can only be processed on those machines in its origin level. This model could be used to accurately represent workers of different ability levels, with the machines on the highest level being the most able, and the machines on the lowest level being the least able.

Theorem 1.1: *There exists a $\frac{4}{3}-$approximation algorithm for scheduling jobs with restrictions described by a leveled hierarchical permissibility graph.*

1.2.2 Scheduling Jobs with a Tree-Structured Permissibility Graph

We next consider the special case in which the task restrictions are captured by a directed rooted tree. Again, a job j, will have a processing time of p_j on any machine to which there is a path in the tree from the job's origin machine. An example of such a tree is shown in Figure 2. We do not give an improved approximation ratio in this case, but rather a much simpler algorithm that does not rely on linear programming or variants of network flow.

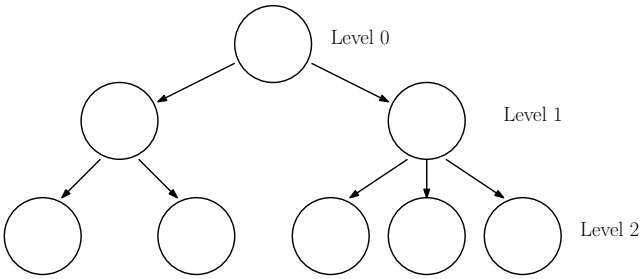

Figure 2: A Tree-Structured Permissibility Graph

Theorem 1.2: *There exists a simple 2-approximation algorithm for scheduling jobs in a tree network.*

1.2.3 Scheduling Jobs in a Network with Constant Latency

We then transition to the problem of scheduling jobs in a network. We note that the network in question here is not related directly to permissibility graphs; rather, in this setting nodes in the graph represent identical machines, each job j originates at some node i, and needs to wait time $d(i,k)$ to be processed on another node k, where $d(i,k)$ is the distance induced by the interconnection network. (The general problem of task scheduling in networks is only related to task scheduling with restrictions in that the latter has been used to give algorithms for the former.)

We consider a special case where the underlying network is completely connected, and there is a constant job transit time of t on any arc. Some job, j, can be processed locally in p_j time on its origin machine. However, if the job is sent to a remote machine, then processing of the job cannot begin until t time has passed since it was scheduled. As noted earlier, several research papers have argued that this is a reasonable way to model interconnection networks in modern distributed memory computer systems.

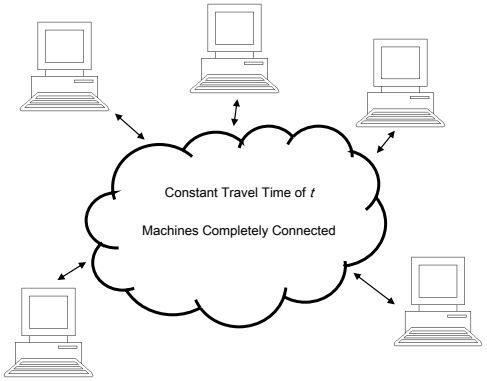

Figure 3: Scheduling in a Constant Latency Network

Theorem 1.3: *There exists a $\frac{4}{3}-$approximation algorithm for scheduling jobs in a network with constant latency.*

Finally, we consider a distributed version of the previous model. In this case, individual machines are aware of the other machines in the network and the network latency required to communicate with other machines. However, an individual machine does not initially know what jobs are originating on other machines, and t time is required for any message to travel from one node to another node of the network. In t time the machines can multicast the size and number of jobs that they originate with to the other machines in the network, or transport any job to another machine for remote processing. We assume that the jobs must have a specific destination machine for transportation and cannot be multicast. There is no centralized control for the machines in the system, so if a machine makes any decisions based on local information, it must transmit its findings to the other machines.

Theorem 1.4: *There exists a non-centralized algorithm that will always produce a schedule with a makespan of length at most $\frac{4}{3}C^*_{max} + t$ for the constant network latency model.*

1.3 Related Work

Lenstra, Shmoys and Tardos's work on unrelated machines scheduling directly gave a 2-approximation algorithm for the special case of task scheduling with restrictions, as well as a proof that no polynomial-time algorithm can be better than a $\frac{3}{2}$-approximation algorithm unless $P = NP$ [8]. The upper bound was improved slightly by Gairing, Lcking, Mavronicolas, and Monien, who gave a $2 - \frac{1}{w}$ approximation algorithm, where w is the size of the largest job [6]. Their algorithm is also fairly simple and allows the schedule to be converted into a Nash equilibrium without increasing the approximation factor.

Task Scheduling in Networks was first studied by Awerbuch, Kutten and Peleg who gave distributed-control algorithms with polylogarithmic performance guarantees for general interconnection networks [1]. Philips, Stein and Wein gave a 2-approximation algorithm for the problem as well as a $\frac{4}{3}$ hardness result. Other groups have looked at special interconnection networks such as the ring [5, 11] or different optimality criteria [10].

The rest of this paper is organized as follows. In Section 2 we give our results for the hierarchical permissibility graph, and in Section 3 we give our results for the tree-structured permissibility graph. Section 4 begins with our results for scheduling jobs in a network with constant latency. We then describe our non-distributed algorithm, as well as a hardness bound for the distributed case.

2 Scheduling Jobs With Leveled Hierarchical Restrictions

We will consider job scheduling with restricted assignments by modeling our permissability graph as a leveled hierarchy. Each machine in the system will be assigned to some level, and every job will have an associated origin level. In this model, jobs may only be assigned to machines on their origin level or higher. For an instance of the problem, we will use s to denote the number of levels. We will say that J_l and M_l are the sets of jobs and machines on level l, respectively. We will define the set N_i as the set of machines that

jobs originating on machine i can be processed on. In the leveled hierarchy permissability graph, if a machine i is on level l, then N_i consists of all machines on level l or higher. We introduce two sets of large jobs as follows: $F^1 = \{j : \frac{2}{3}\hat{D} < p_j\}$ and $F^2 = \{j : \frac{1}{3}\hat{D} < p_j \leq \frac{2}{3}\hat{D}\}$, where $\hat{D}$ is the estimate for the optimal makespan. We will say that the set of all large jobs, F, is defined as $F = F^1 \cup F^2$. We say that F_l is the set of jobs in F that originate on level l.

This algorithm will use a $\rho-$relaxed decision procedure [7] based on an estimate for the optimal makespan. We will use binary search in the search phase to modify $\hat{D}$. After every new assignment to $\hat{D}$, we will run the $\rho-$relaxed decision procedure, which will either return an assignment of jobs to machines or report failure.

The $\rho-$RDP of this algorithm will consist of two main steps. In step 1, we will use bottom-up iteration to assign all jobs from the set, F. In step 2, we will use top-down iteration to assign the remaining small jobs. The leveled hierarchy algorithm will consider all levels in the permissability graph during both of these steps. The following is a basic overview of the main routines of the algorithm.

- Binary Search for $\hat{D}$

  ∘$\rho-$Relaxed Decision Procedure:

  1. Assign all jobs from the set F to machines
  2. Assign all jobs from the set $J - F$ to machines

The steps of the $\rho-$RDP are outlined below:

1. We will begin on the bottom level, 1, and repeat the following procedure for each of the higher levels in the network. Assuming that the algorithm is acting on some level l, there will be $|M_l|$ machines on this level. The algorithm will identify the elements of F_l^1 and F_l^2. The algorithm will consider all the machines in M_l, in any order, and assign to each machine the following job or jobs: $\max(\max_{a \in F_l^1}(p_a), \max_{b \in F_l^2}(p_b) + \max_{c \in F_l^2 - \{b\}}(p_c))$. This means that either the largest unassigned job from F_l^1 or the two largest unassigned jobs from F_l^2 are assigned – whichever assignment requires more processing time will be assigned. If only a single job in F_l^2 remains, and no jobs in F_l^1 remain, then this job from F_l^2 is assigned. After this assignment is made, the job or jobs that have been assigned are removed from the set F. The algorithm repeats this step until every machine in M_l has had some assignment made to it, or until all jobs from F_l have been assigned. If all machines receive some job(s) and $|F_l| > 0$, then the remaining elements of F_l are added to the set F_{l+1}, and the algorithm moves to the level $l + 1$. If the algorithm finishes this phase on level s, and $|F_s| > 0$ after all assignments have been made, then the $\rho-$RDP reports failure.
2. If the leveled hierarchy algorithm did not fail during step 1, then all jobs from F have been assigned to machines. In this second step, we will move through the network level by level, beginning at the highest level, s. If the algorithm is on some arbitrary level, l, then it proceeds by looking at the set of jobs, J_l, that originate at this level in the graph. The jobs in F_l are not considered, because they have already been assigned. We will perform standard list scheduling for these jobs to the machines in N_l. While there is some unassigned job in J_l, we take that job, and assign it to machine i such that $i = \min_{k \in N_l} \sum_{j \in A_k} p_j$. This means that some unassigned job from J_l will be assigned to the machine in level l or higher that has the least amount of work currently assigned to it. We repeat until all jobs in J_l have been assigned. We then proceed to assign the jobs originating on level $l - 1$, but before moving to the next step, we look at the last job to be assigned here. If the starting time of this job is after time $\hat{D}$, then the Leveled Hierarchy algorithm re-

turns failure.

We define W as $W = \sum_{j \in J} p_j$. The binary search for D will be on the interval $[p_j^{max}, W]$, where p_j^{max} is the largest running time of any job. We find the $\hat{D}$ such that the assignment phase on $\hat{D}-1$ fails, but $\hat{D}$ returns a valid assignment. When the algorithm completes, we will have found some assignment of the jobs in the network, such that the makespan of this assignment, C_{max}, satisfies the following statement: $C_{max} \leq \frac{4}{3} C^*_{max}$, where C^*_{max} is the optimal makespan.

2.1 Analysis of Leveled Hierarchy Algorithm

We will now show that the schedule produced by the Leveled Hierarchy algorithm will be a $\frac{4}{3}-$approximation for the leveled hierarchy structure.

Lemma 2.1 *If the assignment phase of the algorithm succeeds for some $\hat{D}$, then the returned schedule will have a makespan such that $C_{max} \leq \frac{4}{3}\hat{D}$.*

Proof: If an assignment is returned, then both the first and the second phase of assignment succeed. If the first step succeeded, then we know that all jobs in F are assigned to some machine. Furthermore, we know that every machine received jobs in one of the four following ways: a single job from F^1, two jobs from F^2, a single job from F^2, or no jobs from F. Machines with a single job from F^1 will have a workload that is less than $\hat{D}$. Machines with two F^2 jobs will have a workload that is less than $\frac{4}{3}\hat{D}$, because the most work it could receive is two jobs of size $\frac{2}{3}\hat{D}$. The machines with only one job from F^2 will not have more than $\frac{2}{3}\hat{D}$ work. Clearly the machines that receive no jobs will have no work. Therefore, the makespan of the whole system must be $C_{max} \leq \frac{4}{3}\hat{D}$ after this first phase.

If the second step of the assignment succeeds, then we know that no job in the set $J - F$ began after time $\hat{D}$. The largest processing time of any job in $J - F$ is $\frac{1}{3}\hat{D}$, and one of these jobs could start at time $\hat{D}$. Therefore, if the second step succeeds, then all jobs are assigned, and $C_{max} \leq \frac{4}{3}\hat{D}$. □

Lemma 2.2 *If some level l receives w work from level $l-1$ in the first step of assignment and $\hat{D}$ represents a feasible makespan, then level l must receive at least w work in the optimal schedule.*

Proof: First we should note that if $\hat{D}$ is the optimal makespan, then no job from F^1 would ever be placed on a machine with any other job from F, because this would leave the machine with more than D work. Also, no machine will ever have more than two jobs from F^2 because this would also result in a machine with more than D work.

If we complete the first step of assignment for some level l and there are some jobs from F_l being passed to level $l+1$, this means that every machine in M_l has either one job from F_l^1 or two jobs from F_l^2. Therefore, if $\hat{D}$ is feasible, then no schedule optimal could add any single job that is being passed on to level $l+1$ to a machine on level l and still keep the makespan less than $\hat{D}$. We also cannot exchange some job that is being passed to level $l+1$ for a job running on a M_l machine to decrease the work being passed and still keep every machine with less than $\hat{D}$ work. If we could make such an exchange for some machine, m, then this means that machine m could have taken a larger job during its assignment—which contradicts the definition of the algorithm. This means that we could not decrease w, the sum of work being passed from some level, and still have a makespan less than or equal to $\hat{D}$. Therefore, the work we pass at every step is not more than the work passed in the optimal. □

Lemma 2.3: *If the assignment phase fails to return an assignment for some value of $\hat{D}$, then there is no assignment of jobs to machines that will create a makespan of that length.*

Proof: The assignment phase might fail during the first or second step of the assignment. First we will consider the first step, the assignment of the large jobs from the set F. If the first step of the assignment phase fails, then we know

that $|F_s| > 0$ after all assignments have been made. We know that if $\hat{D}$ is a feasible makespan, then we must be able to assign the elements of F^1 to a machine without any other element of F, and that we will never have three or more elements of F^2 on one machine. In either of these cases, the work on one machine would surpass $\hat{D}$. If $|F_s| > 0$ after all assignments have been made, this means that every machine in M_s has either one F^1 job on it or two F^2 jobs. If there are still jobs remaining, then this means that $|F_s^1| + \lceil \frac{|F_s^2|}{2} \rceil > |M_s|$. In this case, there is no way to assign the jobs from F_s to the machines in M_s such that $C_{max} \leq \hat{D}$. However, level s might have received some jobs from the level below. If so, then every machine on level $s-1$ must have either one F^1 or two F^2 jobs. Therefore, on level $s-1$ we know that $|F_{s-1}^1| + \frac{|F_{s-1}^2|}{2} = |M_{s-1}|$. If this layer received jobs from $s-2$, then $s-2$ must also be full, and so on. We continue until we find the highest layer that did not receive jobs from the level below it, call this level l. We know that on all levels, from l to s, every machine has either one F^1 job or two F^2 jobs, and there are jobs from F left-over that originated on level l or above. We can conclude, therefore that $\sum_{i=l}^{s} |F_i^1| + \lceil \frac{|F_i^2|}{2} \rceil > |N_l|$. This means that there is no way to construct a schedule on the substructure defined by the nodes in N_l such that only one F^1 or at most two F^2 jobs are on one machine. This means that $\hat{D}$ is an infeasible makespan.

Next we consider the instance in which the assignment phase fails during the second step. This step is the top-down iterative assignment of the jobs in the set $J - F$. We know that this step has failed, if some job on some level, l, has begun its execution period after time $\hat{D}$. We begin all jobs on the machine with the least load currently on it, so if some job starts after time $\hat{D}$, then all machines in N_l have at least $\hat{D}$ work currently on them. We also know from Lemma 2.2 that if we received some work from level $l-1$, then we must have received at least this much work, even in the optimal schedule. Therefore, if we call W the total work in the machines of N_l, then we know that $W > |N_l|\hat{D}$. Because these machines must run at least W work, some machine in N_l must run for more than $\hat{D}$ time and $\hat{D}$ is infeasible. □

Lemma 2.4: *If the assignment phase fails for some value of $\hat{D}$, then no feasible schedule is of makespan less than $\hat{D}$.*

Proof: If the assignment phase failed during the first step, then some subgraph of machines, N_l, had too many jobs from F to process. Making $\hat{D}$ smaller will never decrease the cardinality of F, and all previous jobs in the set will still be there. Therefore, if the first step of assignment fails for $\hat{D}$, then the first step will still fail for $\hat{D} - c$ for any constant c.

If the second step of assignment fails, this essentially means that some job from the set $J - F$ was forced to start after time $\hat{D}$. Substituting $\hat{D} - c$ for $\hat{D}$, will result in failure for the same reason. Therefore, when the search phase completes with some estimate $\hat{D}$, we know that the true optimal makespan is not less than this estimate. □

Theorem 2.5: *The algorithm described here will produce a schedule with a makespan such that $C_{max} \leq \frac{4}{3} C_{max}^*$.*

Proof: We know that the algorithm will return some schedule with a chosen estimate, $\hat{D}$. From Lemma 2.1, we know that the returned makespan will be less than or equal to $\frac{4}{3}\hat{D}$. We also know that the assignment phase on $\hat{D} - 1$ failed, therefore from Lemma 2.4 we know that $\hat{D} < C_{max}^*$. These two facts together tell us that $C_{max} \leq \frac{4}{3} C_{max}^*$. □

3 An Algorithm for Assigning Jobs in a Network Modeled as a Tree

We will now introduce the Tree-Scheduling algorithm for scheduling jobs with restricted assignments. The permissability graph in this problem will be a tree. Once again, we will say that the set N_i is the set of machines that the jobs origi-

nating on machine i can be run on. In this case, N_i is the set of machines in the subtree rooted on machine i in the permissibility graph.

Now we will describe the algorithm. We begin at the lowest level of the tree, call this level l. For each machine i in l, we will list schedule the jobs in J_i to the machines in N_i. This means that we select any job that originated on i and assign it to the machine in N_i that is scheduled to process the smallest sum of work at the time of assignment. After all jobs on all machines on level l have been assigned, we repeat this procedure iteratively up the tree from level $l-1$ to level 0. After all jobs on the root node, level 0, have been assigned, the algorithm is complete.

It should be noted that if the algorithm is assigning jobs from a machine, i, which happens to be a leaf node in the permissability graph, then $N_i = \{i\}$. This means that any job that originated on i must be run on this machine, and the algorithm will behave correctly.

3.1 Analysis of the Tree-Scheduling Algorithm

Observation 3.1: *If D is the length of a feasible schedule, then it must be true that $\forall i \in M, \sum_{k \in N_i} \sum_{j \in J_k} p_j \leq |N_i| D$*

Any job j originating on a machine $k \in N_i$ must be scheduled on a machine in N_i because there is no path from a machine in N_i to a machine in $V - N_i$. And if D is a feasible schedule length, then there must be a way to schedule all these jobs on the machines in N_i so that each one completes by time D. This implies that $\sum_{k \in N_i} \sum_{j \in J_k} p_j \leq |N_i| D$.

Observation 3.2: *The algorithm will not assign any jobs that originated outside of N_i to a machine in N_i until all jobs in the set J_i have been assigned.*

If we consider some machine i, the only jobs that may be run on a machine in N_i that did not originate on a machine in N_i are the jobs that originated above the level that machine i resides on. However, because of the bottom up nature of this algorithm, these jobs have not yet been assigned when i is being processed. Therefore, at this point in the algorithm, all jobs being processed on the machines in N_i also originated on a machine in N_i.

Theorem 3.3: *The schedule created by the Tree-Scheduling algorithm will satisfy the following property: $C_{max} \leq 2(C^*_{max})$*

Proof: If the optimal makespan of the system is C^*_{max}, then it trivially follows that $\forall j \in J\ p_j \leq C^*_{max}$ and $\sum_{j \in J} p_j \leq m(C^*_{max})$. In order to show that the algorithm is a 2-approximation, we need only to prove that no job will ever start after time C^*_{max}. We will show this with a proof by contradiction. Call the bottom layer of the tree l. All the machines on this level are leaves. After our algorithm finishes processing level l, the length of the partial schedule created must be less then C^*_{max}. By Observation 3.2, we know that as the algorithm progresses, and we are assigning jobs from machine i to machines in N_i, any job that has already been scheduled to run on a machine in N_i must have originated on some machine in N_i. If throughout the algorithm, a job j originating on machine i was assigned to run on some machine after time C^*_{max}, then that means that all the machines in N_i must have been busy at least until time C^*_{max}. Otherwise this job would have been assigned to the machine in N_i that had less then C^*_{max} work. Therefore $\sum_{k \in N_i} \sum_{j \in J_k} p_j > |N_i| D$. However, this contradicts Observation 3.1. Therefore we may conclude that no job will begin after time C^*_{max} in the schedule produced by this algorithm. It follows that the schedule produced will have the following property: $C_{max} \leq C^*_{max} + \max_{j \in J} p_j \leq 2C^*_{max}$. □

4 An Algorithm for Job Scheduling in a Network with Constant Latencies

Now we will procede to consider the constant network latency problem with a centralized algorithm. It should first be noted that this problem starts to resemble traditional parallel identical machine scheduling ($P||C_{max}$) as t approaches zero, and there exists a polynomial-time approximation scheme for the $P||C_{max}$ problem[7]. Therefore, we will assume that if $t < \frac{1}{3}D$, we can simply wait for t time and then run the polynomial approximation scheme with some ϵ to easily achieve a $\frac{4}{3}$ approximation. The Single Latency algorithm will be designed for the cases in which $t \geq \frac{1}{3}D$.

We will consider two different sets of machines in this section. The set $X = \{i \in M : \sum_{j \in J_i} p_j \leq t\}$ is the set of machines originating t or less work, and the set $Y = \{i \in M : \sum_{j \in J_i} p_j > t\}$ is the set of machines originating more than t work. We will also say that for all machines, $i \in M$, O_i is the set of jobs from J_i that have not yet been assigned to any machine. The Single Latency algorithm will begin with some value for $\hat{D}$ which will be the hypothesis for the optimal makespan of the system. The algorithm will then attempt to use a $\rho-$relaxed decision procedure based on this hypothesis. The ρ-relaxed decision procedure will either return an assignment of jobs to machines, or it will report failure. We will use binary search to find the value of $\hat{D}$ such that running the ρ-relaxed decision procedure on $\hat{D}$ returns an assignment, but running it on $\hat{D} - 1$ returns failure.

Now we will describe the $\rho-$relaxed decision procedure based on some hypothesis for $\hat{D}$.

1. We know that jobs that have a processing time greater than $\hat{D} - t$ must be run on their origin machines in the optimal schedule in order for $\hat{D}$ to be feasible. Therefore, all such jobs will be immediately assigned in any order to their origin machines. Jobs will be scheduled to run at the earliest possible time. If any machine has a schedule of length greater than $\hat{D}$ at this point, the assignment phase reports that it has failed to create an assignment.

2. We consider all jobs originating on a machine in the set X. For all such jobs, we assign the jobs in any order to their respective origin machines. These jobs are scheduled to be run as soon as their machine becomes available.

3. We now consider all machines on which no jobs were scheduled in steps 1 or 2. We can call one such machine i. If $|F \cap O_i| \geq 2$, then we assign the two largest jobs from F originating on i to be run contiguously on i starting at time zero. If there is only a single job from F originating on i, then this job is scheduled to be run starting at time zero. If there are no jobs from F originating on i, no action is taken. This process is done for all machines that received no jobs in steps 1 or 2.

4. We now consider some machine, i, such that $\sum_{j \in A_i} p_j \leq t$ and $(O_i \cap (J - F)) \neq \emptyset$. As long as there is any such machine, i, we will assign a job from the set $(O_i \cap (J - F))$ to be processed on i at the earliest time possible.

5. We will now consider all the machines that received no jobs in step 1, and 0 or 1 jobs from the set F in steps 2 through 4. We find a mapping of the unassigned jobs in F to these machines, such that only one job from F is processed on any one of these machines. If no such one-to-one mapping exists, the algorithm reports that the assignment phase has failed.

6. The algorithm will now list schedule all unassigned jobs from $J - F$ to any machine in the system, such that every job is assigned

to the machine on which it will begin processing the soonest. If it is discovered that some job from $J - F$ was assigned to begin processing after time $\hat{D}$, the the assignment phase reports failure.

The algorithm will begin with $\hat{D} = p_j^{max} + \frac{\sum_{j \in J} p_j - p_j^{max}}{2}$, where p_j^{max} is the largest job size. We will use binary search over the range $\left[p_j^{max}, \sum_{j \in J} p_j\right]$. If the assignment phase fails in any of the possible failure points, then the algorithm notes that the true value of D must be larger than the current value of $\hat{D}$ and modifies the estimate. If the assignment phase succeeds, then the true value for D is not greater than the value of $\hat{D}$, so the estimate is decreased. Algorithm Single Latency returns the assignment created by the estimate $\hat{D}$ such that trying to run the $\rho-$relaxed decision procedure on $\hat{D} - 1$ returns failure. We will now show that the makespan of this schedule is no greater than $\frac{4}{3}C^*_{max}$.

4.1 Analysis of the Single Latency Algorithm

Lemma 4.1: *If the ρ-relaxed decision procedure of the Single Latency algorithm returns a schedule for some value of $\hat{D}$, then the schedule created has the property: $C_{max} \leq \frac{4}{3}\hat{D}$*

Proof: We will analyze the 6 steps of the assignment phase separately to identify the makespan of the schedule created. Steps 1 and 2 will clearly not exceed a schedule length of size $\hat{D}$. Step 3 may create a schedule of length $\frac{4}{3}\hat{D}$ in the worst case when $t = \frac{1}{3}\hat{D}$. Step 4 could create a schedule of length $t + \frac{1}{3}\hat{D} \leq \frac{4}{3}\hat{D}$.

Now we consider step 5. The jobs from F awaiting assignment must, by the definition of F, have a processing time less than $\hat{D} - t$. We will consider a job j that has been assigned in this step. There are two cases to consider. In case 1, the machine that j is assigned to had no other jobs from F assigned to it in steps 1 through 4. In this case, the total processing time of that machine will include p_j which is not more than $(\hat{D} - t)$, and the work assigned in step 4, which is not more than $t + \frac{1}{3}\hat{D}$. Therefore, we can conclude that in this case, the total processing time of j's machine is at most: $(\hat{D} - t) + t + \frac{1}{3}\hat{D} \leq \frac{4}{3}\hat{D}$. Now we consider the next case, in which j's machine already has a job from F on it. If the machine in question received no more jobs in step four, then the schedule length is clearly no more than $(\hat{D} - t) + (\hat{D} - t) \leq \frac{2}{3}\hat{D} + \frac{2}{3}\hat{D} = \frac{4}{3}\hat{D}$. If the machine received more work in step four, then the job from F had a processing time less than t, and we are essentially back to the first case where the schedule is not greater than $(\hat{D} - t) + t + \frac{1}{3}\hat{D} \leq \frac{4}{3}\hat{D}$.

Finally the sixth step is designed to fail if some job from $J - F$ starts after time $\hat{D}$. Therefore, the makespan will be no greater than $\hat{D} + \frac{1}{3}\hat{D} = \frac{4}{3}\hat{D}$. The $\rho-$relaxed DP is complete after this step, so we can conclude that if it succeeds then it returns a schedule such that $C_{max} \leq \frac{4}{3}\hat{D}$. □

Lemma 4.2: *If the assignment phase fails, then the true optimal makespan C^*_{max} is not less than the estimate that returned failure, $\hat{D}$.*

Proof: The assignment phase might fail in one of three places. The first point of failure in step one is trivial. Any job of size greater than $\hat{D} - t$ must be run on its origin machine in order to produce a schedule of length $\hat{D}$, so if assigning these jobs makes $\hat{D}$ infeasible, then there can be no assignment to satisfy this schedule length. If there is no schedule of length $\hat{D}$, then there is also clearly no schedule of a shorter length.

The second failure point is in step five of the assignment. It should be noted that if $\hat{D}$ is feasible, then there must be some assignment such that every machine is running at most 2 jobs from F, and no machine is running two remote jobs from F. If either of these criteria is violated, then the schedule produced will be of length greater than $\hat{D}$. In step three of the algorithm, we assigned two jobs from F to the origin machine of those jobs wherever possible. These are the jobs from F that will be processed locally, and it would not be possible to construct a schedule of $\hat{D}$ or less where more jobs are processed locally. There-

fore, the number of jobs being run remotely in our schedule is less than or equal to the number of jobs being run remotely in an optimal schedule. We must be able to schedule these jobs such that no two of them run together on one machine, and they only run with at most one other large job. If such an assignment is not possible, then $\hat{D}$ could not be constructed to begin with.

Finally the assignment might fail in the sixth step. If $\hat{D}$ is feasible, then the following fact must hold: $|X|(\hat{D} - t) + |Y|\hat{D} \leq \sum_{i \in Y} \sum_{j \in A_i} p_j$. All the jobs being assigned in step six originated in Y. If some job from $J - F$ is forced to start after time $\hat{D}$, then we know that all machines in the system were busy up until time $\hat{D}$. If this is the case, then the previously stated inequality is reversed, and $\hat{D}$ is infeasible. □

Theorem 4.3: *The makespan of the schedule produced by this algorithm will never be larger than* $\frac{4}{3}(C^*_{max})$

Proof: When the algorithm returns a schedule with estimate $\hat{D}$, we know that $\hat{D} - 1$ fails to produce a schedule. Therefore, we know by Lemma 4.2 that $\hat{D} \leq C^*_{max}$. We also know that $C_{max} \leq \frac{4}{3}\hat{D}$ by Lemma 4.1. Therefore, we conclude that $C_{max} \leq \frac{4}{3}C^*_{max}$. □

4.2 Distributed Version of Single Latency Algorithm

Now we will consider the distributed-control setting. We will remove all centralized control from the Single Latency algorithm and leave only local-control. First, we will demonstrate that no algorithm can provide a better approximation factor than $\frac{3}{2}$ for this problem. Figure 4 shows 3 machines in a network, with corresponding origin jobs.

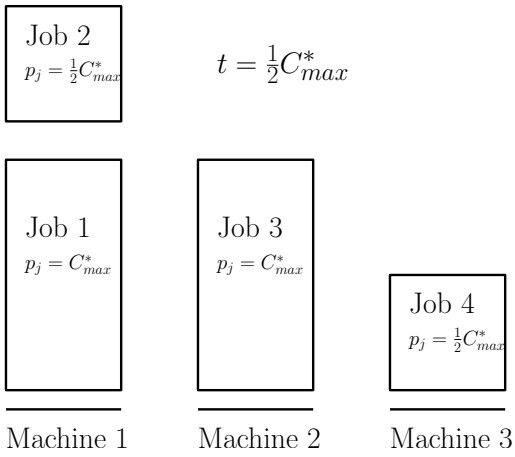

Figure 4: Demonstration of Hardness Bound

In this instance, $t = \frac{1}{2}C^*_{max}$. We will focus on machine 1. In the optimal schedule, job 2 would be sent to machine 3 at time 0. Doing this would produce an optimal makespan of length C^*_{max}. We will consider some algorithm, A. If A selects all of the jobs on machine 1 to be processed locally, then the schedule length clearly cannot be less than $\frac{3}{2}C^*_{max}$. If A chooses to remotely process a job from machine 1, then there are two options. Either A sends jobs from machine 1 for remote processing before time t, or it sends them for processing at time t or later. In the former case, A will try to send jobs without knowledge of the amount of work originating on the destination machine. Therefore, it is clear that even if A chooses only to send job 2, and no other, there will always be a chance that A chooses to send job 2 to a machine with C^*_{max} work originating on it. In the latter case, $C_{max} \geq \frac{3}{2}C^*_{max}$ regardless of the action taken by A. In addition to this $\frac{3}{2}$ hardness bound, we also observe that any algorithm that waits t time before sending jobs out for remote processing cannot guarantee an approximation factor less than 2. This is apparent when t approaches C^*_{max}.

We will now describe how to adapt the Single Latency algorithm for use in this distributed model, at an additive cost of t to the approximation factor. At time t, every machine becomes aware of the rest of the system, so the conditions are identical to the centralized model. We can

hold the machines in an idle state until time t, and then execute the standard Single Latency algorithm. Doing this will result in schedule with a makespan that is at most $\frac{4}{3}C^*_{max} + t$. However, in the case in which $C^*_{max} < t$, no job would travel for remote processing in the optimal schedule. Therefore, for any machine i, such that $\sum_{j \in J_i} p_j \leq t$, we will have i schedule all of its jobs to be processed locally and contiguously starting at time 0. Doing this will produce a schedule with an optimal makespan when $C^*_{max} \leq t$. It also means that for all instances of the problem, this algorithm will produce a schedule length of at most $2\frac{1}{3}C^*_{max}$.

Acknowledgements

We thank Cliff Stein for helpful discussions.

References

[1] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 571–581, 1992.

[2] A. Bar-Noy, J. Bruck, CT. Ho, S. Kipnis, and B. Schieber. Computing global combine operations in the multiport postal model. *IEEE Transactions on Parallel and Distributed Systems*, 6:896–900, 1995.

[3] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. *Springer New York*, 27:431–452, 1994.

[4] A. Bar-Noy and S. Kipnis. Multiple message broadcasting in the postal model. *Networks*, 29:1–10, 1997.

[5] P. Fizzano, D. Karger, C. Stein, and J. Wein. Job scheduling in rings. In *Proceedings of the 6th ACM Symposium on Parallel Algorithms and Architectures*, pages 210–219, 1994.

[6] M. Gairing, T. Lcking, M. Mavronicolas, and B. Monien. Computing nash equilibria for scheduling on restricted parallel links. *36th ACM Symposium on Theory of Computing*, pages 613–622, 2004.

[7] 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.

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

[9] C. Phillips, C. Stein, and J. Wein. Task scheduling in networks. *SIAM Journal on Discrete Mathematics*, 10(4):573–598, 1997.

[10] M. Skutella. Semidefinite relaxations for parallel machine scheduling. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, page 472, 1998.

[11] D. Tsur. Improved scheduling in rings. *Journal of Parallel and Distributed Computing*, 67(5):531–535, 2007.