The Loading Time Scheduling Problem [1]

Randeep Bhatia [2] Samir Khuller [3] Joseph (Seffi) Naor [4]

[1]An extended abstract of this paper appeared in the Proceedings of the 36th IEEE Conference on Foundations of Computer Science, Milwaukee, Wisconsin, (1995), pp. 72-81.

[2]Computer Science Department, University of Maryland, College Park, MD 20742 and LCC Inc., 2300 Clarendon Blvd, Suite 800, Arlington, VA 22201. Email: randeep@cs.umd.edu.

[3]Computer Science Department and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. Research supported by an NSF Research Initiation Award CCR-9307462 and an NSF CAREER Award CCR-9501355. E-mail : samir@cs.umd.edu.

[4]Department of Computer Science, Technion, Haifa 32000, Israel. Research supported in part by Grant No. 92-00225 from the United States-Israel Binational Science Foundation (BSF), Jerusalem. Research also supported by the Technion V.P.R. fund No. 120-882. Israel. E-mail : naor@cs.technion.ac.il.

Abstract

In this paper we study precedence constrained scheduling problems, where the tasks can only be executed on a specified subset of the machines. Each machine has a loading time that is incurred only for the first task that is scheduled on the machine in a particular run. This basic scheduling problem arises in the context of machining on numerically controlled machines, query optimization in databases, and in other artificial intelligence applications. We give the first non-trivial approximation algorithm for this problem. We also prove non-trivial lower bounds on best possible approximation ratios for these problems. These improve on the non-approximability results that are implied by the non-approximability results for the shortest common supersequence problem.

We use the same algorithmic technique to obtain approximation algorithms for a problem arising in the context of code generation for parallel machines, and for the weighted shortest common supersequence problem.

1 Introduction

In this paper we study precedence constrained scheduling problems. The tasks are denoted by the vertex set of an acyclic graph. Precedence constraints are denoted by directed edges in the usual way; an edge from i to j indicates that task i should be completed before task j can be started. Each task needs to be scheduled on one of a specified *subset* of machines (for example, the machines may have different capabilities).

The cost of executing a task can be decomposed into two components. One component is the inherent execution time of the task itself. The other component is a loading time, which is the setup time of the machine we choose to perform the task on. When we perform a set of tasks consecutively on a particular machine, we incur the loading time only for the *first* task performed on the machine.

A schedule which is a feasible sequence of tasks, represents the order of execution of tasks. At any time at most one task is executed, and the schedule specifies the machine on which each task is executed. Every time the next task in the sequence is executed on a different machine than the current one, the loading time of the new machine is incurred.

We call this basic problem the Loading Time Scheduling Problem (LTSP). A special case of this problem was first mentioned by Hayes [8] in the context of machining metal parts. The objective is to start with a block of metal, and to use a numerically controlled machining center to cut a variety of features into the block. Each geometric feature is a task, and there are precedence constraints on the order in which certain tasks can be performed. Different methods may be used to perform the tasks. Each method can be performed on the machining center, which can accomplish a variety of different operations (drilling, end-milling, etc.), but can only perform one operation at a time. When we are able to overlap the machining operations, we do not incur the loading time delay for the machine repeatedly. (For example, when we do two drilling operations consecutively, we only have to load the block of metal on the drilling machine once.) According to Hayes [8], this set-up time is a large fraction of the time for each operation, sometimes as much as 90% of the time is spent in setting up for one machining operation. All other times are relatively small compared to the set-up time.

A second motivation given by Hayes [8] is shown in Fig. 1. (We are not solving the same problem, but this explains some of the intuition behind the loading time scheduling problem.) Suppose we have to run a few errands. The time to do each errand can be decomposed into the time to get to the place where the errand is to be done, together with the time to actually do the task. The time for performing the errand depends on whether we need to go to the location where the errand is to be performed, or, whether we are already there. The optimal solution is to first go home, then to the grocery store, and finally to the post-office. This is a more general problem where there is a "switching" cost between machines. We handle the special case when the switching cost is the sum of the loading and unloading costs.

An extensive survey of "operator overlap" problems in Artificial Intelligence appears in the work by Foulser, Li and Yang [5]. In particular, they discuss a variety of heuristics, with an average case analysis for them, as well as empirical results. Other applications of overlapping operators arise in databases when we try to do multiple query optimization [16].

A problem related to the Loading Time Scheduling Problem is the *Shortest Common Supersequence* problem [6, problem SR8]. Here, a collection of sequences over a fixed alphabet is given, and the goal is to find a shortest common supersequence (SCS), such that all given sequences appear as a subsequence in the common supersequence.

The previous results shown for the SCS problem by Jiang and Li [9] are hardness results for

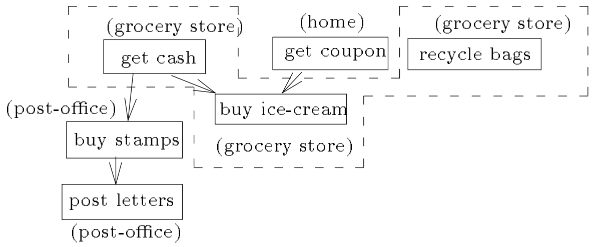

Figure 1: Motivation for the problem.

approximation, and a ρ-approximation algorithm, where ρ is the size of the alphabet. Let n denote the number of sequences. Specifically, Jiang and Li [9] show that (i) SCS does not have a polynomial time constant factor approximation algorithm, unless $P = NP$; (ii) there *exists a constant* δ such that if SCS has a polynomial time approximation algorithm with ratio $\log^{\delta} n$ then $NP \subseteq DTIME(2^{poly \log n})$. They also give algorithms that produce solutions close to the optimal when the supersequences are random (see [9] for more details).

A generalization is the weighted shortest common supersequence(WSCS), where each letter of the alphabet has a weight, and the weight of the supersequence is the sum of the weights of its constituent letters. The WSCS problem is closely related to the LTSP by viewing the alphabet letters as machines, loading times as weights on the alphabets and each sequence as defining precedence constraints between tasks (the precedence graph is a path). The objective functions of the two problems *differ* in how they treat the case when the same letter appears consecutively in a sequence. As an application of our results we show that we can obtain a ρ-approximation algorithm for the WSCS problem.

The literature concerning scheduling problems is very extensive (see e.g., [10]). However, it appears that the specific constraints on the Loading Time Scheduling Problem are very different from the kinds of problems that have been previously considered in the scheduling literature.

A different motivation for our work stems from the design of compilers for multiprocessor architectures that use the fork-join model of parallelism. In a fork-join model, the only operations available for expressing parallelism are *fork* (spawn a vertex's execution as a new thread of control), and *join* (wait for all previously forked threads to complete). The input is a DAG whose vertices represent tasks and whose edges reflect all the control and data dependencies among these instructions. The objective is to generate fork-join parallel code for the DAG, with minimum overall execution time. It is assumed that the parallel code would be executed on a machine with unbounded number of processors and no communication overhead.

Sarkar [15] investigated this problem of generating maximally parallel code using only *fork* and *join* operations to correctly satisfy all the control and data dependences in the program. This problem is of interest when compiling for multiprocessor architectures and runtime systems where fork-join is the only mechanism, or the most efficient mechanism, available for satisfying dependences. In Section 1.2 we provide a detailed description of the problem, and its connection to the Loading Time Scheduling Problem.

1.1 The Loading Time Scheduling Problem

Let the *tasks* be denoted by the vertex set of a directed acyclic graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$. The precedence constraints are denoted in the usual way by directed edges; if there is an edge from i to j then i needs to be done before j.

Suppose there are ρ machines $m_1, \ldots, m_\rho$. Each task i can be performed only on a subset $M(i)$ of the machines (the machines have different capabilities). Each machine m_j has a loading time $\ell(m_j)$. Any task i that can be performed on m_j, ($m_j \in M(i)$), and which satisfies the precedence constraints, may be scheduled with an execution time of $e(i)$. When we perform a set of tasks consecutively on a particular machine, we pay the loading time only *once*. With these constraints, we wish to minimize the total makespan. The execution times for all tasks are fixed (the choice of machine does not affect the execution time), so we can assume that the execution times are zero, and concentrate on minimizing the loading time. More formally,

1. The problem is to partition V into k subsets $V_1 \cup V_2 \ldots \cup V_k$ such that
$$\forall p = 1 \ldots k, \qquad M(V_p) \neq \emptyset.$$
where $M(V_p) = \cap_{i \in V_p} M(i)$. In other words, all tasks assigned to set V_p share at least one machine in common.

2. For each edge $x \to y$ in E, if $x \in V_i$ and $y \in V_j$ we require that $i \leq j$.

3. The goal is to minimize $\sum_{p=1}^{k} \ell(V_p)$, where $\ell(V_p) = \min_{m_k \in M(V_p)} \ell(m_k)$.

In many manufacturing applications [8, 3], typically, $|M(i)| = 1$. The tasks, for example, could be drilling, end-milling, etc. Let the term *job* denote the block of metal mentioned earlier. The following simple heuristic is commonly used in such applications. After constructing the task graph, load the job on a machine, and perform the set of tasks that can be done on this machine, such that they have no unfinished pre-requisites. When there is a choice of machine, pick the machine on which the largest set of jobs can be performed consecutively. Stop when all the tasks that are ready to be performed cannot be done on the current machine the job is on. Now move the job to a different machine (this incurs a loading time) and continue. Notice that, in general, the job could be loaded on the same machine many times.

1.2 Fork-Join Parallelism Problem

Let $G = (V, E)$ be a DAG representing the fork-join model. There is a non-negative cost function w, denoting the execution time associated with each vertex. Let W denote the ratio between the maximum and minimum costs of vertices in V. The cost of a set of vertices B, denoted by $w(B)$, is defined as the maximum cost of a vertex belonging to B.

The problem is defined as follows: Partition the vertices of the DAG into a set of blocks $B_1, B_2 \ldots B_k$ such that

- If $i \to j$ is an edge, and if $i \in B_{i'}$ and $j \in B_{j'}$ then $i' < j'$
- Minimize $\sum_{i=1}^{k} w(B_i)$

An *antichain* in a DAG is a set of incomparable elements, i.e., there is no directed path between any pair of elements in an antichain. In the context of the fork-join model, an antichain denotes a set of fork operations followed by a join operation. Essentially, we are asking for a partitioning of the DAG into a set of antichains (each block is an antichain), such that we minimize the sum of the costs of the antichains. We also require that for each edge $i \to j$, the block that i belongs to is "before" the block j belongs to (in other words, the antichains cannot "cross").

Notice that a greedy algorithm may perform very poorly for the fork-join problem, since it may spread high cost vertices between several blocks, instead of grouping them in the same one.

1.3 Our Results

It is easy to show that the Loading Time Scheduling Problem is NP-complete for arbitrary $M(i)$, even when there are no precedence constraints, by a reduction from the set-cover problem [6]. (The elements correspond to tasks, and each subset corresponds to a machine. A task can be done on a machine if the corresponding element belongs to the set corresponding to the machine.) When $|M(i)| = 1$, and $\rho \geq 4$, the problem can be shown to be NP-complete (and MAX SNP-hard) by a reduction from the shortest common supersequence problem [6, problem SR8]. Moreover, the reduction proves the hardness results even for the case of unit loading times. A recent result of [13] implies that LTSP is NP-complete for $\rho = 3$.

Hardness Results: Furthermore, we show that when ρ is a constant, then there exists an α such that no polynomial time approximation algorithm with a factor ρ^{α} is possible unless $P = NP$. We also show that for *any constant* δ, no polynomial time approximation algorithm with a factor of $\log^{\delta} n$ is possible unless $NP \subseteq DTIME(n^{O(\log\log n)})$. (This for the case when the number of machines is not restricted to a constant.)

Greedy Algorithm: First we show that the greedy algorithm performs poorly for LTSP. The greedy algorithm is a very natural algorithm for this problem: at each step it schedules the machine on which the maximum number of tasks can be performed. We give an example where the approximation ratio of the greedy algorithm is $\Omega(\sqrt{n})$, even when $\rho = 4$. When $\rho = \log n$, then the approximation ratio can be as bad as $\Omega(\frac{n}{\log n})$.

Approximation Algorithm: Our main contribution is an approximation algorithm to solve LTSP approximately. This algorithm achieves a worst case approximation of ρ, where ρ is the total number of machines. The idea is to compute for each task i a function which is a lower bound on the total loading time needed to schedule i. We sort the set of tasks according to this function, and then schedule the tasks in a greedy manner. The proof of the approximation factor for this algorithm is quite delicate. We also show that the algorithm can be implemented in $O(n \log n + \rho e)$ time and $O(\rho n + e)$ space, where n and e are the number of vertices and number of edges respectively in the graph.

From the practical point of view, an approximation factor of ρ is much better than an approximation factor that is a function of n, since ρ is typically very small (4 or 5), compared to the size of the task graph that can have, for example, over 1000 features for an engine block [7].

We also give a second approach that gives a simpler ρ approximation for LTSP. However, we believe that the previous method will produce better solutions in practice. In any case, both approaches are described as they use very different techniques.

Finally, we discuss a natural linear programming approach to LTSP and show that the integrality gap of the integer program derived is $(\rho-1)/4$, meaning that this approach is not useful for obtaining improved approximation factors.

Fork-Join Problem: Sarkar [15] presents a heuristic for solving the fork-join problem, with no analysis, and conjectures that his heuristic has a constant worst case guarantee. We have been able to construct an example for which Sarkar's algorithm has a performance of $\Omega(\log n)$ times the optimal cost, disproving his conjecture. However, no proof is known that this is also an upper bound for the algorithm's performance. It should be noted that Sarkar reports that the heuristic works very well in practice. We show that an instance of the fork-join problem can be mapped to an instance of LTSP. We use the same technique to obtain an algorithm with an approximation ratio of $O(\min(\log W, \log n))$ can be designed for this problem. This is the first worst case approximation algorithm for this problem.

Weighted SCS Problem: For the SCS problem where each letter of the alphabet has an arbitrary weight, we are able to obtain an algorithm with an approximation factor of ρ. Here ρ is the size of the alphabet.

2 Greedy Algorithm

The most obvious algorithm for the Loading Time Scheduling Problem is the greedy algorithm. This algorithm schedules at each step the machine on which the largest number of tasks can be performed on a single run. We show that this algorithm can perform very poorly, i.e., it may achieve an approximation factor of $\Omega(n/\log n)$.

We first show an example where the approximation factor is $\Omega(\sqrt{n})$. (See Fig. 2). The first row contains $\sqrt{n}$ tasks. All these tasks can be performed on machine 3. They can also be performed on machines 1 and 2 (alternately). The second row contains n tasks, all of which can be performed on machine 4, as well as on either 1 or 2 (depending on the machine their parent can be done on). Clearly, the maximum number of tasks that can be done on any single machine initially is $1+\sqrt{n}$ on machine 1, $\sqrt{n}$ on machine 3, and 0 on machines 2 and 4. The greedy algorithm will schedule machine 1. After performing $1+\sqrt{n}$ tasks, the greedy algorithm schedules machine 2 and performs another $1+\sqrt{n}$ tasks. Since there are $n+\sqrt{n}$ tasks in all, the greedy algorithm will obtain a solution of length $\sqrt{n}$ that alternates between the 1's and the 2's. The optimum solution is to schedule all the tasks on machines 3 and 4. First do $\sqrt{n}$ tasks on machine 3, then do the remaining n tasks on machine 4.

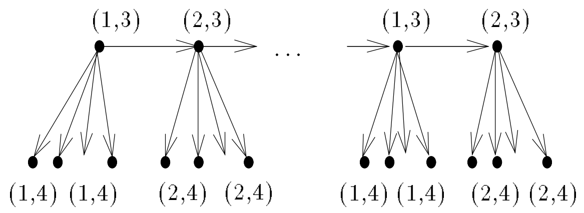

Figure 2: Bad example for the greedy algorithm.

We now show how to modify this instance so as to obtain an instance for which the performance ratio of the greedy algorithm can be as bad as $\Omega(n/\log n)$. The idea is to extend the instance depicted in Fig. 2 to an instance containing k (to be defined later) replicas of this instance, connected in "levels".

Our basic building block has the same structure as the instance depicted in Fig. 2. It has two layers, where each layer induces a chain. The first layer contains r (to be defined later) tasks, and each task in the first layer has 2 successors in the second layer. Let u and v be two tasks in the first layer: the sets induced by the successors of u and v in the second layer are disjoint.

We will now connect k instances of our basic building block in levels. Let the building blocks be denoted by $B_1, \ldots, B_k$, where the indices are ordered according to levels. For $1 \le i \le k$, in block B_i, the value of r is equal to 2^i. For $1 \le i < k$: there is a *single* outgoing edge from the jth task in the second layer of block B_i to the jth task in the first layer of block B_{i+1}, for $1 \le j \le 2^i$.

In block B_i, for $1 \le i \le k$:

1. All tasks in the first layer can be performed on machine ℓ_i, and all tasks in the second layer can be performed on machine ℓ_i'.

2. In the first layer: (i) all tasks such that their distance from the beginning of the layer is odd, and their successors in the second layer, can be performed on the same machine, denoted by o_i. (ii) all tasks such that their distance from the beginning of the layer is even, and their successors in the second layer, can be performed on the same machine, denoted by e_i.

This construction can be thought of as a tree, where all edges are directed towards the leaves. The optimal algorithm will traverse the tree in a BFS fashion. In other words, it will perform the tasks block-by-block, where the cost of each block is 2, since each layer has to be scheduled separately. Thus, the cost of an optimal algorithm is $2k$. In contrast, the greedy algorithm will traverse the tree in a DFS fashion. This follows since, at each step, there is a greedy choice that complies with a DFS traversal of the tree. The cost of the greedy algorithm is $\sum_{i=1}^{k} 2^i$, yielding that the performance ratio of the greedy algorithm is $\sum_{i=1}^{k} 2^i/(2k)$. The number of tasks that need to be processed is denoted by n. We choose $k = \Theta(\log n)$, yielding that the approximation factor of the greedy algorithm can be as bad as $\Omega(n/\log n)$.

3 Approximation Algorithms

The main idea behind the algorithm is to compute a function $T^*(i)$ that represents the lower bound on the total loading time incurred to schedule task i on *any* machine. For each task i, let $Pred(i)$ denote the set of predecessors of i in G.

We first compute the function $T(i,j)$ which is a lower bound on the time incurred if task i is scheduled on machine m_j. $T^*(i)$ is the time for scheduling task i as quickly as possible on any machine $m_j \in M(i)$. $T^*(i) = \min_{m_j \in M(i)} T(i,j)$ where $m^*(i) = m_j$, such that $T(i,j) = T^*(i)$.

We now define $T(i,j)$.

$$T(i,j) = \begin{cases} \infty & \text{if } m_j \notin M(i) \\ \ell(m_j) & \text{if } Pred(i) = \emptyset \\ \max_{i_p \in Pred(i)} \{\min(T(i_p, j), T^*(i_p) + \ell(m_j))\} & \text{otherwise} \end{cases}$$

For each task we create a vertical interval of length $\ell(m^*(i))$, with the lower end of the interval at distance $T^*(i)$ from the $x - axis$ (see Fig. 3). Two intervals are said to *overlap*, if there is a horizontal line that cuts both the intervals. The following two propositions are immediate.

Proposition 1 *To compute $T(i,j)$ we only need to consider the immediate predecessors of i.*

Proposition 2 *If $i_p \in Pred(i)$ then $T^*(i_p) \le T^*(i)$.*

We now give a high level description of the algorithm. Assume that S is the set of tasks that still needs to be scheduled (initially S is the entire set of tasks). The algorithm sweeps a horizontal line from top to bottom. When the sweep-line crosses the lower end of the vertical interval, we schedule the task. Assume that task x is the first task to be scheduled on $m^*(x)$. At this point we also schedule other tasks that can be done on $m^*(x)$ (R will denote the set of these tasks).

Algorithm:

Step 1. The set S is sorted by increasing $T^*(i)$ value (the lower end of the intervals). If $T^*(i) = T^*(j)$, and $i \in Pred(j)$, then i occurs before j in S.

Step 2. Pick the first task from S and call it x.

Step 3. Pick as many tasks from S as can be performed on $m^*(x)$. Formally, let $R = \{y | \, (m^*(x) \in$

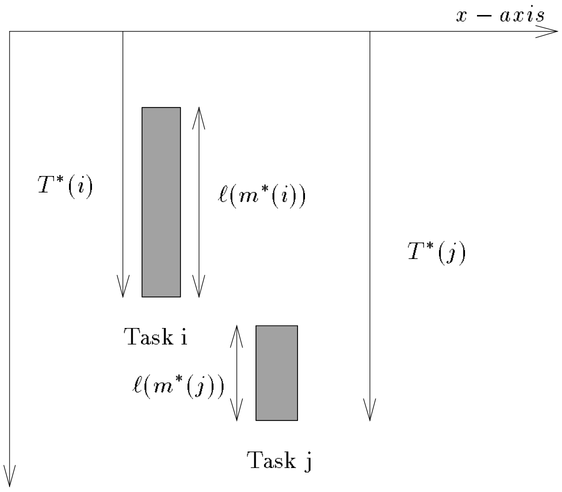

Figure 3: Example to show two non-overlapping intervals.

$M(y)) \wedge [(\forall z \in S \cap Pred(y)), \quad m^*(x) \in M(z)]\}$.

Step 4. Schedule R on $m^*(x)$.

Step 5. Remove R from S and return to Step 2.

Notice that for any two tasks i and i', if $i' \in Pred(i)$, then $T^*(i') \leq T^*(i)$, and i' precedes i in the set S. Hence, the solution produced by the above algorithm is feasible. Let OPT be the cost of the optimal solution (with minimum loading time to complete all the tasks).

Lemma 3 *For any i, $OPT \geq T^*(i)$.*

Proof: We show that $T(i,j)$ is a lower bound on the time elapsed if task i is scheduled on machine m_j. In other words, we show that in any feasible solution, in which task i is scheduled on machine m_j, the sum of the loading times of all the machines which are scheduled before, or, at the same time when task i is scheduled, is at least $T(i,j)$. Note that this would imply, for any i, OPT $\geq T^*(i)$.

The proof is by induction on the levels in the DAG. For tasks at level 0, (defined by $Pred(i) = \emptyset$), it is obviously true, since $T(i,j) = \ell(m_j)$ if $m_j \in M(i)$. Assume it is true for tasks in the first k levels. Consider now a task i at level $k+1$. Let $m_j \in M(i)$, and let i_p be a predecessor of i. In any feasible solution, i_p is scheduled before, or, at the same time as i. Assume that i_p is scheduled on machine m_{j_p}.

By the induction hypothesis, the sum of the loading times of all the machines which are scheduled before, or, with task i_p, is at least $T(i_p, j_p)$. There are two cases to be considered:

(a) $j = j_p$: then, at best, task i can be scheduled with task i_p, and hence the sum of the loading times of all the machines which are scheduled before, or, with i, is at least $T(i_p, j)$.

(b) $j \neq j_p$: in this case, i is scheduled strictly later than i_p, and hence the sum of the loading times of all the machines which are scheduled before, or, with i, is at least $T(i_p, j_p) + \ell(m_j) \geq T^*(i_p) + \ell(m_j)$.

The above implies that the sum of the loading times of all the machines which are scheduled before, or, with i, is at least $\max_{i_p \in Pred(i)}\{\min(T(i_p, j), T^*(i_p) + \ell(m_j))\} = T(i, j)$ □

Lemma 4 *Let x be a task picked in Step 2. Let $y \in S$ such that $m^*(x) = m^*(y)$. If the intervals of x and y overlap, then $y \in R$.*

Proof: Suppose that there exists a task $z \in S \cap Pred(y)$, such that $m^*(x) \notin M(z)$, and $T^*(x) \leq T^*(z) \leq T^*(y)$. This implies that $T(z, m^*(x)) = \infty$, and $T(z, m^*(y)) = \infty$. Since $z \in Pred(y)$, $T^*(y) = T(y, m^*(y)) = T(y, m^*(x)) \geq \min(T(z, m^*(y)), T^*(z) + \ell(m^*(y)))$. This proves that $T^*(y) \geq T^*(z) + \ell(m^*(y)) \geq T^*(x) + \ell(m^*(y))$, contradicting the assumption that the intervals of x and y overlap. □

Theorem 5 *The total loading time incurred for any particular machine m_j in a schedule output by the algorithm is at most* $\max_i T^*(i)$.

Proof: Consider any particular machine m_j. This machine may occur in the schedule many times. Each time the loading time is incurred for the first task (in particular, the task picked in Step 2). We "charge" this task for the loading time. By Lemma 4 the intervals for the tasks that get charged for loading machine m_j *do not overlap*. Hence, the total charge on the loading times for machine m_j is $T^*(L(j))$ where $L(j)$ is the last task to be charged for machine m_j. □
We conclude with the following theorem.

Theorem 6 *If the total number of machines is ρ, the total loading time is at most $\rho \cdot \max_i T^*(i) \leq \rho \cdot OPT$.*

3.1 Implementation

Theorem 7 *The algorithm can be implemented in $O(n \log n + \rho e)$ time and $O(\rho n + e)$ space, where n and e are the number of vertices and number of edges respectively in the DAG.*

Proof: Note that the computation of $T(i, j)$ and $T^*(i)$ can be done in $O(\rho e)$ time, by topologically sorting the DAG, and then doing a local computation at every task, according to the order in which the tasks appear in the topological sort. As mentioned before, the value of $T(i, j)$ can be computed by just looking at the immediate predecessors of task i, and therefore the local computation for task i takes $O(\rho \cdot indeg(i))$ time, where $indeg(i)$ is the in-degree of task i. For each task we need to store ρ values, so the total space requirement is $O(\rho n)$.

The rest of the algorithm is implemented as follows: For each task we maintain its in-degree in the current DAG (the tasks that are already scheduled are not in the current DAG). For each of the ρ machines, we store a doubly linked list of pointers to tasks in the current DAG with the following property. Let m and i be a machine and task respectively, with $m \in M(i)$. If the in-degree of i in the current DAG is 0, then the linked list for m contains task i. We also store back pointers (at most ρ) from i to facilitate deletion in constant time. The sorted list S of tasks is implemented as a doubly linked list with back pointers from the tasks in the DAG. Note that all this requires $O(\rho n)$ space.

Call the following one phase of the algorithm. Let x be the first task in the current set S. Using the linked list for $m^*(x)$ find set R. Remove all tasks in R from the DAG, update all the doubly linked lists and the in-degree of the tasks.

We will now show that a phase of the algorithm can be implemented in $O(\rho \sum_{r \in R} outdeg(r))$ time, where $outdeg(r)$ is the out-degree of task r in the original DAG. Since every node is in

R once, the time bound follows. We use an additional $Queue$ of size $O(n)$ to do this. R is also implemented as a linked list.

Put all the tasks in the list for machine $m^*(x)$ in the $Queue$. Repeatedly do the following until the $Queue$ is empty. Remove a task from the $Queue$, delete it from S and the linked lists for all the machines and add it to R. For all its immediate successors, if i is an immediate successor, decrease $indeg(i)$ by 1. If $indeg(i)$ becomes 0 then if

(a) $m^*(x) \in M(i)$, then add i to the Queue,

(b) $m^*(x) \notin M(i)$, then add i to the linked list for every machine m such that $m \in M(i)$.

When the $Queue$ is empty we use the back pointers to delete every task in R from S. Note that in a phase of the algorithm, every task i that is added to the $Queue$, ends up in R. Since i is added at most once to the $Queue$, and for i we do $O(\rho \cdot outdeg(i))$ work, the bound follows. □

3.2 Universal Sequences

A universal sequence is an infinite list of machines such that if the machines are scheduled according to the list, then any set of tasks can be performed on them, independent of the DAG. This concept was recently suggested by Yishay Mansour [12] in the context of randomized approximation algorithms which can be shown to achieve an expected approximation factor of $\rho - 1$. The difficulty with this approach is that it may generate a universal sequence of exponential length. Yossi Azar [1] suggested a way to make it deterministic, but this increased the approximation ratio by a factor of 2. Here we show how to get a ρ approximation in polynomial time deterministically. We describe a deterministic ρ approximation algorithm that also generates a universal sequence.

Recall that $\ell(m_j)$ is the loading time of machine m_j. Let set $S_j = \{k \cdot \ell(m_j) | k = 1, 2, \ldots\}$ be an infinite set. Define $S = \cup_{j=1}^{\rho} S_j$.

Sort the elements of set S in non-decreasing order and output the machines in that order (in case of a tie the machine with lower index comes first) This is an (infinite) universal sequence.

Example: Suppose we have 3 machines with loading times of 2, 3 and 7 respectively. $S_1 = \{2, 4, 6, 8, 10, 12, \ldots\}$, $S_2 = \{3, 6, 9, 12, \ldots\}$ and $S_3 = \{7, 14, \ldots\}$.

If we sort S in non-decreasing order, the sequence that we obtain is: $2, 3, 4, 6, 6, 7, 8, 9, 10, 12, 12, 14, 14 \ldots$ The corresponding sequence of machines is $m_1, m_2, m_1, m_1, m_2, m_3, m_1, m_2, m_1, m_1, m_2, m_1, m_3 \ldots$

To obtain the actual schedule of machines, we will scan the universal sequence (from left to right) and output a machine if it can do one or more ready jobs from the DAG. The completed jobs will then be deleted from the DAG. A potential problem with this scheme is that the portion of the universal sequence that we need to generate may be exponential in length. One way to address this is to keep track of the set of "ready" machines at each step by examining the DAG, i.e., machines that can do one or more ready jobs. We then need to output the next such machine from the universal sequence. This can easily be done very efficiently.

We now prove that the total loading time of the machines in the prefix of the universal sequence that was generated, does not exceed $\rho \cdot \mathrm{OPT}$. Let the loading time of a sequence of machines be the total loading time of all the machines in the sequence.

Theorem 8 *Any sequence of machines (over $m_1, m_2, \ldots m_\rho$) with total loading time l occurs as a subsequence of a prefix of the universal sequence of total loading time at most $\rho \cdot l$.*

Proof: First of all note that, in the construction of the universal sequence, if we restrict the largest element in set S to l (call this finite set S^l), then we get a prefix of the universal sequence of total loading time at most $\rho \cdot l$. This follows from the definition of universal sequence and the fact that, for any machine, the total loading time of all of its instances outputted for S^l, is upper bounded by l. Let U^l be the prefix of the universal sequence thus generated. Let $M = m_{\alpha_1}, m_{\alpha_2}, m_{\alpha_3} \ldots m_{\alpha_k}$ be a sequence of machines of total loading time $\sum_{j=1}^{k} \ell(m_{\alpha_j}) = l$. We now show that M is a subsequence of U^l thus proving the theorem. Let f be a function that assigns numbers to the k machines in the sequence M defined as:

(a) $f(m_{\alpha_1}) = \ell(m_{\alpha_1})$

(b) $f(m_{\alpha_i}) =$ smallest integer in $S_{\alpha_i} > f(m_{\alpha_{i-1}})$ if $\alpha_i \leq \alpha_{i-1}, i > 1$

(c) $f(m_{\alpha_i}) =$ smallest integer in $S_{\alpha_i} \geq f(m_{\alpha_{i-1}})$ if $\alpha_i > \alpha_{i-1}, i > 1$

Note that $f(m_{\alpha_i}) - f(m_{\alpha_{i-1}}) \leq \ell(m_{\alpha_i}), i > 1$ and $f(m_{\alpha_1}) = \ell(m_{\alpha_1})$. Therefore $f(m_{\alpha_k}) \leq \sum_{j=1}^{k} \ell(m_{\alpha_j}) = l$. Also note that the numbers assigned by f to the machines in the sequence M form a nondecreasing sequence with ties only when the earlier machine has lower index. Finally note that the number assigned to any machine m_i is from the set S_i. Therefore the sequence of numbers $f(m_{\alpha_1}), f(m_{\alpha_2}) \ldots f(m_{\alpha_k})$ forms a subsequence of the sorted elements of the set S^l and the sequence M forms a subsequence of U^l. □

3.3 Integrality gap

In this section we discuss a natural linear programming approach to the loading time scheduling problem. We prove that the integrality gap for the integer program that we will construct is $(\rho - 1)/4$, meaning that this approach cannot yield improved approximation factors. In fact, the integrality gap holds for the shortest common supersequence problem as well. Recall that the shortest common supersequence problem is a special case of the loading time scheduling problem by viewing alphabet letters as machines, and each sequence as defining precedence constraints between tasks. For example, a sequence $a_1, \ldots, a_k$ corresponds to tasks $t_1, \ldots, t_k$ such that: (i) tasks $t_1, \ldots, t_{i-1}$ must be executed before task t_i, for all $2 \leq i \leq k$; (ii) task t_i can only be executed on the machine corresponding to letter a_i, for all $1 \leq i \leq k$.

Our instance of the loading time scheduling problem is a DAG consisting of chains, where each task can be performed on a single machine. Suppose there are ρ machines, $m_1, \ldots, m_\rho$. The machines on which the tasks can be performed in each chain induce a permutation of $1, \ldots, \rho$, and there are $\rho!$ chains altogether.

A fractional schedule is defined as follows. At each time slot we allow a fraction of a machine to be scheduled with the following constraints: (i) the sum of the fractions of the machines scheduled together at a time slot cannot exceed 1, and (ii) a fraction α of a task can be performed in time slot t, only if an α fraction of all of its predecessors has already been completed up to time slot t', where $t' < t$. (We omit the details of defining a fractional solution for arbitrary instances of the loading time scheduling problem, since our goal here is proving a lower bound).

Let the length of our fractional schedule for the above instance be $2\rho - 1$. At each time slot, a fraction of $1/\rho$ of each machine is scheduled. It is easy to verify that all $\rho!$ chains of tasks can be performed using this schedule.

In contrast, we claim that the length (number of time slots) of any integral solution S is at least $\Omega(\rho^2)$. This is proved as follows. Scan S from left to right, and stop when all machines are recorded. Without loss of generality, suppose that m_ρ is the last machine recorded. Let j_ρ be the place in the list where m_ρ appears. Clearly, $j_\rho \geq \rho$. Resume the scanning of the list and stop again when all machines $m_1, \ldots, m_{\rho-1}$ are recorded. Without loss of generality, suppose that $m_{\rho-1}$ is the last machine recorded. Let $j_{\rho-1}$ be the place in the list where $m_{\rho-1}$ appears. Clearly, $j_{\rho-1} \geq 2\rho - 1$. Repeat the scanning, and let $j_{\rho-2}, ..., j_1$ be defined similarly. There are two cases. If the list S is exhausted before j_i is defined, (for some $i \geq 1$), then the chain of tasks that corresponds to the permutation $m_\rho, m_{\rho-1}, \ldots, m_1$ cannot be performed by S. Otherwise, the length of S is at least j_1, which is $\rho(\rho-1)/2$, yielding that the integrality gap is at least $(\rho-1)/4$.

4 Hardness Results

We prove the following two theorems regarding the Loading Time Scheduling Problem. These hold even for the restricted case, when each job can be done only on a single machine.

Theorem 9 *For any constant δ, there does not exist a polynomial time algorithm that has an approximation factor of $(\log n)^\delta$, unless $NP \subseteq DTIME(n^{O(\log\log n)})$.*

Theorem 10 *There does not exist a polynomial time ρ^α-approximation algorithm for some constant α, unless $P = NP$.*

The main idea is to take an instance X of a restricted version of the SCS problem, and to convert it into a "large" instance of the LTSP problem. This is done by modifying the construction in [9]. Using an approximation algorithm for the LTSP problem, we are able to obtain a c-approximation algorithm for the restricted SCS problem, for any c. We then use the fact that the restricted SCS problem is MAX SNP-hard, so a c-approximation algorithm would imply the existence of an algorithm to find the optimal solution with the same running time.

4.1 Preliminaries

Definition 1 *An LDAG is an acyclic digraph for which each vertex is labeled by a single letter (from a given alphabet).*

Definition 2 *A minimal supersequence z of an LDAG is defined as follows:*

- *If the LDAG is empty, then $z = \emptyset$*
- *Let a be the first letter of z, i.e., $z = a \cdot z_1$, then some indegree 0 node in the LDAG is labeled with a; and z_1 is a minimal supersequence of the LDAG obtained by deleting all indegree 0 nodes that have label a.*

Note that a supersequence of an LDAG is any sequence that contains a minimal supersequence of the LDAG as a subsequence.

Definition 3 *Let $X_1, X_2, \ldots, X_k$ be a collection of LDAG's. Let $X = X_1 \cdot X_2 \cdots X_k$ denote the LDAG that is obtained by connecting each X_i to X_{i+1} by a set of directed edges that go from each vertex of out-degree 0 in X_i, to each vertex of in-degree 0 in X_{i+1}.*

The following definitions from [9] are extended to LDAG's.

Definition 4 *Let Σ and Σ' be two alphabets. Let $a \in \Sigma$ and $b \in \Sigma'$ be two letters. The product $a \times b$ is the composite letter $(a, b) \in \Sigma \times \Sigma'$.*

Definition 5 *The product of an LDAG X and a letter b is the LDAG $(X \times b)$ obtained by taking the product of the label of each vertex of X with b. (The structure of the LDAG stays the same, only the labels change.)*

Definition 6 *The product of an LDAG X and a sequence $y = b_1 \dots b_k$ is the LDAG $(X \times y) = (X \times b_1) \cdot (X \times b_2) \cdots (X \times b_k)$.*

Definition 7 *The product of an LDAG X with a set $Y = \{y_1, \dots, y_n\}$ of sequences is denoted by $(X \times Y) = \cup_{i=1}^{n} (X \times y_i)$.*

Definition 8 *Let X be a set of sequences, where each sequence is also viewed as a chain. Then, $X^k = X^{k-1} \times X$, where $k > 1$.*

The size of X^k is n^k, where n is the size of X. The alphabet size of X^k is m^k, where m is the alphabet size of X.

Definition 9 *By $z[a \dots b]$, we denote the substring of z from position a to position b.*

Proposition 11 *Let $X = X_1 \cdot X_2 \cdots X_k$ where each X_i is an LDAG. Let z_i be a minimal supersequence for X_i. Then, $z = z_1 \cdot z_2 \cdot \ldots \cdot z_k$ is a minimal supersequence for X.*

Proposition 12 *Let $X = X_1 \cdot X_2 \cdots X_k$ where each X_i is an LDAG. Let z be a minimal supersequence for X. We can write z as $z_1 \cdot z'$, where z_1 is a minimal supersequence for X_1 and z' is a minimal supersequence for $X_2 \cdots X_k$.*

Observe that we can obtain the unique decomposition of z in time $O(|z|)$, by scanning z. Let X be an LDAG and z be a supersequence of X. Let the length of the smallest prefix z' of z which is a supersequence of X be denoted by the function $g(X, z) = |z'|$. The prefix z' can be computed in polynomial time by scanning z and X. Let z'' be a subsequence of z' such that z'' is also a minimal supersequence of X. Let the function $h(X, z) = z''$. Note that z'' is unique for a fixed (X, z) pair and can be computed and extracted from z in polynomial time.

4.2 Main Lemmas

Lemma 13 *Let X be a set of sequences, and z a supersequence of X^k. We can find k minimal supersequences $z_1, z_2 \dots z_k$ of X such that the product of the length of these sequences is at most the length of z, i.e., $|z_1| \cdot |z_2| \cdots |z_k| \le |z|$. Hence we can find a supersequence of X of size at most $|z|^{1/k}$. This can be done in time which is polynomial in $|X^k|$.*

Proof: The proof is by induction on k. The lemma is clearly true for $k = 1$. We prove the induction step separately, in Lemma 14. □

Consider X^k, where X is a set of sequences and consider a sequence $a_1.a_2.\ldots.a_\ell$ in X. Corresponding to this sequence, there is an LDAG in X^k of the form $(X^{k-1} \times a_1) \cdot (X^{k-1} \times a_2) \cdots (X^{k-1} \times a_\ell)$.

Lemma 14 *Let X be a set of sequences and let z be a supersequence for X^k. In time polynomial in $|X^k|$ we can find z' such that $|z'| = |z|$ and a decomposition $z' = (z_1 \times x_1) \cdot (z_2 \times x_2) \cdots (z_r \times x_r)$, where each z_i is a minimal supersequence of X^{k-1}, x_i is a letter of the alphabet of X, and $x = x_1 \cdot x_2 \cdots x_r$ is a minimal supersequence of X. This implies that for some j, $|z_j| \cdot |x| \le |z'| = |z|$.*

Proof: We give a constructive proof. Let $S = \{x_i | x_i$ appears as the first letter in some sequence of $X\}$ and let x_p be a letter in S for which $\min_{x_i \in S} g((X^{k-1} \times x_i), z)$ is attained. Delete $z'' = h((X^{k-1} \times x_p), z)$ from z, yielding sequence z'. Output z'', which is a minimal supersequence for $(X^{k-1} \times x_p)$. Replace X^k by $X^{k-1} \times X_1$ where X_1 is obtained by replacing every sequence of the form $x_p \cdot Y$ (in X) with Y.

Note that $|z'| + |z''| = |z|$. We claim that $z'' \cdot z'$ is a supersequence for X^k for the following reasons.

- For an LDAG in X^k of the form $R = (X^{k-1} \times x_p) \cdot T$ we have by the previous propositions that $h(R, z) = z'' \cdot t$, where t is a minimal supersequence of T and is a subsequence of $z[g((X^{k-1} \times x_p), z) + 1 \ldots |z|]$. Hence t is a subsequence of z'. Thus $z'' \cdot z'$ contains a minimal supersequence of R.

- For any LDAG in X^k of the form $R = (X^{k-1} \times x_i) \cdot T$, where $x_i \neq x_p$ we have by the propositions and by the fact $g((X^{k-1} \times x_i), z) \geq g((X^{k-1} \times x_p), z)$ that $h(R, z) = h(R, z'' \cdot z')$. Thus $z'' \cdot z'$ contains a minimal supersequence of R

Therefore z' is a supersequence for the modified X^k. We iteratively apply this procedure to z' with the modified X^k given by $X^{k-1} \times X_1$ until X_1 is empty. It is easy to see that if this algorithm outputs $L_1, L_2, \ldots, L_r$ sequences in that order then each L_i is a minimal supersequence of $(X^{k-1} \times x_i)$ for some x_i letter in the alphabet of X and $x_1 \cdot x_2 \cdot \ldots \cdot x_r$ is a minimal supersequence of X. The running time of the algorithm is polynomial in $|X^k|$. □

Lemma 15 *Given a supersequence z of X^k, we can compute a supersequence z' of X of size $|z'| = |z|^{1/k}$ in time polynomial in $|X^k|$. Hence, given a polynomial time approximation algorithm that achieves an approximation factor of $f(N)$ (where N is the input size) whose instance is X^k, we can construct an $f^{1/k}(N)$-approximation algorithm for the problem whose instance is X that runs in time polynomial in $|X^k|$.*

Proof: Let OPT_k be the size of the optimal solution for the problem whose instance is X^k. The optimal solution for the problem whose instance is X is exactly $OPT = OPT_k^{1/k}$. This follows from the previous proposition and the fact that if Y is a supersequence for X then Y^k is a supersequence for X^k. Assume there exists a polynomial time $f(N)$-approximation algorithm for the LTSP problem. This means that in polynomial time we can find a supersequence for the instance of X^k, which is of size at most $f(N) \cdot OPT_k$. But this implies the size of the solution for the problem whose instance is X is at most $(f(N) \cdot OPT_k)^{1/k} = f^{1/k}(N) \cdot OPT$. □

4.3 Proofs of Theorems

In the following proofs we use a restricted version of the SCS problem, where there are no consecutive run of the same alphabet in the sequences. Note that an instance of this restricted version of the SCS problem is also an instance of LTSP, with unit loading time. This problem is MAX SNP-hard (see Section 6), for alphabet size four or more. This implies that there is a constant c, such that if there exists a polynomial time c-approximation algorithm for the restricted SCS problem, then $P = NP$. In other words, there is a constant c, such that if there exists a $DTIME(n^{O(\log\log n)})$ time c-approximation algorithm for the restricted SCS problem, then $NP \subseteq DTIME(n^{O(\log\log n)})$. Let X be an instance of size n, of the restricted SCS problem, over an alphabet of size four.

We first prove Theorem 9.

Proof: Let c be the constant such that for the problem with instance X, there does not exist a $DTIME(n^{O(\log\log n)})$ time c-approximation algorithm, unless $NP \subseteq DTIME(n^{O(\log\log n)})$. We will show that if there is a polynomial time $(\log n)^\delta$ approximation algorithm for LTSP (for any δ), then we can construct a c-approximation algorithm for the restricted SCS problem, that runs in $DTIME(n^{O(\log\log n)})$ time. This would imply that $NP \subseteq DTIME(n^{O(\log\log n)})$.

Suppose we are given a polynomial time approximation algorithm that achieves an approximation factor of $f(N)$ (where N is the input size), for LTSP. Since X and hence X^k is an instance of LTSP, by Lemma 15, applying this algorithm to instance X^k, would imply an approximation factor of $f^{1/k}(n^k)$ for instance X. We would like to choose k such that $f^{1/k}(n^k) \leq c$. Hence, we require $f(n^k) < c^k$. Let $f(n) = \log^\delta n$, and thus $(k \log n)^\delta < c^k$. We now pick

$$k = 2 \log_c \log^\delta n + 2\delta \log_c \delta$$

and this yields a c-approximation algorithm for X that runs in time $O(\text{poly}(N))$, where $N = n^k$ is $O(n^{O(\log\log n)})$. □

We now prove Theorem 10.

Proof: The idea is the same as the previous proof. The only difference is that k will be chosen to be a constant. Let c be the constant such that there does not exist a polynomial time c-approximation algorithm for instance X unless $P = NP$. Let us choose $k = \log_4 \rho$. Since the alphabet size of X four, the problem whose instance is X^k has alphabet size $4^{\log_4 \rho} = \rho$. Let $\alpha = \log_4 c$. If there exists a polynomial time ρ^α-approximation algorithm for LTSP, for constant ρ machines, then there exists a polynomial time $\rho^{\alpha/k} = 4^\alpha$-approximation algorithm for the problem whose instance is X. But, $4^\alpha = c$, so we get a contradiction. Finally, note that $c < 3$, since there exists a polynomial time 3-approximation algorithm. □

5 Fork-Join Problems

Given an instance of the Fork-Join problem, we show how to create an instance of the LTSP such that the ratio of the cost of their optimal solutions is a constant. If W is the highest execution time instruction, (assuming that the lowest execution time is 1), in the Fork-Join problem, then for the instance of the LTSP, $\rho = \log W$. We then use our approximation algorithm to solve the instance of LTSP within a factor $\log W$. We finally show that any solution for the LTSP instance can be mapped back to a solution for the Fork-Join problem instance without any cost increase. This yields an algorithm with approximation ratio $O(\log W)$ for the Fork-Join problem. We would also like to mention that a slight modification of this technique yields an algorithm with approximation ratio $O(\log n)$ where n is the number of nodes.

5.1 Mapping a Fork-Join instance to an LTSP instance

The instance of the Fork-Join problem is given as a DAG (V, U). V is the set of nodes and U is the set of edges. Every node is labeled with an execution cost. Assume all the execution costs in the Fork-Join problem are between 1 and W. This can be achieved by a proper scaling. Increase any cost that lies in the range $(2^i, 2^{i+1}]$ to 2^{i+1} . Note that now we have $\leq \lceil \log W \rceil$ distinct costs and the cost of any solution of the original problem gets increased by at most a factor of 2.

We now create a new DAG (V', U') by introducing $|U|$ new nodes, each with execution cost 0 and by replacing the i^{th} edge (x, y) by two edges (x, r_i) and (r_i, y), where r_i is the i^{th} new node. Note that this changes neither the set of feasible solutions nor the cost associated with them.

We map this new instance of the Fork-Join problem to an instance of the LTSP as follows. The underlying DAG for the LTSP is the same, i.e., V' is mapped to the set of tasks and U' is the set of edges. For every distinct execution cost w_i we introduce a machine m_i with loading cost w_i, thus $w_0 = 0$ and $w_j = 2^j, j \geq 1$ and $\rho \leq \lceil \log W \rceil + 1$. For every node $j \in V'$, $M(j) = \{m_i | w_i \geq$ execution cost of node j$\}$.

Lemma 16 *If OPT_{LTSP} is the size of the optimal solution for the instance of LTSP and OPT_{FJ} the size of the optimal solution for the original instance of the Fork-Join problem then $OPT_{LTSP} \leq 4 \cdot OPT_{FJ}$.*

Proof: Let OPT'_{FJ} be the size of the optimal solution for the instance of the Fork-Join problem created by our mapping scheme. We have already argued $OPT'_{FJ} \leq 2 \times OPT_{FJ}$. Also note that any parallel execution of instructions with execution costs in the set $(2^{k_1}, 2^{k_2} \ldots 2^{k_l})$ where $(k_1 > k_2 \ldots > k_l)$ can be replaced by exactly one execution of each of the machines in the set $(m_{k_1}, m_{k_2} \ldots m_{k_l})$ in any order, with at most a doubling of the cost. This is because $2^{k_1} + 2^{k_2} \ldots + 2^{k_l} \leq 2 \cdot 2^{k_1}$. Therefore $OPT_{LTSP} \leq 2 \cdot OPT'_{FJ} \leq 4 \cdot OPT_{FJ}$ □

Proposition 17 *Any feasible solution for the LTSP is a feasible solution for the instance of the Fork-Join problem, obtained by mapping tasks to instructions.*

Theorem 18 *An algorithm with an approximation ratio of $O(min(\log W, \log n))$ can be designed for the problem of generating code for a parallel machine.*

5.2 Sarkar's algorithm

Let $pred(v)$ be the cost of the heaviest weight path from some node of indegree 0 to v, and let $succ(v)$ be the cost of the most expensive path from v to some node of outdegree 0. Note that they do not include the cost of v itself. Sarkar calls $pred(v)$ the Earliest Starting Time and $DAGCP - succ(v)$ the Latest Completion Time of node v, where $DAGCP$ is the weight of the heaviest weight path in the DAG.

The algorithm works in phases. Each phase has 5 steps:

(1) Set k to the node with largest execution cost. The set $CurBlock$ is initialized to k.

(2) Compute $pred(v)$ and $succ(v)$ for every node v. Let $T_{PRED} = pred(k)$, $T_{SUCC} = succ(k)$.

(3) Compute $ParallelSet$, the set of nodes that can execute concurrently with k.

(4) Repeatedly do:
choose a NODE j from $ParallelSet$ so as to minimize

$$\max(T_{PRED}, pred(j)) + \max(T_{SUCC}, succ(j)).$$

Put j in $CurBlock$. $ParallelSet$ is updated to contain nodes which can be done in parallel with nodes in $CurBlock$ and $T_{PRED} = \max(T_{PRED}, pred(j)), T_{SUCC} = \max(T_{SUCC}, succ(j))$

(5) If $ParallelSet = \emptyset$ then collapse all nodes in $CurBlock$ to node k, i.e., all the nodes in $CurBlock$ will be done with node k in the final schedule. Go back to step (1) with the reduced DAG, and we mark k to be a vertex that cannot be picked by step (1).

5.3 Bad example for Sarkar's algorithm

Let $G = (V, E)$ be a DAG representing the fork-join model, where there is a non-negative cost function w, denoting the execution time, associated with each vertex. We will denote each vertex in V by a rectangle whose vertical length is proportional to its cost. Consider an instance of the fork-join problem where the DAG is a collection of $\log n$ chains each of unit total cost. Vertices in each chain have the same cost. The first chain has one vertex, the second chain has two vertices, the third chain has four vertices, the fifth chain has eight vertices etc. The last chain has $n/2$ vertices. We claim that any solution to this fork-join instance has cost at least $\frac{1}{2}\log n$. Consider any solution for this instance, it is a set of blocks $B_1, B_2 \ldots B_k$ as defined before. Note that there must be one block of cost 1, one other block of cost at least 1/2 (because there are two vertices of cost 1/2 which cannot be in the same block), two other blocks of cost 1/4 (because there are four vertices of cost 1/4 and only two of them can be in the previous blocks), four other blocks of cost 1/8 and so on. So the total cost of the blocks must be at least $1 + 1/2 + 2 \cdot 1/4 + 4 \cdot 1/8 \ldots \frac{n}{4}\frac{2}{n} = 1/2 + \frac{1}{2}\log n$

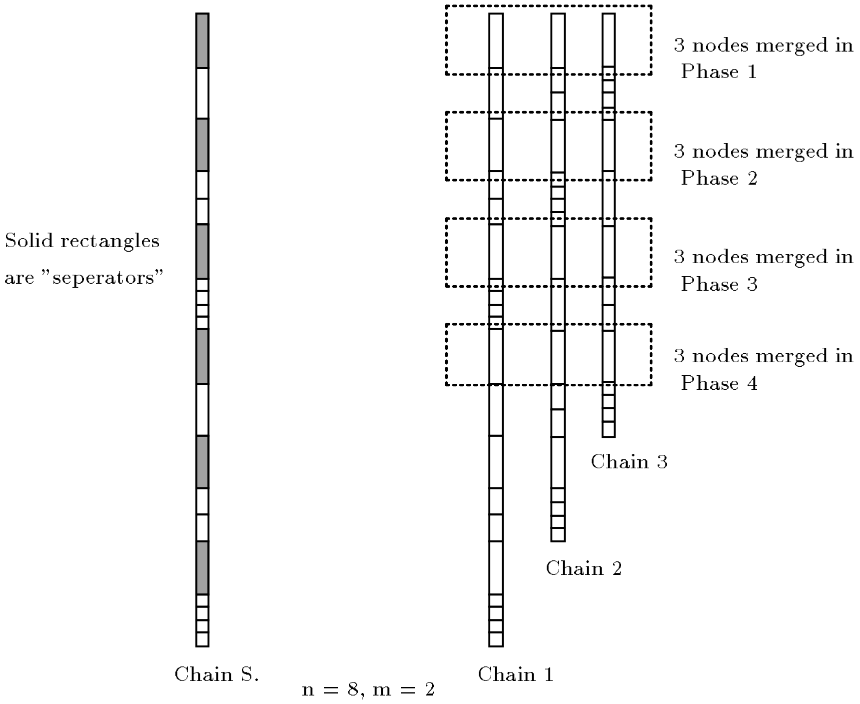

Figure 4: Bad example for Sarkars algorithm

We now construct a bad example for Sarkar's algorithm. Consider chain X constructed by concatenating the chains of the previous example in order. That is, chain $i+1$ is placed directly after chain i. Between chain i and $i+1$ we introduce a vertex of unit cost, and a new vertex of unit cost is placed on top of this chain. We call these newly introduced unit cost vertices "separator" vertices. Now concatenate m copies of X to obtain a new chain S. From S we create $\log n$ chains, where the i^{th} chain is obtained by removing from S all the vertices strictly before the i^{th} (from top) "separator" vertex. An example for $n = 8, m = 2$ is shown in Fig. 4. Note that the edges in the chain go from top to bottom.

Sarkar's algorithm when applied to our example does the following. In the i^{th} phase it merges the i^{th} (from top) "separator" vertices of the chains, for the first $\Theta(m \log n)$ phases. This is because each "separator" vertex has unit cost (the largest cost for any vertex), and the merging leads to no increase in T_{PRED} or T_{SUCC} if the first vertex to be picked (vertex k in the algorithm) for the phase is the "separator" vertex in the longest chain. Note also that the vertices between the i^{th} and $(i+1)^{th}$ merged "separator" vertices are an instance of the fork-join problem, mentioned at the beginning of the section for which the cost of any solution is at least $\frac{1}{2}\log n$. So the solution produced by Sarkar's algorithm would have cost at least $\Theta(m \log^2 n)$, but the optimal solution is just the length of the longest chain, which is $2m \log n$. Therefore (for large m) Sarkar's algorithm gives a $\Omega(\log n)$ approximation.

6 NP-completeness Proof

We prove that the Loading Time Scheduling Problem is NP-complete even for the case of a constant number of machines, and $M(i) = 1$, by a polynomial time reduction from the *Shortest Common Supersequence* problem [6, problem SR8]).

Shortest Common Supersequence: Given a finite alphabet Σ, finite set R of sequences from Σ^*, and a positive integer K. Is there a string $X \in \Sigma^*$ with $|X| \leq K$ such that each sequence $S^i \in R$ is a subsequence of X, i.e., $X = x_0 s_1^i x_1 s_2^i \ldots s_p^i x_p$ where each $x_j \in \Sigma^*$ and $S^i = s_1^i s_2^i \ldots s_p^i$? This problem is known to be NP-complete even when $|\Sigma| = 5$ [11] as well as when $|\Sigma| = 2$ [14]. Recently, it has been show to be MAX SNP-hard over a binary alphabet as well [2].

Theorem 19 *LTSP is NP-complete and MAX SNP-hard for $\rho \geq 4$.*

Proof: It is easy to see that the problem is in NP since we can verify a given partitioning of V easily. We will prove that it is NP-hard by a reduction from the Shortest Common Supersequence problem (SCS). Assume that R contains sequences $S^1, \ldots, S^\ell$, and that $\sum_{i=1}^{\ell} |S^i| = n$.

We will prove the problem NP-complete and MAX SNP-hard for the case when each task can be done only on a single machine, i.e., $|M(i)| = 1$, and $\rho = 4$. A sequence $x_1 x_2 \ldots$ simply denotes a set of tasks, such that task i can be done on machine x_i, and that x_i needs to be done before x_{i+1}. The set of machines is $\Sigma \cup \Sigma'$, where $\Sigma' = \{a' | a \in \Sigma\}$. We are assuming that for every $a \in \Sigma, a'$ is a new letter not already in Σ. So the number of machines is twice the alphabet size. The loading time for all the machines is the same (unit loading time).

For the reduction, we create a set R' of chains $C^1, \ldots, C^\ell$.
$R' = \{s_1 s_1' s_2 s_2' s_3 s_3' \ldots s_p s_p' | s_1 \ldots s_p \in R\}$. In other words, we replace every letter s_j by $s_j s_j'$.

We claim that the optimal schedule for the LTSP has loading time $2K$ if and only if the shortest supersequence of the SCS is of length K.

We first prove that if the shortest supersequence is of size K then there is a schedule with loading time $2K$. Let the shortest supersequence be $X = X_1 X_2 \ldots X_K$, where $X_i \in \Sigma$. We leave it to the reader to verify that $X_1 X_1' \ldots X_K X_K'$ is a valid schedule for all the chains.

We now prove that if the shortest supersequence is of length K, then any schedule must have length $\geq 2K$. Let the schedule have length L. We can view this schedule as a sequence X from the alphabet $\Sigma \cup \Sigma'$ of length L. From this sequence we obtain 2 sequences X' and X''. X' is obtained from X by removing all the letters from Σ'. X'' is obtained from X by removing all the letters from Σ, and then substituting every letter by its corresponding letter in Σ (i.e., replace s_j' by s_j). Note that both X' and X'' are supersequences for the SCS. Therefore $|X'| \geq K$ and $|X''| \geq K$. Thus, $L = |X'| + |X''| \geq 2K$. □

The following theorem is obvious.

Theorem 20 *The Loading Time Scheduling Problem can be solved in polynomial time for $\rho = 2$.*

A recent result of [13] implies that LTSP is NP-complete for $\rho = 3$.

7 Weighted Shortest Common Supersequence problem

As an application of our result we give a ρ approximation algorithm for the WSCS problem. WSCS is a generalization of the SCS problem where the letters in the alphabet have weights associated with them and we want to compute a common supersequence with minimum weight. Note that, if with every letter of the alphabet, we associate a machine, with loading time equal to the weight of the letter, then the sequence of machines that correspond to the optimal solution to the WSCS instance, would occur as a subsequence of a prefix of the universal sequence of total loading time at most $\rho \cdot \mathrm{OPT}$. Here OPT is the weight of the optimal solution of the WSCS instance. This follows from the fact that Theorem 8 holds even for those schedules where consecutive schedules of the same machine are allowed. So the algorithm presented in Subsection 3.2 gives a ρ approximation for the WSCS problem.

Acknowledgements

We are extremely grateful to Satyandra Gupta for telling us about this scheduling problem. We would like to thank Yossi Azar and Yishay Mansour for letting us include their suggestions regarding universal sequences (Subsection 3.2). We thank Amos Fiat and Uzi Vishkin for helpful discussions. We thank Tao Jiang and Ming Li for pointers to references [2, 14].

References

[1] Y. Azar. personal communication (1995).

[2] P. Bonizzoni, M. Duella and G. Mauri. *Approximation complexity of longest common subsequence and shortest common supersequence over fixed alphabet.* Technical Report 117/94, Universita degli Studi di Milano, (1994).

[3] D. Das, S. Gupta and D. Nau. *Reducing setup cost by automated generation of redesign suggestions.* Proc. ASME Computers in Engineering Conference, pages 159–170, (1994).

[4] J. Ferannte, K. Ottenstein and J. Warren. *The program dependence graph and its uses in optimization.* ACM Transactions of Programming Languages and Systems, pages 319–349, (1987).

[5] D. Foulser, M. Li and Q. Yang. *Theory and algorithms for plan merging.* Artificial Intelligence Vol 57:143–181, (1992).

[6] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*, Freeman, San Francisco, (1979).

[7] S. Gupta. personal communication.

[8] C. C. Hayes. *A model of planning for plan efficiency: Taking advantage of operator overlap.* Proc. of the 11th International Joint Conference of Artificial Intelligence, pages 949–953 (1989).

[9] T. Jiang and M. Li. *On the approximation of shortest common supersequences and longest common subsequences.* Proc. of 21st International Colloq. on Automata Langs. and Programming, pages 191–202, (1994).

[10] E. Lawler, J. Lenstra, A. Rinnooy-Kan, D. Shmoys. *Sequencing and scheduling: algorithms and complexity.* Handbooks in Operations Research and Management Science, Vol 4: Logistics of Production and Inventory, (Eds: S. C. Graves, A. H. G. Rinnooy Kan and P. Zipkin).

[11] D. Maier. *The complexity of some problems on subsequences and supersequences.* Journal of the ACM, Vol 25:322–336, (1978).

[12] Y. Mansour. personal communication (1995).

[13] M. Middendorf. *Supersequences, Runs, and CD Grammar Systems.* Developments in Theoretical Computer Science, 101-114, Topics in Computer Science, Vol. 6, (Eds: J. Dassow, A. Kelemenova).

[14] K. J. Raiha and E. Ukkonen. *The shortest common supersequence problem over a binary alphabet is NP-complete.* Theoretical Computer Science Vol 16(2):187–198, (1981).

[15] V. Sarkar. *Instruction reordering for fork-join parallelism.* Proc. of ACM SIGPLAN-PLDI Conf., pages 322–336 (1990).

[16] T. Sellis. *Multiple-query optimization.* Transactions on Database Systems, Vol 13(1): 23–52, (1988).