Scheduling For Distributed Computing *

Sartaj Sahni

Computer Info. Sci. and Engr.
University of Florida
Gainesville, FL 32611
sahni@cise.ufl.edu

George Vairaktarakis

Management Department
Marquette University
Milwaukee, WI 53233
vairaktaraki@vms.csd.mu.edu

Abstract

A typical model for distributed computing is to have a main program thread that runs on one processor. This thread spawns a number of tasks from time-to-time. When tasks are spawned, they are sent to other processors for completion and the main thread waits till the results of all tasks are received from the remote processors. This is the typical fork-join paradigm. This paradigm results in an interesting scheduling problem that is studied in this paper. Several heuristics are proposed for various variants of this scheduling problem.

1 Introduction

A popular paradigm for distributed computing involves a main computational thread that runs on a master processor; this thread forks several processes that can be run on other processors; the main thread continues once all the processes spawned by the fork complete. Many programs may be written so that the number of processes spawned by the fork is determined dynamically; i.e., at the time of the fork, the program determines the number of available processors and spawns at most this many processes. To execute the processes spawned at the fork, program and data need to be transmitted to the remote processors (also called slave processors) that will execute each. In turn, the results are transmitted back from the remote processors to the master. The master must spend time creating and receiving the transmission packets to/from each slave. In addition, the actual transmission times are significant and need to be accounted for.

The fork operation therefore, leads to the following scheduling problem. A master processor must create and receive the results of p processes. Each process has a preprocessing time (this includes the time the master must spend to create and initiate the transmission of the data and program packets), an execution time (this includes the time a slave processor must spend executing the process, receiving the program and data, creating and initiating the transmission of the result), and a postprocessing time (the time the master must spend receiving the result packets from the completed process). When we have just one master, the transmission times may be added into the slave execution time. The master processor needs to determine an order in which the preprocessing and postprocessing tasks are done. Different orders result in different completion time for the fork operation.

It is easy to see that the above model also applies to a parallel computing environment in which we have a multiprocessor resource (the slaves) that is attached to a host computer (the master) and the primary program thread runs on the master. For this application, we may generalize the model to the case of multiple master processors that share the same set of slave processors. The total number of slave processes generated by the threads running on all masters does not exceed the number of slave processors.

The master-slave scheduling model described above may also be used to model industrial applications. For example, consider the case of consolidators that receive orders to manufacture quantities of various items. The actual manufacturing is done by a collection of slave agencies. The consolidator needs to assemble the raw material (from his/her inventory) needed for each task, load the trucks that will deliver this material to the slave processors, and perform an inspection before the consignment leaves. All of these are part of the task preprocessing done by the master processor (i.e., the consolidator). The slave processors need to wait for the arrival of the raw material, inspect the received goods, perform the manufacture,

*This work was supported in part by the National Science Foundation under grant MIP-9103379 and the Army Research Office under grant DAA H04-95-1-0111.

load the goods on to the trucks for delivery, perform an inspection as the trucks are leaving. These activities together with the delay involved in getting the trucks to their destination (i.e., the consolidator) represent the slave work. When the finished goods arrive at the consolidator, they are inspected and inventoried. This represents the postprocessing.

The consolidator example may be generalized to include several consolidators. Now, the resulting scheduling problem may be modeled as a restricted multiple master system. On the other hand if there is a single consolidator with multiple trucks and each truck has its own crew for loading, inspecting, etc., then the scheduling problem can be modeled as a multiple master system (each truck and crew define one master) in which the master that pre-processes job i (i.e., the truck that delivers the raw material for the job) need not be the same as the one that post-processes job i (i.e., the truck that brings back the finished goods corresponding to this job).

While the problem of scheduling multiprocessor computer systems has received considerable attention [3], [4], [10], [12], [14], [15], [18], [22], it appears that the master-slave model has not been studied prior to the work of Sahni [19]. It is interesting to note that the master-slave scheduling model may be regarded as a variant of the job shop (see [1], [2] for a definition of a job shop as well as for elementary terminology concerning scheduling) as described below:

1. the job shop has two classes of machines: master and slave
2. there is exactly one master machine and the number of slave machines equals the number of jobs
3. each job has three tasks to be done in order; the first and third on the master and the second on a slave

The two machine flowshop model with transfer lags (2FTL) is a close relative to the master-slave model. In this model the preprocessing task has to be processed by the upstream machine, followed by a waiting period known as transfer lag, followed by the postprocessing task at the downstream machine. Special cases of this model are among the first problems considered in scheduling theory; see [8], [16], [21]. In [7], the problem of finding minimum makespan schedules for 2FTL was shown to be strongly NP-hard. Further results on 2FTL may be found in [5]. The problem of scheduling single machines with time lags and two tasks per job is identical to the single-master master-slave model. Since the former problem is strongly NP-hard [9], the single master problem is also strongly NP-hard.

In [19], the problem of finding minimum makespan no-wait-in-process schedules is shown to be NP-hard for the case of a single master. This remains true even when the pre- and post-processing tasks are required to be done in the same order. When the order in which the post-processing tasks is done is required to be reverse of the pre-processing order, the minimum makespan schedule can be found in $O(n \log n)$ time. Fast polynomial time algorithms to obtain minimum makespan schedules in which the pre- and post-processing orders are the same (or reverse) and a job may wait between the completion of one task and the start of the next are also developed in [19].

For no-wait scheduling, the single-master master-slave model and the coupled-task model of [17] are identical. Orman and Potts [17] show that many versions of this latter problem are strongly NP-hard. These results carry over to the no-wait master-slave model.

The outline of the rest of this paper is as follows. In Section 2 we define the problems to be considered and present some basic results. In Section 3, we develop fast approximate algorithms for problems on a single master processor. In Section 4, we consider the problem of obtaining minimum finish time schedules for multiple master systems. We conclude with future research directions in Section 5.

2 Notation and Basic Results

A set of jobs is to be processed by a system of master and slave processors. Each job has three tasks associated with it. The first is a preprocessing task, the second is a slave task, and the third a postprocessing task. The tasks of each job are to be performed in the order: preprocessing, slave, postprocessing. Let a_i, b_i, and c_i, respectively, denote the preprocessing, slave, and postprocessing tasks (and task times) of job i. All task times are assumed to be greater than zero (i.e., $a_i > 0$, $b_i > 0$, and $c_i > 0$, for all i). The available processors are divided into two categories: master and slave. If n denotes the number of jobs, then no schedule can use more than n slaves. Hence we may assume that there are exactly n slaves. The *makespan* or *finish time* of a schedule is the earliest time at which all tasks have been completed.

Figure 1 (a) shows a possible schedule for the case when $n = 2$, $(a_1, b_1, c_1) = (2, 6, 1)$, and $(a_2, b_2, c_2) = (1, 2, 3)$. In this schedule, the preprocessing of job 1 is handled first by the master; all other tasks begin at the earliest possible time. M denotes the master processor and S_1 and S_2 denote the slaves. The finish time is 9. The schedule that results when the master pre-processes job 2 first and all other tasks begin at

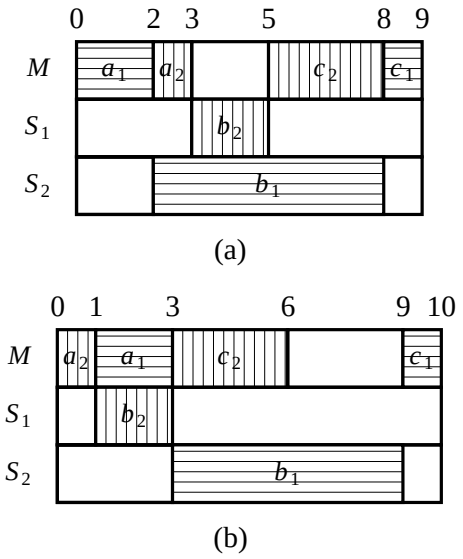

Figure 1: Example schedules

the earliest possible time is shown in Figure 1 (b). This has a finish time of 10.

Let us examine the schedules of Figure 1. Notice that in both schedules, once the processing of a job begins, the job is processed continuously until completion. Schedules with this property are said to have *no-wait-in-process*. In industrial applications, one may impose this requirement on a schedule. Another interesting feature of the schedules of Figure 1 is that in one the postprocessing is done in the reverse order of the preprocessing while in the other the pre- and post-processing orders are the same. In some settings, we may require that schedules satisfy one order or the other. For example, this could simplify the postprocessing if a stack is used, by the master, to maintain a record of jobs in process. Similarly, if the master uses a queue to maintain this information, we might require that the postprocessing be done in the same relative order as the preprocessing. Another discipline that might be imposed on the master is to complete all the preprocessing tasks before beginning the first postprocessing task. Both of the schedules of Figure 1 obey this discipline.

Similar requirements may be imposed in our consolidator example. This time suppose that all the raw material is loaded on a single truck and that the slaves are uniformly spaced. Whenever the truck stops, it has to wait at the slave location while the material for that location is unloaded and checked. This constitutes the preprocessing. When the truck returns to pick up the finished goods, it must again wait to load and check. This constitutes the postprocessing. If the truck route is circular, then the pre- and post-processing orders are the same. If the route is linear, then the postprocessing is done when the truck is returning to its point of origin and so is done in the reverse order of preprocessing. In both cases, all preprocessing tasks are done before the first postprocessing task.

For the case of a single master processor, Sahni [19] has considered order preserving sequencing (**OPS(1)**) and reverse order sequencing (**ROS(1)**). In the former case the pre- and post-processing tasks must be processed in the same order while in the latter these orders should be in reverse order. Optimal algorithms with complexity $\mathcal{O}(n \log n)$ have been developed for both of these cases. To facilitate later developments we provide a description of these algorithms denoted by **OOPS(1)** and **OROS(1)** respectively.

OOPS(1)

1. Jobs with $c_j > a_j$ come first in nondecreasing order of $a_j + b_j$
2. Jobs with $c_j = a_j$ come next in any order
3. Jobs with $c_j < a_j$ come last in nonincreasing order of $b_j + c_j$
4. Generate the order preserving schedule whose preprocessing tasks are ordered according to steps 1-3

OROS(1)

1. Order the jobs according to nonincreasing order of b_j
2. Generate the reverse order schedule whose preprocessing tasks are ordered according to step 1

The single master problem to minimize makespan with no restriction on the relative ordering of tasks of different jobs has not been considered before. We refer to this problem as *unconstrained minimum finish time* or **UMFT**. In light of the strong $\mathcal{NP}$-completeness of the UMFT problem, we develop an approximation algorithm in Section 3.

For master-slave systems with multiple master processors we can distinguish two classes of problems. In the first class we require both pre- and post-processing tasks to be processed by the same processor; we shall

refer to such systems as *restricted multiple master systems.* In the second class we allow the pre- and post-processing task of each job to be processed by different processors; we shall refer to such systems as *unrestricted multiple master systems.*

For unrestricted multiple master systems we need to be careful about the definition of order-preserving and reverse-order schedules as the pre- and post-processing tasks of a job may be done by different master processors.

Definition 1 *For multiple master processor systems we shall say that a schedule is order preserving iff for every pair of jobs i and j such that the preprocessing of i begins before the preprocessing of j, the postprocessing of i completes before or at the same time as the postprocessing of j.*

Definition 2 *For multiple master processor systems we shall say that a schedule is a reverse order schedule iff for every pair of jobs i and j such that the preprocessing of i begins before the preprocessing of j, the postprocessing of i completes after or at the same time as the postprocessing of j.*

In Section 4 we will develop unconstrained, order preserving and reverse order schedules for both restricted and unrestricted multiple master systems.

3 Approximation Algorithms for Unconstrained MFT

In light of the complexity status of UMFT we are motivated to investigate heuristic algorithms that have good worst case performance. If S is an unconstrained schedule, then a straightforward interchange argument shows that we may rearrange the master tasks so that all preprocessing tasks complete before any postprocessing task starts. Such a rearrangement can be done without increasing the makespan of the schedule. Further, the rearranged schedule has no preemptions. We may shift the a tasks in the rearranged schedule left so as to start at time 0 and complete at time $\sum a_i$ and the b tasks may be shifted left so as to begin as soon as their corresponding a tasks complete. The c tasks may be ordered to begin in the same order as the b tasks complete. None of these rearrangement operations affects the makespan of S. With this as motivation, we define a *canonical schedule* to be one which satisfies the following properties:

1. There are no preemptions.
2. The a tasks begin on the master at time 0 and complete at time $\sum a_i$.
3. The b tasks begin as soon as their corresponding a tasks complete.
4. The c tasks are done in the same order as the b tasks complete and as soon as possible.

It is evident that for every unconstrained schedule S, there is a corresponding canonical schedule with better or the same makespan. So, in the remainder of this section we limit ourselves to canonical schedules. Note that a canonical schedule is completely specified by giving the relative order in which the preprocessing tasks are done. As a result, such a schedule is defined by a permutation that gives the relative order in which the preprocessing tasks are done. We will use the terminology i follows (precedes) j to mean i comes after (before) j in the permutation that defines the schedule.

The next theorem finds the worst case performance of an arbitrary canonical schedule S. Let C^S be the makespan of the canonical schedule S and $C^\star$ the optimal makespan of UMFT.

Theorem 1 *For any canonical schedule S, $\frac{C^S}{C^\star} \leq 2$ and the bound is tight.*

Proof: If $C^S = \sum_i (a_i + c_i)$ then S is optimal and the error bound of 2 is valid. Else, $C^S > \sum_i (a_i + c_i)$ in which case there exists idle time on the master processor. Since S is canonical, this idle time will have to precede one or more postprocessing tasks. Let c_{i_0} be the last postprocessing task in S that starts immediately after its corresponding slave task b_{i_0}. Since there is idle time on the master, such an i_0 exists. Then,

$$C^S = \sum_{i \ precedes \ i_0} a_i + (a_{i_0} + b_{i_0} + c_{i_0}) + \sum_{i \ follows \ i_0} c_i \leq 2C^\star$$

since $a_{i_0} + b_{i_0} + c_{i_0} \leq C^\star$ and $\sum_i (a_i + c_i) \leq C^\star$.

To see that the error bound is tight consider an instance with $k+1$ jobs where k is an arbitrary positive integer. The first k jobs have processing requirements $(1, \epsilon, \epsilon)$ while the $(k+1)$-st job has requirements (ϵ, k, ϵ), $\epsilon < 1/k$. The schedule S that processes $a_{k+1} = \epsilon$ last among all preprocessing tasks has makespan $C^S = 2k + 2\epsilon$. The schedule $S^\star$ that processes a_{k+1} first among all preprocessing tasks has makespan $C^\star = k + (k+2)\epsilon$ and hence $\frac{C^S}{C^\star} \to 2$ as $\epsilon \to 0$. □

In what follows we present a heuristic whose error bound is $\frac{3}{2}$.

Heuristic H

1. Let $S_1 = \{i : a_i \leq c_i\}$ and $S_2 = \{i : a_i > c_i\}$.
2. Reorder the jobs in S_1 according to nondecreasing order of b_i
3. Reorder the jobs in S_2 according to nonincreasing order of b_i
4. Generate the canonical schedule in which the a tasks of S_1 precede those of S_2

The complexity of heuristic H is readily seen to be $O(n \log n)$. Let C^H be the makespan of the schedule generated by the above heuristic. Then,

Theorem 2 $\frac{C^H}{C^*} \leq \frac{3}{2}$ *and the bound is tight.*

Proof: See [20]. □

4 Multiple Master Systems

A versatile heuristic, *general*, that obtains multi-master schedules with an error bound of at most 2 is developed in Section 4.1. For the case of reversed order sequencing a heuristic with worst case error bound $2 - \frac{1}{m}$ (m is the number of master processors) is presented in Section 4.2.

4.1 A General Heuristic

The heuristic *general* may be used for both restricted and unrestricted systems as well as when constraints are placed between the orders in which the pre- and post- processing tasks are executed. Before presenting this heuristic, we define the *first available machine* (FAM) rule. In this, jobs are assigned to master processors one-at-a-time. Each job has a time t_i associated with it and the jobs are considered in a given order σ. When a job is considered, it is assigned to the master on which the sum of the times of already assigned jobs is the least (ties are broken arbitrarily).

Heuristic general(m)

1. For each job, let $t_i = a_i + c_i$. Sort the jobs so that $t_1 \geq t_2 \geq \cdots \geq t_n$.
2. Consider the jobs in this order and use the FAM rule to assign jobs to masters.
3. On each master, schedule the preprocessing tasks in any order from time 0 to time T where T is the sum of the preprocessing tasks of the jobs assigned to this master. The slave tasks are scheduled to begin as soon as their corresponding preprocessing tasks are complete. The postprocessing tasks are scheduled to begin as soon after the completion of their slave tasks as is feasible.

The heuristic *general*(m) constructs schedules with the property that each job's pre- and post-processing tasks are done by the same master. Hence the schedules are feasible for both the restricted and unrestricted master models. The complexity of the heuristic is readily seen to be $O(n \log n)$.

Let $C^{general}$ be the makespan of the schedule generated by heuristic *general*. Let C^*_{UMFT} and C^*_{RMFT}, respectively, be the makespans of the optimal unrestricted and restricted master system schedules.

Theorem 3 $C^{general}/C^*_{UMFT} \leq 2$ *and* $C^{general}/C^*_{RMFT} \leq 2$.

Proof: See [20]. □

To see that the bound of 2 is a tight one, consider the $n(m-1)+2$ job instance in which the first job's pre-, slave, and post-processing tasks are given by $(n-\epsilon, \epsilon, \epsilon/2)$, the next $n(m-1)$ job task times are $(1/2, \epsilon, 1/2)$ and the last job has times (ϵ, n, ϵ). Here, $0 < \epsilon < 1/2$. The jobs have been given in the order produced in step 1. The heuristic assigns jobs 1 and $n(m-1)+2$ to master 1. The remaining jobs are distributed evenly across the remaining masters. If in step 3, the first master is scheduled to process a_1 first, then $C^{general} = 2n + \epsilon$. However, $C^*_{UMFT} = C^*_{RMFT} = n + 2.5\epsilon$. The ratio approaches 2 as $\epsilon \to 0$.

Heuristic *general* may be used to obtain order preserving and reverse order schedules by modifying step 3 to produce such schedules. In fact, since optimal single master order preserving and reverse order schedules can be obtained in polynomial time ([19]), step 3 can generate optimal schedules using the jobs assigned to each master. Since the proof of Theorem 3 does not rely on how the schedule is constructed in step 3, the error bound of 2 applies even for the case of order preserving and reverse order schedules.

4.2 Restricted Reverse Order Schedules

In this subsection we develop an approximation algorithm for restricted multiple master systems in which each master processor is required to process its postprocessing tasks in an order that is the reverse of the order in which it processes its preprocessing tasks. This problem is abbreviated as $ROS(m)$ (reverse order scheduling with m masters). The $OROS(1)$ algorithm provided in Section 2 solves optimally the $ROS(1)$ problem.

The approximation algorithm, *Heuristic* $ROS(m)$, given below obtains schedules with an error bound no more than $2 - 1/m$.

Heuristic $ROS(m)$

1. Sort the jobs so that $b_1 \geq b_2 \geq \cdots \geq b_n$.
2. Consider the jobs in this order and use the FAM rule to assign jobs to masters using $t_i = a_i + c_i$.
3. On each master, schedule the preprocessing tasks in the order the jobs were assigned to the master. Schedule the postprocessing tasks in the reverse order and to begin as soon as possible after all preprocessing tasks complete.

Note that in step 1, we obtain the ordering needed to construct an $OROS(1)$ for the n jobs and that in step 3 the jobs assigned to each master are scheduled to form an $OROS(1)$ for that master. The complexity of $ROS(m)$ is easily seen to be $O(n \log n)$. To establish the error bound, we need to first establish two other results. This is done in Lemmata 1 and 2. The error bound itself is established in Theorem 4.

Let I, I', and I'' be three sets of jobs. $I = \{(a_i, b_i, c_i) | 1 \leq i \leq n\}$, I' has n jobs defined by $a'_i = c'_i = (a_i + c_i)/2$ and $b'_i = b_i$, and I'' has n jobs defined by $a''_i = c''_i = (a_i + c_i)/(2m)$ and $b'_i = b_i$. Let $C^*_I(m)$, $C^*_{I'}(m)$, and $C^*_{I''}(m)$, respectively, denote the makespans of the $OROS(m)$ for I, I', and I''.

Lemma 1 *$C^*_I(m) = C^*_{I'}(m)$ for all m.*

Proof: See [20]. □

Lemma 2 *$C^*_{I''}(1) \leq C^*_I(m)$ for all m.*

Proof: See [20]. □

Effectively, Lemmas 1 and 2 show that the makespan of the schedule produced by $OROS(1)$ on I'' is a lower bound on the optimal makespan value for $ROS(m)$ which is denoted by $C^*_I(m)$. The following theorem makes use of this result.

Theorem 4 *Let $C^{ROS}_I(m)$ be the makespan of the schedule generated by $ROS(m)$ on instance I. $C^{ROS}_I(m)/C^*_I(m) \leq 2 - 1/m$ and this bound is tight.*

Proof: See [20]. □

Note that the above result has some similarities with the problem of minimizing makespan in a two-stage hybrid flowshop considered by Lee and Vairaktarakis [11]. In this problem preprocessing tasks are executed on the machines of stage 1, postprocessing tasks are executed on the machines of stage 2, and the slave tasks are null. A heuristic with bound $2 - \frac{1}{m}$ was developed for that problem as well.

5 Conclusion

We have proposed efficient heuristics for various variants of the scheduling problem that arises when the fork-join paradigm is used for distributed computing. These heuristics are bounded performance heuristics as we are able to bound their worst case performance by a constant.

References

[1] K. Baker, *Introduction to Sequencing and Scheduling*, John Wiley, New York, 1974.

[2] E. Coffman, *Computer & Job/Shop Scheduling Theory*, John Wiley, New York, 1976.

[3] G. Chen and T. Lai, Preemptive scheduling of independent jobs on a hypercube, *Information Processing Letters*, 28, 201-206, 1988.

[4] G. Chen and T. Lai, Scheduling independent jobs on partitionable hypercubes, *Jr. of Parallel & Distributed Computing*, 12, 74-78, 1991.

[5] M. Dell'Amico, Shop problems with two machine and time lags, *Operation Research*, to appear.

[6] M. Garey and D. Johnson, *Computers and Intractability: A guide to the theory of NP-completeness*, W. H. Freeman and Co., New York, 1979.

[7] R. Graham, E. Lawler, J. Lenstra, and A. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling: A survey, *Annals of Discrete Mathematics*, 5, 287-326, 1979.

[8] S.M. Johnson, Discussion: Sequencing n jobs on two machines with arbitrary time lags, *Management Science*, 5, 299-303, 1959.

[9] W. Kern and W. Nawijn, Scheduling multi-operation jobs with time lags on a single machine, University of Twente, 1993.

[10] P. Krueger, T. Lai, and V. Dixit-Radiya, Job scheduling is more important than processor allocation for hypercube computers, *IEEE Trans. on Parallel & Distributed Systems*, 5, 5, 488-497, 1994.

[11] C.-Y. Lee and G.L. Vairaktarakis, Minimizing makespan in hybrid flowshops, *Operations Research Letters*, 16, 149-158, 1994.

[12] S. Leutenegger and M. Vernon, The performance of multiprogrammed multiprocessor scheduling policies, *Proc. 1990 ACM SIGMETRICS Conference on Measurement & Modeling of Computer Systems*, 226-236, 1990.

[13] C.-Y. Lee, R. Uzsoy and L.A.M. Vega, Efficient algorithms for scheduling semiconductor burn-in operations, *Operations Research*, 40, 4, 764-775, 1992.

[14] S. Majumdar, D. Eager, and R. Bunt, Scheduling in multiprogrammed parallel systems, *Proc. 1988 ACM SIGMETRICS*, 104-113, 1988.

[15] C. McCreary, A. Khan, J. Thompson, and M. McArdle, A comparison of heuristics for scheduling DAGS on multiprocessors, *8th International Parallel Processing Symposium*, 446-451, 1994.

[16] L.G. Mitten, Sequencing n jobs on two machines with arbitrary time lags, *Management Science*, 5, 293-298, 1959.

[17] A. Orman and C. Potts On the complexity of coupled-task scheduling, *Discrete Applied Mathematics*, To appear.

[18] S. Sahni, Scheduling multipipeline and multiprocessor computers, *IEEE Trans on Computers*, C-33, 7, 637-645, 1984.

[19] S. Sahni, Scheduling master-slave multiprocessor systems, *IEEE Trans. on Computers*, 45, 10, 1996, 1195-1199.

[20] S. Sahni and G. Vairaktarakis, The master-slave paradigm in parallel computer and industrial settings, *Journal of Global Optimization*, Kluwer Academic Publishers, 9: 357-377, 1996.

[21] W. Szwarc, On some sequencing problems, *Naval Research Logistics Quarterly*, 15, 127-155, 1968.

[22] Y. Zhu and M. Ahuja, Preemptive job scheduling on a hypercube, *Proc. 1990 International Conference on Parallel Processing*, 301-304, 1990.