Scheduling Master-Slave Multiprocessor Systems *

Sartaj Sahni

Computer & Information Sciences Department

University of Florida

Gainesville, FL 32611, USA

Abstract

We define the master-slave multiprocessor scheduling model and provide several applications for the model. $O(n \log n)$ algorithms are developed for some of the problems formulated and some others are shown to be NP-hard.

1 Introduction

The problem of scheduling a multiprocessor computer system has received considerable attention [3, 4, 5, 6, 7, 8, 9, 10]. In this paper, we develop a model to schedule a parallel computer system in which the parallel computer operates under control of a host processor. The host processor is referred to as the *master* processor and the processors in the parallel computer are referred to as the *slave* processors. The nCube hypercube is an example of such a parallel computer system. When programming such a system, one typically writes a program that runs on the master computer. This is a sequential program that spawns parallel tasks to be run on the slave processors. When these tasks complete, the sequential thread on the master continues and possibly (later) spawns a new set of parallel tasks on the slaves, and so on. The number of parallel tasks spawned is always less than or equal to the number of slave processors.

If we examine the execution profile of such a computer system, we see that, in general, there are time intervals in which only the master is active, only the slaves are active, both the master and the slaves are active. With each task to be run on a slave processor, we may associate three activities:

1. *Preprocessing.* This is the work the master has to do to collect the data needed by the slave and includes the overhead involved in initiating the transfer of this data as well as the code to be run by the slave.

*This work was supported in part by the National Science Foundation under grant MIP-9103379

2. *Slave work.* This includes the work the slave must do to complete the assigned computation task, receive the data and code from the master, and transfer the results back to the master. Into this work, we also include the transmission dalays experienced in receiving the data and code from the time the master initiates transmission to the time the slave receives as well from the time the slave initiates transmission of the results to the time the master receives the results.

3. *Postprocessing.* This is the work the master must do to receive the results and store them in the desired format. It also includes any checking or data combining work the master may do on the results.

As examples, consider the following:

1. If the master processor reaches a point in its computation when two matrices A and B are to be multiplied, then it would partition the matrix multiplication problem into p (p is the number of slave processors) such problems each involving a submatrix of A and B. These submatrix pairs together with the multilpication code would be transmitted to the p slaves (one pair per slave); the slaves would execute the code once they have received the data and code; the slaves would transmit the product submatrix back to the master; and finally the master would store the received submatrix of C into the proper locations in C. Since matrix multiplication is a highly structured problem, it is possible to partition the matrices so that the amount of pre-processing work for each slave task is the same, the amount of post-processing work is the same for each task, the amount of work done by each slave is the same (This assumes uniform data transmission times between the master and slaves.). When the submatrices are square, the task pre-processing time is roughly twice the post-processing time.

2. Suppose we are working with a computer vision or VLSI CAD problem that involves objects in a two-dimensional region. To process these objects, the region may be divided into p parts; each part is sent to a slave processor; the results are returned to the master. Because of the nonuniform distribution of objects and an often imposed requirement that the region be partitioned using regular geometries (for example, we may require a rectangular region be partitioned using either vertical or horizontal cut lines so that the pre- and post- processing taskes are simplified), the number of objects in each partition may vary widely. As a result, the amount of pre-processing work varies widely from task to task, and so also does the amount of work assigned to individual slaves as well as the post-processing work (which may now also invlove worrying about partition boundary effects).

3. A common parallel programming paradigm involves the use of a single main computational thread that employs the fork and join operations to spawn parallel tasks/threads and then to synchronize following the completion of these tasks. The fork operation involves the passing

of varying amounts of data to remote processors that will execute the spawned threads (we assume that each spawned thread will execute on a different processor). These processors will, in turn, return the results to the main thread. So, associated which each of the spawned threads, we have three amounts of work:

(a) Preprocessing by main thread. This is the work needed to initiate the thread. It includes the effort expended in collecting the data needed by the remote processor (in case of a distributed memory environment); overheads involved in transmitting this data to the remote processor, etc.

(b) Work done in the thread. This includes the computational activity assigned to the remote processor, the work this processor must do to recieve the data and send back the results, and the transmission times in receiving and sending.

(c) Postprocessing by the main thread. This represents the effort expended in receiving the answers and performing any post-processing on them.

Since the different threads may execute very different pieces of code, the relative values of the amounts of work invloved in pre-processing, in thread execution, and in post-processing can vary widely from thread to thread.

In addition to applications to parallel computer scheduling, the master-slave model can be used in industrial settings.

1. A consolidator receives 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 pre-processing 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, load the goods on to the trucks, 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 post-processing.

2. In certain maintenance/repair environments, the maintenance manager examines the maintenance tasks to be performed and writes up a formal work order for each and prepares the task for maintenance; the work orders are executed by different maintenance crews that are dispatched following the receipt of the work order; upon completion, the maintenance manager inspects the completed work and signs an acceptance document.

The master-slave scheduling model defined here has the following attributes:

1. there is a single master processor

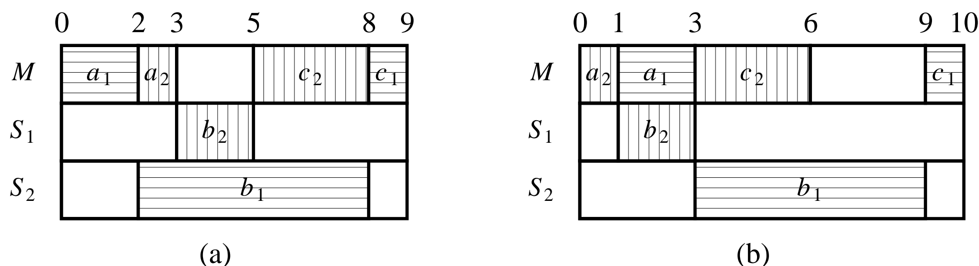

Figure 1: Example schedules

2. thre are as many slave processors as parallel jobs
3. associated with each job, there are three tasks: pre-processing (performed by the master), slave work (performed by the slaves), and post-processing (performed by the master).
4. for each job, the tasks are to be performed in the order: pre-processing, slave work, post-processing.

The master-slave scheduling model may be regarded as a generalized 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

Let $a_i > 0$, $b_i > 0$, and $c_i > 0$, respectively, denote the time needed to perform the three tasks associated with job i and let n be the number of jobs as well as the number of slaves. 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 pre-processing of job 1 is handled first by the master; all other tasks are begun at the earliest possible time. M denotes the master processor and S_1 and S_2 denote the slaves. The finish time (i.e., the earliest time at which all tasks have been completed) is 9. The schedule that results when the master pre-processes job 2 first and all other tasks are begun at the earliest possible time is shown in Figure 1 (b). This has a finish time of 10. The mean finish time (i.e., the flow time or the average of the finish times of the jobs) is 8.5 for schedule (a) and 8 for schedule (b).

In this paper, we shall use the notations a_i, b_i, c_i to represent both the tasks of job i as well as the time needed to complete these tasks.

When we are scheduling a parallel computer system using the above model, we are interested in schedules that minimize the finish time. However, when scheduling industrial systems using the

above model we may be interested in minimizing either the schedule finish time or the mean finish time of the jobs.

Notice that in both the schedules of Figure 1, 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. Neither of the schedules use preemption. Clearly preemptions on the slave processors are unneccessary. It is easy to verify that preemptions on the master processor are unnecessary if we wish to minimize finish time but are useful in reducing the mean finish time. Another interesting feature of the schedules of Figure 1 is that in one the post-processing is done in the reverse order of the pre-processing while in the other the pre- and post- processing orders are the same. In some settings, we may require that schedules satisfy this. For example, this could simplify the post-processing 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 post-processing be done in the same relative order as the pre-processing. Another discipline that might be imposed on the master is that it complete all the pre-processing tasks before beginning the first post-processing task. Both 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 pre-processing. When the truck returns to pick up the finished goods, it must again wait to load and check. This constitutes the post-processing. If the truck route is circular, then the pre-processing and post-processing orders are the same. If the route is linear, then the post-processing is done when the truck is returning to its point of origin and so is done in the reverse order of pre-processing. In both cases, all pre-processing tasks are done before the first post-processing task.

For the industrial setting, one could generalize the model to permit several master processors. In both the computer and industrial settings, one could have nonhomogenous slave processors. In this case, with each slave task we have a list of slaves on which it can be run and the time needed on each slave.

There is a large variety of schedule optimization problems that one can formulate for the master-slave model. In this paper, we study just a few of these.

1. In Section 2, we show that obtaining minimium finish time no-wait-in-process (MFTNW) schedules is NP-hard for each of the following scheduling disciplines:
   (a) Each job's pre-processing must be done before its post-processing. No other constraint is put on the master.
   (b) The pre-processing and post-processing orders are the same.

   In this section, we also develop an $O(n \log n)$ algorithm to obtain MFTNW schedules when

the pre-processing order is required to be the reverse of the post-processing order.

2. In Section 3, we develop $O(n \log n)$ algorithms to minimize finish time for each the following scheduling constraints:
   (a) The pre-processing and post-processing orders are the same.
   (b) The pre-processing order is the reverse of the post-processing order.

2 No Wait In Process

Our NP-hard proofs use the subset sum problem which is known to be NP-hard [GARE79]. This problem is defined below:

Input A collection of positive integers x_i, $1 \leq i \leq n$ and a positive integer M.

Output "Yes" iff there is a subset with sum exactly equal to M.

From any instance of the subset sum problem, we may construct an equivalent instance of MFTNW as below:

$a_i = c_i = x_i/2$, $b_i = \epsilon$, $1 \leq i \leq n$

$a_{n+1} = c_{n+1} = S - M + 1$, $b_{n+1} = M + n\epsilon$

$a_{n+2} = c_{n+2} = M + 1$, $b_{n+2} = S - M + n\epsilon$

where S is the sum of the x_i's and $0 < \epsilon < 1/n$.

In the no wait case, the master processor cannot preempt any job as such a preemption would violate the no wait constraint. In the preceding section, we remarked that there is no advantage to preemptions on slave processors. So, we may assume non-preemptive schedules. Since $a_{n+1} = c_{n+1} > b_{n+2}$, the pre-processing and/or post-processing tasks of job $n+1$ cannot be done while a slave is working on job $n+2$. Similarly the pre-processing and/or post-processing tasks of job $n+2$ cannot be overlapped with the slave task of job $n+1$. Hence, every no wait schedule has a finish time f that is at least the sum of the task times of these two jobs. That is,

$$f \geq a_{n+1} + b_{n+1} + c_{n+1} + a_{n+2} + b_{n+2} + c_{n+2} = 3S + 4 + 2n\epsilon$$

There are exactly two templates for schedules with this length. One has job $n+1$ processed before job $n+2$ and the other has $n+2$ preceding $n+1$ (see Figure 2).

To complete the schedule using either of the templates and not exceed the finish time of $3S + 4 + 2n\epsilon$, some of the remaining jobs must fully overlap with b_{n+1} and the remainder with b_{n+2}. For this, the sum of the first groups task times cannot exceed $b_{n+1} = M + n\epsilon$ and the sum of second groups task times cannot exceed $b_{n+2} = S - M + n\epsilon$. Since the sum of the task times for the remaining jobs is $S + n\epsilon$ and $\epsilon < 1/n$, the only way to accomplish this is when there is a subset of the x_i's that sums to M. Hence, MFTNW is NP-hard.

We can modify the above proof to show that order preserving MFTNW (OP-MFTNW) is also NP-hard. The task times for the $n+2$ jobs are:

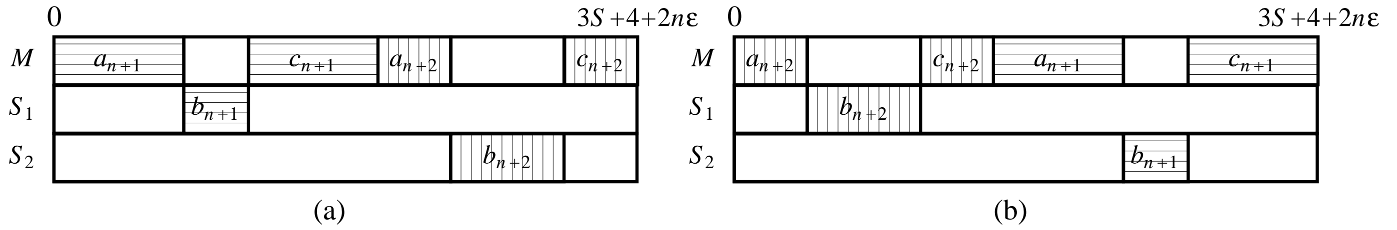

Figure 2: Templates for NP-hard proof

$a_i = c_i = x_i,\ b_i = S - x_i + 1,\ 1 \le i \le n$

$a_{n+1} = c_{n+1} = S - M + 1,\ b_{n+1} = M$

$a_{n+2} = c_{n+2} = M + 1,\ b_{n+2} = S - M$

The finish time is at least the sum of the master processor task times. So,

$$f \ge \sum a_i + \sum c_i = 4S + 4$$

It is easy to see that there is an order preserving no wait schedule with length $4S+4$ whenever there is a subset of the s_i's that sums to M. We shall show that whenever there is a schedule with this length, there is a subset that sums to M.

As in the previous proof, the tasks of jobs $n+1$ and $n+2$ cannot overlap. So, jobs $n+1$ and $n+2$ are done in sequence. Suppose that job $n+1$ is done before $n+2$ (the case $n+2$ before $n+1$ is similar). Since the sum of the task times for these two jobs is $3S+4$, the only way to finish processing by time $4S+4$ is for the master processor to be busy throughout the time the slaves are working on tasks b_{n+1} and b_{n+2} and for task a_{n+1} to begin by time S. The first requirement means that there are only S other time units when the master can work on the remaining S units of pre- and post- processing needed by jobs $1, \cdots, n$. There are three cases to consider:

Case 1: *There is at least one job whose pre-processing is done before a_{n+1} and whose post-processing is done after a_{n+1}.* Let u, $1 \le u \le n$, be the first such job. The post-processing of this job must be done while a slave is working on b_{n+1} as $a_{n+1} + b_{n+1} = S + 1 > b_u = S - x_u + 1$. Hence, we have the situation shown in Figure 3 (a).

The tasks (if any) scheduled between a_{n+1} and c_u, must be pre-processing tasks. To see this, note that to schedule a post-processing task here, the corresponding pre-processing task must have

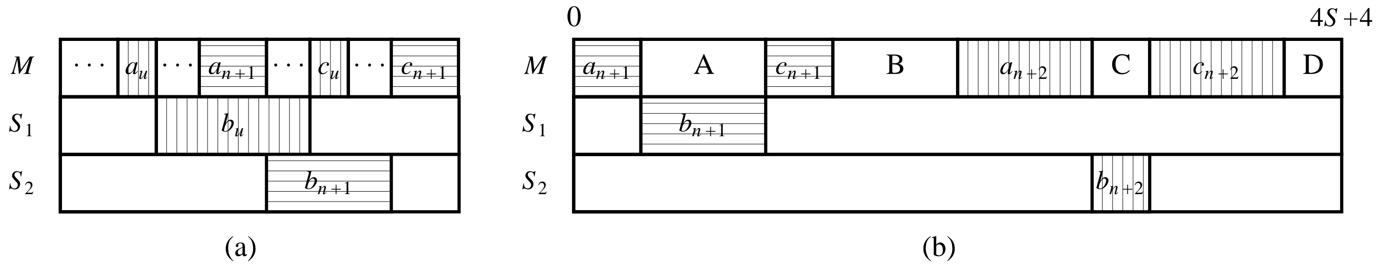

Figure 3: Templates for order preserving NP-hard proof

been scheduled either before a_u (in which case u is not the first job with pre-processing before a_{n+1} and post-processing after a_{n+1}), or in between a_u and a_{n+1} (in which case the order requirement is violated as the post-processing of this job precedes that of u), or between a_{n+1} and c_u (which is not possible as the sum of task lengths for each of jobs $1 \cdots n$ exceeds $S+1$ which in turn is larger than b_{n+1}.

The tasks scheduled between a_u and a_{n+1} are either post-processing tasks of jobs started before a_u or pre-processing tasks of jobs that will finish after c_u (because of the order requirement). Hence, the tasks beginning with a_u and ending just before c_u that are processed by the master correspond to different jobs. The total amount of time from the beginning of a_u to the start of c_u is $a_u + b_u = S + 1$. Subtracting a_{n+1} from this leaves us with M units of time, all of which must be utilized by the master in order for the schedule to complete by $4S + 4$. This can happen iff there is a subset of the x_i's that sums to M.

Case 2: *There is at least one job whose pre- and post- processing are done before* a_{n+1}. Let u be one such job. Since the sum of the task lengths of u is $S + x_u + 1$, task a_{n+1} cannot begin until $S + x_u + 1$ and so the schedule cannot complete by $4S + 4$. Therefore, this case is not possible.

Case 3: *Task* a_{n+1} *is the first task scheduled.* Figure 3 (b) shows the scheduling template for this case. For the schedule length to be $4S + 4$, the total time represented by the regions A, B, C, and D must be $2S$. The master processor cannot be idle in any of these regions as the amount of pre- and post- processing not scheduled in Figure 3 (b) is exactly $2S$. Because of the order constraint, in region A, we can schedule only the pre-processing of some subset of the jobs $1, \cdots, n$. Hence, there needs to be a subset of the x_i's that sums to $b_{n+1} = M$.

Hence, OP-MFTNW is NP-hard.

The MFTNW problem is quite easy to solve when the post-processing is to be done in the reverse order of the pre-processing. In this case, there is at most one feasible solution. Hence, if such a solution exists, it has minimum finish time and also minimum mean finish time. Note that when there is no ordering constraint between pre- and post- processing and also when these two orders are required to be the same, there is always at least one feasible solution (i.e., process the jobs in sequence using any permutation). When the post-processing order is required to be the reverse of the pre-processing order and no wait is permitted in process, then the processing of the jobs must be fully nested. That is, the processing of the j'th scheduled job must begin and end while a slave is working on the $j-1$'th job. As a result, if jobs are pre-processed in the order 1, 2, $\cdots$, n, then the following must be true:

$$b_i \geq a_{i+1} + b_{i+1} + c_{i+1}, \quad 1 \leq i < n \tag{1}$$

Since the a_j's and c_j's are positive, it follows that:

$$b_1 > b_2 > \cdots > b_n \tag{2}$$

The preceding inequality implies a unique ordering of the jobs. The algorithm to determine feasibilty, as well as a feasible schedule that minimizes both the finish and mean finish times is:

1. **(Verify Equation 2)** Sort the jobs into decreasing order of b_j's. If such an ordering does not exist, there is no feasible schedule. In this case, terminate.
2. **(Verify Equation 1)** For $i = 1, \cdots, n-1$, verfify that $b_i \geq a_{i+1} + b_{i+1} + c_{i+1}$. If there is an i for which this is not true, then there is no feasible schedule. In this case, terminate.
3. The minimum finish time and mean finish time schedule is obtained by pre-processing the jobs in the order determined in step 1.

The complexity of the above algorithm is readily seen to be $O(n \log n)$.

3 Same Pre- and Post- Processing Orders

In this section, we develop an $O(n \log n)$ algorithm to construct an order preserving minimum finish time (OPMFT) schedule. Without loss of generality, we place the following restrictions on schedules we consider in this section:

R1: The schedules are non-premptive.

R2: Slave tasks begin as soon as their corresponding pre-processing tasks are complete.

R3: Each post-processing task begins as soon after the completion of its slave task as is consistent with the order preserving constraint.

First, we establish some properties of order preserving schedules that satisfy these assumptions.

Definition 1 *A* canonical *order preserving schedule (COPS) is an order preserving schedule in which (a) the master processor completes the pre-processing tasks of all jobs before beginning any of the post-processing tasks, and (b) the pre-processing tasks begin at time zero and complete at time* $\sum_{i=1}^{n} a_i$.

Because of restrictions R1 – R3, every COPS is uniquely described by providing the order in which the pre-processing is done.

Lemma 1 There is a canonical OPMFT schedule.

Proof: Consider any non-canonical OPMFT schedule. Let c_j be the first post-processing task that the master works on. Since the schedule is non-canonical, there is a pre-processing task that is executed at a later time. Let a_i be the first of these. Slide a_i to the left so that it begins just after the pre-processing task (if any) that immediately precedes c_j (if there is no such task preceding c_j, then slide a_i left so as to start at time 0). Slide the post-processing tasks beginning with c_j and ending at the post-processing task that immediately preceded a_i (before it was moved) rightwards by a_i units. Slide the slave and post-processing tasks left so as to satisfy restrictions R2 and R3. The result is another OPMFT schedule that is closer to canonical form. By repeating this transormation at most $n - 1$ times we can obtain a canonical OPMFT schedule. □

Lemma 2 If $a_i = c_i$, $1 \leq i \leq n$, then every COPS is an OPMFT schedule.

Proof: Because of the preceding lemma, it is sufficient to show that all COPS have the same length. Each COPS is uniquely identified by the order in which the pre-processing tasks are executed. We shall show that exchanging two adjacent jobs in this ordering does not increase the schedule length. Since we can go from one permutation to any other via a finite sequence of adjacent exchanges, it follows that no matter what the pre-processing order, canonical schedules have the same finish time when jobs have equal pre- and post- processing times. Hence, all COPS are OMFT schedules.

Consider two jobs j and $j+1$ that are adjacent in the pre-processing order (Figure 4). Let t_j and t_{j+1}, respectively, be the times at which the master begins tasks c_j and c_{j+1}. Slide job $j+1$ left by a_j so that all its tasks begin a_j units earlier than before, slide tasks a_j and b_j right by a_{j+1} units so that they begin a_{j+1} units later than before, and move task c_j so that it begins just after c_{j+1} finishes. As a result, task c_{j+1} now begins at $t_{j+1} - a_j = t_{j+1} - c_j \geq t_j$. Hence, the rescheduling of job $j+1$ does not result in the master working on two or more jobs simultaneously. In addition, the post-processing of job $j+1$ does not begin until after its slave task is complete. The post-processing of task c_j now begins at $t_{j+1} - a_j + c_{j+1} = t_{j+1} - c_j + a_{j+1} \geq t_j + a_{j+1}$ which is greater than or equal to the time at which the slave finishes b_j. Task c_j finishes at $t_{j+1} + a_{j+1}$. Hence, the schedule for the remaining jobs is unchanged. □

Lemma 3 *Consider the COPS defined by some permutation* σ. *Assume that job* j *is pre-processed immediately before job* $j+1$ *(i.e.,* j *immediately precedes* $j+1$ *in* σ*). If* $c_j \leq a_j$ *and* $c_{j+1} \geq a_{j+1}$*, then*

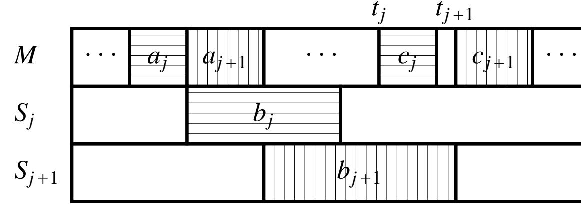

Figure 4: Figure for Lemma 2

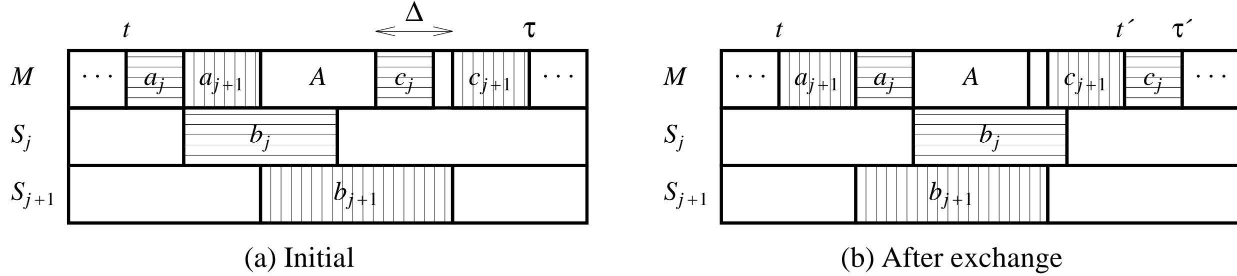

Figure 5: Figure for Lemma 3

the schedule length (i.e., its finish time) is no less than that of the COPS obtained by interchanging j and $j+1$ in σ.

Proof: A diagram of the schedule with job j immediately preceding job $j+1$ is shown in Figure 5 (a). In this figure, t is the time at which the pre-processing of job j starts, A is the elapsed time between the completion of task a_{j+1} and the start of the post-processing of job j (note that A $\geq \sum_{k \text{ follows } j+1} a_k + \sum_{k \text{ precedes } j} c_k$), $\Delta > 0$ is the time between the start of c_j and c_{j+1}, and τ is the time at which c_{j+1} completes.

Let σ' be the permutation obtained by interchanging jobs j and $j+1$ in σ. The schedule corresponding to σ' is shown in Figure 5 (b). Let t' and τ', respectively, be the times at which c_{j+1} and c_j finish in this schedule. If $\Delta \geq a_j$, then $t' \leq \tau - a_j$. Also, from Figure 5 (a), we observe that $b_j \leq a_{j+1} + A \leq c_{j+1} + A$. So, b_j finishes by t' in Figure 5 (b). Hence, $\tau' = t' + c_j \leq \tau - a_j + c_j$ $\leq \tau$. As a result, the post-processing tasks of the remaining jobs can be done so as to complete at or before their completion times in σ and the interchanging of j and $j+1$ does not increase the

schedule length.

If $\Delta < a_j$, then c_{j+1} starts at time $t + a_{j+1} + a_j + A$ in σ'. So, $t' = t + a_{j+1} + a_j + A + c_{j+1}$. The time at which b_j finishes in σ' is $t + a_{j+1} + a_j + b_j \leq t + 2a_{j+1} + a_j + A \leq t + a_{j+1} + a_j + A + c_{j+1}$ $= t'$. So, c_j finishes at $t' + c_j = t + a_{j+1} + a_j + A + c_{j+1} + c_j \leq \tau$. Consequently, the OPS defined by σ' has a finish time that is $\leq$ that of the OPS defined by σ. □

Theorem 1 There is an OPMFT schedule which is a COPS in which the pre-processing order satisfies the following:

1. jobs with $c_j > a_j$ come first
2. those with $c_j = a_j$ come next
3. those with $c_j < a_j$ come last

Proof: Immediate consequence of Lemma 3. □

Lemma 4 Let σ define an OPMFT COPS that satisfies Theorem 1. Its length is unaffected by the relative order of jobs with $a_j = c_j$.

Proof: Follows from Lemma 3. □

Lemma 5 There is an OPMFT COPS in which all jobs with $c_j > a_j$ are at the left end in non-decreasing order of $a_j + b_j$.

Proof: From Theorem 1, we know that there is an OPMFT COPS in which all jobs with $c_j > a_j$ are at the left end. Let $\sigma = (1, 2, \cdots, n)$ define such an OPMFT COPS. Let j be the least integer such that:

1. $a_j + b_j > a_{j+1} + b_{j+1}$
2. $c_j > a_j$
3. $c_{j+1} > a_{j+1}$

If there is no such j, then the lemma is established. So, assume that such a j exists. Figure 5 (a) shows the relevant part of the schedule. A denotes the time span between the finish of task a_{j+1} and the finish of the task that immediately precedes c_j (in the figure, this happens to coincide with the start of c_j). Figure 5 (b) shows the relevant part of the schedule, σ', that results from interchanging the jobs j and $j+1$. We shall show that $\tau' \leq \tau$. As a result, the finish time of σ' is no more than that of σ. So, σ' is also an OPMFT schedule. By repeated application of this exchange process, σ is transformed into an OPMFT that satisfies the lemma.

case(a) $b_j \leq a_{j+1} + A$ and $b_{j+1} \leq A + a_j$

Now, $b_j < c_{j+1} + A$ and $b_{j+1} < A + c_j$. So, $\tau = t + a_j + a_{j+1} + A + c_j + c_{j+1} = \tau'$.

case(b) $b_j \leq a_{j+1} + A$ and $b_{j+1} > A + a_j$

The conditions for this case imply that $A + a_j + b_j < A + a_{j+1} + b_{j+1}$ or $a_j + b_j < a_{j+1} + b_{j+1}$ which contradicts the assumption on j. Hence, this case cannot arise.

case(c) $b_j > a_{j+1} + A$ and $b_{j+1} \le A + a_j$

Since, $c_j > a_j$, $b_{j+1} < A + c_j$, $\tau = t + a_j + b_j + c_j + c_{j+1}$, and $\tau' = t + a_{j+1} + a_j + \max\{A + c_{j+1}, b_j\} + c_j$. For τ' to be $\le \tau$, we need:

$$b_j + c_{j+1} \ge a_{j+1} + \max\{A + c_{j+1}, b_j\}$$

So, if $b_j \ge A + c_{j+1}$, we need $b_j + c_{j+1} \ge a_{j+1} + b_j$ or $c_{j+1} \ge a_{j+1}$. This is true by choice of j. If $b_j < A + c_{j+1}$, we need $b_j + c_{j+1} \ge a_{j+1} + A + c_{j+1}$ or $b_j \ge a_{j+1} + A$. This is part of the assumption for this case.

case(d) $b_j > a_{j+1} + A$ and $b_{j+1} > A + a_j$

This time, $\tau = t + a_j + \max\{b_j + c_j, a_{j+1} + b_{j+1}\} + c_{j+1} = t + \max\{a_j + b_j + c_j + c_{j+1}, a_j + a_{j+1} + b_{j+1} + c_{j+1}\}$, and $\tau' = t + a_{j+1} + \max\{b_{j+1} + c_{j+1}, a_j + b_j\} + c_j = t + \max\{a_{j+1} + b_{j+1} + c_{j+1} + c_j, a_j + b_j + c_j + a_{j+1}\}$. Since, $a_j + b_j > a_{j+1} + b_{j+1}$, $a_j + b_j + c_j + c_{j+1} > a_{j+1} + b_{j+1} + c_{j+1} + c_j$. Also, since $c_{j+1} > a_{j+1}$, $a_j + b_j + c_j + c_{j+1} > a_j + b_j + c_j + a_{j+1}$. Hence, $\tau > \tau'$. □

Lemma 6 There is an OPMFT COPS in which all jobs with $c_j < a_j$ are at the right end in non-increasing order of $b_j + c_j$.

Proof: Similar to that of Lemma 5. □

Theorem 2 There is an OPMFT COPS in which the pre-processing order satisfies the following:

1. jobs with $c_j > a_j$ come first and in non-decreasing order of $a_j + b_j$
2. those with $c_j = a_j$ come next in any order
3. those with $c_j < a_j$ come last and in non-increasing order of $b_j + c_j$

Proof: This follows from Theorem 1 and the fact that the proofs of Lemmas 4, 5, 6 are local to the portion of the schedule they are applied to. □

Theorem 2 results in the simple $O(n \log n)$ algorithm given below to find a pre-processing order that defines a COPS which is an OPMFT schedule.

Step 1: Partition the jobs into three sets L, M, and R such that $L = \{j | c_j > a_j\}$, $M = \{j | c_j = a_j\}$, and $R = \{c_j < a_j\}$.

Step 2: Sort the jobs in L such that $a_j + b_j \le a_{j+1} + b_{j+1}$. Let $\bar{L}$ be the resulting ordered sequence.

Step 3: Sort the jobs in R such that $b_j + c_j \ge b_{j+1} + c_{j+1}$. Let $\bar{R}$ be the resulting ordered sequence.

Step 4: The pre-processing order for the COPS is: $\bar{L}$ followed by the jobs in M in any order followed by $\bar{R}$.

4 Reverse Order Post-processing

While there are no-wait-in-process master-slave instances that are infeasible when the post-processing order is required to be the reverse of the pre-processing order, this is not the case when the no-wait constraint is removed. For any given pre-processing permutation, σ, we can construct a reverse-order schedule as below:

1. the master pre-processes the n jobs in the order σ
2. slave i begins the slave processing of job i as soon as the master completes its pre-processing
3. the master begins the post-processing of the last job (say k) in σ as soon as its slave task is complete
4. the master begins the post-processing of job $j \neq k$ at the later of the two times (a) when it has finished the post-processing of the succesor of j in σ, and (b) when slave j has finished b_j

Schedules constructed in the above manner will be referred to as *canonical reverse order schedules (CROS)*. Given a pre-processing permutation σ, the corresponding CROS is unique. It is easy to establish that every minimum finish-time reverse order (ROMFT) schedule is a CROS. So, we can limit ourselves to finding a minimum finish-time CROS.

Lemma 7 *Let $\sigma = (1, 2, \cdots, n)$ be a pre-processing permutation. Let $j < n$ be such that $b_j < b_{j+1}$. Let σ' be obtained from σ by interchanging jobs j and $j+1$. Let τ and τ', respectively, be the finish times of the CROSs S and S' corresponding to σ and σ'. $\tau' \leq \tau$.*

Proof: If $j > 1$, then let t be the time at which job $j-1$ finishes in S and S'. If $j = 1$, let $t = 0$. Let s_j (s'_j) be the time at which task b_j finishes in S (S'). Let s_{j+1} and s'_{j+1} be similarly defined. From the definition of a CROS, it follows that:

$$s_j = \sum_1^j a_k + b_j \qquad s_{j+1} = \sum_1^{j+1} a_k + b_{j+1} \tag{3}$$

$$s'_j = \sum_1^{j+1} a_k + b_j \qquad s'_{j+1} = \sum_1^{j+1} a_k - a_j + b_{j+1} \tag{4}$$

Let q (q') be the time at which c_j (c_{j+1}) finishes in σ (σ'). It is sufficient to show that $q' \leq q$. We see that:

$$\begin{aligned} q &= \max\{\max\{t, s_{j+1}\} + c_{j+1}, s_j\} + c_j \\ &= \max\{t + c_j + c_{j+1}, s_{j+1} + c_j + c_{j+1}, s_j + c_j\} \end{aligned} \tag{5}$$

and

$$\begin{aligned} q' &= \max\{\max\{t, s'_j\} + c_j, s'_{j+1}\} + c_{j+1} \\ &= \max\{t + c_j + c_{j+1}, s'_j + c_j + c_{j+1}, s'_{j+1} + c_{j+1}\} \end{aligned} \tag{6}$$

From Equations 3, 4, 5, and the inequality $b_j < b_{j+1}$, we obtain:

$$\begin{aligned} s'_j + c_j + c_{j+1} &= s_{j+1} + b_j - b_{j+1} + c_j + c_{j+1} \\ &< s_{j+1} + c_j + c_{j+1} \\ &\leq q \end{aligned} \quad (7)$$

and

$$\begin{aligned} s'_{j+1} + c_{j+1} &= s_{j+1} - a_j + c_{j+1} \\ &< s_{j+1} + c_{j+1} \\ &< s_{j+1} + c_{j+1} + c_j \\ &\leq q \end{aligned} \quad (8)$$

From Equations 7, 8, 5, and 6, it follows that $q' \leq q$. □

Theorem 3 The CROS defined by the ordering $b_1 \geq b_2 \geq \cdots \geq b_n$ is an ROMFT schedule.

Proof: Follows from Lemma 7. □

Using Theorem 3, one readily obtains an $O(n \log n)$ algorithm to construct an ROMFT schedule.

5 Conclusion

In this paper, we have introduced and motivated the master-slave scheduling model. We have shown that obtaining minimum finish-time schedules under the no-wait-in-process constraint is NP-hard when the schedule is required to be order preserving as well as when no constraint is imposed between the pre- and post- processing orders. The no-wait-in-process minimum finish time problem is solvable in $O(n \log n)$ time when the post-processing order is required to be the reverse of the pre-processing order.

When the no-wait constraint is eliminated, OPMFT as well as ROMFT schedules can be found in $O(n \log n)$ time.

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] 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.

[6] 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.

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

[8] 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.

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

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