Scheduling Parallelizable Jobs on Multiprocessors†

Ching-Chih Han and Kwei-Jay Lin

Department of Computer Science
University of Illinois
1304 West Springfield Avenue
Urbana, Illinois 61801

Abstract

This paper analyzes the effect of parallel execution on the complexity of scheduling hard real-time jobs on multiprocessors. In particular, we study the scheduling problem in which a job may be parallelized and executed on any number of processors concurrently. In hard real-time systems, each job must complete before a deadline. Parallelization gives a scheduler the flexibility to allocate more processors to a job whose deadline is near. Unfortunately, with this flexibility, some of the multiprocessor scheduling problems are very difficult. We prove the NP-hardness of scheduling parallelizable jobs where each job has a fixed priority. We thus propose a heuristic algorithm for finding an approximate job partition on two processors. Simluation results show that the heuristic algorithm usually has a very good performance.

1. Introduction

A real-time system usually consists of a set of *jobs* to be executed. Jobs may be *periodic* if they are to be executed periodically; otherwise, we say they are *aperiodic*. Each job has a *ready* time when the job is ready to be executed, a *deadline* when the job must be finished, and its *execution* time which is the amount of time required to finish the job. In this paper, we assume all these timing properties are predetermined. With such timing requirements, we want to find a *feasible* schedule for a system in which every job begins its execution after its ready time and completes before its deadline.

A deadline for a computation is said to be *hard* if any result produced by the computation must be before the deadline, or the result is useless. A real-time system with hard deadlines is called a *hard real-time system*. In hard real-time systems, jobs must not only be functionally correct, but must also meet strict timing requirements. A hard deadline cannot be missed, even just by a small amount of time, since the host system may be in the middle of a safety critical event. Missing a hard deadline may cause a job failure, a system crash, or even an environmental disaster. Job scheduling thus plays an important role in the design of real-time systems.

Future real-time systems are likely to be built on multiprocessor architectures. Efficient scheduling algorithms for multiprocessors are needed so that jobs will have predictable behaviors. Unfortunately, many multiprocessor scheduling problems have been shown to be NP-complete. For example, the problem of scheduling nonpreemptive jobs on two processors in order to minimize the overall completion time of jobs in a system is NP-complete [6]. One possible way to handle the problem is to employ job parallelization. Many task partitioning and distribution algorithms have been developed recently. Rather than running on a single processor, many computations can now be divided into parallel components and executed on more than one processor concurrently. This gives schedulers more flexibility since they can allocate several processors to execute a job in parallel to reduce the execution time.

This paper investigates the effect of job parallelization on the complexity of real-time scheduling on multiprocessors. In particular, we show that with the parallel execution capability, some of the multiprocessor scheduling problems are solvable in polynomial time. Unfortunately, many others remain intractable. The remainder of this paper is organized as follows: Section 2 defines the multiprocessor scheduling problem with parallelization and the basic policy for job scheduling. In Section 3, we analyze the effect of parallel execution capability on several basic problems when all jobs are aperiodic. In Section 4, we discuss the problems for scheduling periodic jobs with parallel execution capability, and show that one of the problem is NP-hard. Section 5 presents a heuristic algorithm that can be used to provide feasible schedules for some of the problems defined in Section 3. Our conclusions are presented in Section 6.

2. Background

An assumption in many scheduling problems is that each job can be executed on only one processor at a time. This assumption was modified in [2,3] such that different jobs are to be executed on different, but fixed, numbers of processors at a time. With recent development of parallel algorithms and architectures, however, neither of these assumptions remains necessary in many systems. For example, using the Parallel Random Access Machine (PRAM) model [10], algorithms requiring $O(n)$ time on one processor can now be executed on m processors in $O(n/m + \log m)$ time. Cvetanovic [4] analyzed the effects of problem partitioning, allocation and granularity on the performance of multiprocessor systems. In both works, jobs are not required to be executed on some fixed number of processors. Instead, they can be run on a variable number of processors.

†This work was supported in part by contracts from the ONR N00014-87-K-0827 and N00014-89-J-1181.

With such a parallel execution capability, the scheduler has more flexibility in assigning the jobs to processors. Real-time jobs thus have a better chance to meet their timing constraints. This can be seen from the following example.

Example 2.1. A set of jobs J = $\{J_1, J_2, J_3\}$, where job J_i has execution time t_i and deadline d_i, are executed on two processors. Let $t_1 = 4$, $t_2 = 4$, and $t_3 = 3$. Also assume that $d_i = 6$, for all i. If no parallel execution is allowed, it is easy to see that no feasible schedule is possible for the jobs. However, Figure 2.1 shows that there is processor idle time before the deadline at time 6. In fact, if job J_3 could be executed on both processors simultaneously, all deadlines can be met as shown in Figure 2.2.

Suppose that the time needed to execute job J_3 on one processor is t_3. A reasonable amount of time needed to execute J_3 on two processors in parallel is $t_3/2$ + *overhead*. Some examples of the overhead are synchronization, communication, and strict serial nature of the program. A detailed discussion about possible overheads is outside of the scope of this paper. In Figure 2.2, the overhead is assumed to be zero. However, even if the overhead is 0.5 the above job set is still schedulable. □

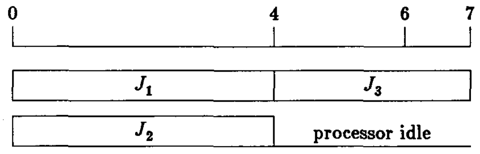


Figure 2.1.

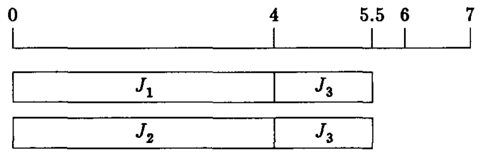


Figure 2.2.

Example 2.1 shows that the multiprocessor scheduling problem with parallel execution capability may achieve a better processor utilization. We define the scheduling problem with parallelizable jobs as follows.

The problem of scheduling parallelizable jobs is to find a feasible schedule for J = $\{J_1, J_2, ..., J_n\}$ on m processors. Job J_i has a deadline d_i and its *sequential* execution time is t_i if it is executed on one processor. Moreover, one or more of the jobs in J may be executed in parallel on multiple processors. In general, a job may be divided into many segments, each of which is composed of a different number of parallel tasks. The parallel execution time e_i of J_i is defined as follows:

$$e_i = \sum_{k=1}^{m} F_k(t_{ik}) \qquad \textbf{(2.1)}$$

where $F_k(t)$ is the parallel execution time function on k processors, t_{ik} is the portion of the sequential execution time of job J_i that is executed on k processors ($\sum_{k=1}^{m} t_{ik} = t_i$). □

A possible definition of F is

$$F_k(t) = t/k + Ov(k,t) \qquad (2.2)$$

where t is the execution time required if the segment is executed on a single processor and $Ov(k,t)$ is the overhead function. It is obvious that $F_1(t)=t$, that is, $Ov(1,t)=0$. In other words, no parallelization overhead occurs if a job is executed on a single processor.

An example of such a system is shown in Figure 2.3. Assume that the execution time of job J_i on a single processor is 28. Now it is divided into three segments: t_{i1} = 10 of J_i is executed on one processor, t_{i2} = 6 of J_i is executed on two processors, and t_{i3} = 12 of J_i is executed on three processors. Assume that the overhead is always zero, the total execution time for J_i will be 10 + 6/2 + 12/3 = 17.

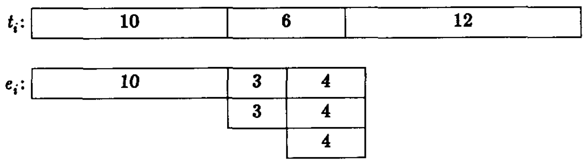


Figure 2.3.

In this paper, we assume that systems use a preemptive priority-driven scheduling approach. A priority-driven scheduling always selects the highest priority job among all ready jobs to execute next. Using such a policy, each job is assigned a priority before its execution. A preemptive scheduling allows a high priority job to preempt a low priority active job whenever the high priority job becomes ready. The scheduling algorithms that we are interested must meet the following rules:

(1) Before a job may be executed, all ready but unfinished jobs with higher priorities must be running on at least one processor;

(2) Processors may not all be idle if there is at least one ready job;

(3) A job may be executed on any number of processors as long as they are free.

Consequently, in addition to assigning a priority to each job, the scheduling algorithm must decide the number of processors to be allocated to a job during its execution.

A related work on parallel task scheduling was done by Du and Leung [5]. In their work, a job can be executed by 1, 2, ..., m processors. The execution time of job J_i on k processors is defined by a function $\tau(J_i,k)$. However, in their model, the number of processors assigned to a job is predetermined and not allowed to change during the execution of the job. Moreover, no deadline is given to any job and the goal is to find the shortest overall schedule length for a set of jobs with or without precedence constraints. The problem with precedence constraints has been shown to be strongly NP–hard for $m \geq 2$. The problem without precedence constraints can be solved in pseudo–polynomial time for $m=2$ and 3. For $m \geq 5$, it is again strongly NP–hard.

3. Simple Scheduling Problems for Parallelizable Jobs

In this section, we discuss several simple scheduling problems for real-time systems with parallelizable jobs under varying sets of assumptions. For job J_i, we define r_i to be its ready time, d_i its deadline, and t_i its required sequential execution time.

Problem 1. All jobs in J $= \{J_1, J_2, ..., J_n\}$ have the same ready time $r_i = 0$. Without loss of generality, assume that their deadlines are ordered $d_1 \leq d_2 \leq \cdots \leq d_n$. Jobs are totally independent, i.e. there is no precedence constraint and jobs do not share any resource except processors. All jobs can be parallelized with no overhead, that is, $Ov(k,t) = 0$ for all k and t.

For this problem the ***earliest-deadline-first*** (EDF) algorithm [9] with all jobs parallelized on all m processors is optimal. The Parallel EDF (PEDF) algorithm always executes the job with the earliest deadline on all processors when it becomes ready. To see that PEDF is optimal, suppose J cannot be feasibly scheduled using PEDF. There exists at least one job which cannot meet its deadline. Let us choose the first ***tardy*** job (a ***tardy*** job is a job that does not meet its deadline), say J_k. Then we have $\sum_{i=1}^{k} t_i/m > d_k$, or $\sum_{i=1}^{k} t_i > m \cdot d_k$. In other words, the total processor time required before d_k is more than m processors can provide. There is no way to schedule $J_1, J_2, ..., J_k$ before time d_k. Hence, the PEDF algorithm must be optimal.

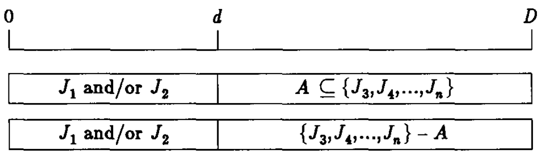
Although we are interested in preemptive scheduling, the job executions in this problem will be exactly the same as those using non-preemptive scheduling. This is because all jobs have the same ready time and they are independent. With these assumptions, a job is executed only when it has the highest priority among all remaining jobs and thus will not be preempted. The non-preemptive multiproessor scheduling problem has been shown to be NP–complete if no parallel execution is allowed [6]. With the flexibility of parallel execution, jobs can be easily scheduled. Also, the schedule length $\sum_{i=1}^{n} t_i/m$ is minimum. In fact, the above problem is the same as the single processor problem with $e_i = t_i/m$ for all jobs.

Problem 2. All jobs have the same ready time but different deadlines as in Problem 1, but only a proper subset of the jobs can be parallelized.

Although the problem is only slightly different from Problem 1, it is NP–complete. The reason is that for those non-parallelizable jobs, finding a feasible schedule is still intractable.

Theorem 3.1. Problem 2 is NP–complete.

The NP–completeness of Problem 2 can be seen from the following example. A job set J $= \{J_1, J_2, ..., J_n\}$ is to be run on two processors with only jobs J_1 and J_2 being parallelizable. Let $t_1 = t_2 = d_1 = d_2 = d$, and $d_i = D > d$, for $i = 3, 4, ..., n$. Since both J_1 and J_2 have execution times equal to their deadlines, J_1 and J_2 must be executed within time intervial $[0,d]$ and all the other jobs must be executed within the time interval $[d,D]$ (Figure 3.1). To schedule jobs $J_3, J_4, ..., J_n$ on two processors in $[d,D]$ without parallelization is the same as the traditional multiprocessor non-preemptive scheduling problem with one system deadline, which is known to be NP–complete [6] (transformed from the PARTITION problem). We do not show the formal proof in this paper which can be easily constructed.



0
d
D
J_1 and/or J_2
$A \subseteq \{J_3, J_4, ..., J_n\}$
J_1 and/or J_2
$\{J_3, J_4, ..., J_n\} - A$

Figure 3.1.

Problem 3. All jobs have the same ready time and deadline D. Also, they are totally independent. All jobs are parallelizable with constant overhead, that is, $Ov(k,t) = C$, where $C > 0$.

Since the overheads are non-zero the optimal way to schedule a job set might be to execute each job on a single processor so that no overhead will be introduced. But to test whether a job set can be scheduled on m processors without parallelization is an NP–complete problem. Therefore, this problem is NP–complete.

Theorem 3.2. Problem 3 is NP–complete.

Again we do not present the formal proof in this paper. In fact, the problem is NP–complete even on only two processors with $F_2(t) = t/2 + C$. The necessary condition for the set of jobs to be schedulable is:

$$\sum_{i=1}^{n} t_i \leq 2D \tag{3.1}$$

but a sufficient condition for a feasible schedule is:

$$\sum_{i=1}^{n} t_i + 2C \leq 2D \tag{3.2}$$

We now show that equation (3.2) is a sufficient condition. Let us first consider scheduling with no parallel execution capability. For any fixed priority assignment, suppose the last job scheduled on processor 1 is J_1 and on processor 2 is J_2, with finish times f_1 and f_2, respectively. Without loss of generality, assume that $f_1 < f_2$. The inequality $(f_2 - f_1) \leq t_2$, where t_2 is the execution time of job J_2, must always hold (Figure 3.2). This is because if $f_1 < (f_2 - t_2)$, then we will schedule J_2 on processor 1, instead of processor 2, and have an earlier overall finish time.

Now, instead of executing J_2 only on processor 2, we can parallelize the part of J_2 that runs beyond f_1, and execute them on both processors (Figure 3.3). In this way, only one job needs to carry the overhead. Thus, equation (3.2) is a sufficient condition for the job set to be schedulable.

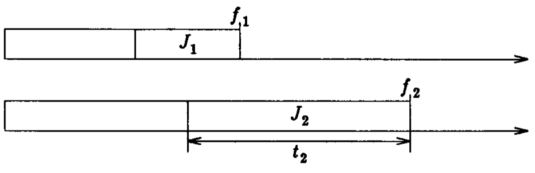


Figure 3.2.

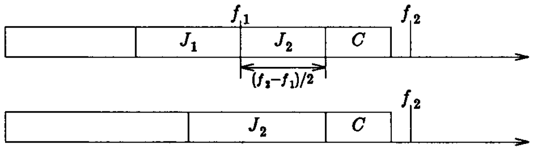


Figure 3.3.

4. Scheduling Periodic Parallelizable Jobs

Problem 3 shows that if the overhead function is not equal to zero, the scheduling problem is intractable. In the following discussion, we shall assume that the overhead function is always zero. We investigate real-time systems with periodic jobs that are parallelizable. Two classes of scheduling policies are investigated; they are referred to as *fixed* and *dynamic* priority assignment in [9]. For job J_i, we define P_i to be its period and U_i its utilization factor ($U_i=t_i/P_i$). We also define U to be the total utilization factor for the entire system, that is $U=U_1+U_2+\cdots+U_n$.

Problem 4. Job priorities are assigned dynamically according to some run-time characteristics (Dynamic Priority Scheduling).

The PEDF algorithm with all jobs parallelized on all processors is optimal for both *synchronous* and *asynchronous* systems [7] with $d_i=P_i$. A periodic job system is called *synchronous* if all jobs start their first period at the same time; otherwise, it is *asynchronous*. The optimality of PEDF can be proved in both cases, and will be presented in a future paper.

Problem 5. Jobs have predetermined priorities which are fixed throughout their executions (Fixed Priority Scheduling).

Unfortunately, Rate–Monotonic Algorithm (RMA) [9] with all jobs executed on all processors is not optimal for this case. The RMA assigns job priorities according to their periods: the shorter the period, the higher the priority is assigned to a job. For example, a system with the jobs (P_i,t_i) = (5,2.5), (5,2.5), (6,6) will have a priority assignment of $J_1=J_2>J_3$. However, this priority assignment will not produce a feasible schedule with all jobs executed on 2 processors (Figure 4.1). The job set is schedulable with the priority assignment $J_3 > J_1 > J_2$ and each job executed on one processor (Figure 4.2).

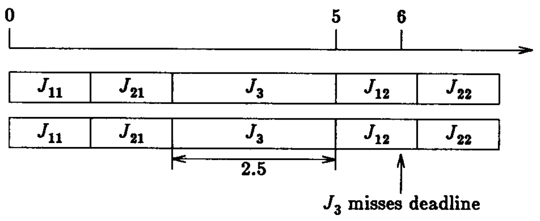


Figure 4.1.

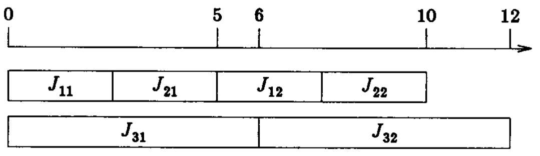


Figure 4.2.

Before proceeding, let us take a closer look at the multiprocessor scheduling problem with fixed priority assignment. We observe that no fixed priority algorithm is optimal if the priority assignment is based solely on the period P_i or the execution time t_i of jobs in a system. This can be easily seen from the following example.

Example 4.1. There are three sets of jobs to be scheduled on two processors.

Set 1: (P_i,t_i) = (5, 2.5), (5, 2.5), (6, 6)

Set 2: (P_i,t_i) = (5, 5), (5, 0.5), (6, 5)

Set 3: (P_i,t_i) = (2.5, 2.5), (5, 2.5), (13.5, 6)

From earlier discussion, we know that the only feasible priority assignments for the first set are $(J_3>J_1>J_2)$, $(J_3>J_2>J_1)$, $(J_1>J_3>J_2)$, and $(J_2>J_3>J_1)$. Actually, there are only two variations since J_1 and J_2 are interchangeable.

Jobs in the second set have the same periods as those jobs in the first set. But the four priority assignments are no longer feasible to the system. The only feasible priority assignment is $(J_1>J_2>J_3)$, or $(J_2>J_1>J_3)$. Therefore, any algorithm assigning priority based solely on the periods is not optimal.

Jobs in the third set have the same execution times as those jobs in the first set. The only feasible priority assignment is $(J_1>J_2>J_3)$, or $(J_2>J_1>J_3)$. Therefore, assigning priority based solely on the execution times will not be the optimal algorithm. □

Since a set of periodic jobs can be uniquely identified by { (P_i,t_i) }, an interesting question to ask is whether there is an optimal polynomial-time scheduling algorithm whose priority assignment is based on some simple function $f(P_i,t_i)$. For example, two such simple functions are:

(1) $f(P_i, t_i) = t_i/P_i$ (utilization)

(2) $f(P_i, t_i) = P_i - t_i$ (slack)

Unfortunately, these two functions do not always produce a feasible priority assignment when one exists. In Set 2 of Example 4.1, $U_1 > U_3 > U_2$. But the only feasible priority assignments are: $J_1 > J_2 > J_3$ and $J_2 > J_1 > J_3$. We may conclude that algorithms based only on *utilization* are not optimal. A similar example shows that algorithms based only on *slack* are not optimal.

We suspect that the difficulty of multiprocessor scheduling problem (with or without parallel execution capability) is in the priority assignment. Even with the flexibility of parallel execution, the problem is still intractable. The following theorem shows the NP–hardness of the problem.

Theorem 4.1. The problem of deciding whether there exists a fixed–priority assignment for a periodic job system such that the jobs are schedulable on two processors with parallel execution capability is NP–hard.

Proof: We reduce the PARTITION problem to our problem. The PARTITION problem is defined as follows:

> Given a set of numbers $A = \{a_1, a_2, ..., a_n\}$, is there a subset A' of A such that $\sum_{a \in A'} a = \sum_{a \in A-A'} a = S/2$, where $S = \sum_{a \in A} a$?

We assume that all $a_i < S/2$, otherwise the problem is trivial. Given an instance of the PARTITION problem, we create an instance of our *schedulability* problem:

> A system has jobs $J = B_1 \bigcup B_2$, where $B_1 = \{J_1, J_2, ..., J_n\}$ and $B_2 = \{J_{n+1}, J_{n+2}\}$. The execution time of J_i is a_i in the PARTITION problem, for $i \leq n$. The execution times of J_{n+1} and J_{n+2} are $5S/4$ and $S/4$, respectively. J_{n+1} and J_{n+2} have the same period $3S/2$, while the other J_i's all have the same period S.

We want to show that A' for the PARTITION problem exists if and only if the jobs are schedulable on two processors. We first show the if part: if the job set is schedulable then the partition exists.

We only need to consider the time interval which is the least common multiple (LCM) of all periods. In our case, LCM is $3S$. For ease of discussion, the LCM is divided into 4 regions (Figure 4.3): $R_1 = [0, S]$, $R_2 = [S, 3S/2]$, $R_3 = [3S/2, 2S]$ and $R_4 = [2S, 3S]$. Assume that J is schedulable on two processors, we can show the following lemmas.

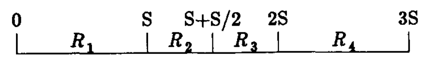


Figure 4.3.

Lemma 4.1. Jobs in B_2 will consume exactly $S/2$ unit of processor time in R_2.

Proof: There can be no processor idle time in the schedule since the total utilization U is 2. Since all jobs in B_1 have the deadline S, all B_1 jobs must be scheduled in the interval R_1. They will require exactly S units of processor time. Therefore, another S units of processor time in R_1 will be available and used by jobs of B_2. The rest of the required execution time for B_2, $S/2$, must be allocated from R_2. □

In the following, we use $J_{i,j}$ for the j–th period task of J_i.

Lemma 4.2. $J_{n+2,1}$ must be finished in R_1.

Proof: Assume that part of (or all of) $J_{n+2,1}$, $J'_{n+2,1}$, and part of $J_{n+1,1}$, $J'_{n+1,1}$, are executed in R_2. J_{n+1} must have been started earlier than J_{n+2} since only $S/2$ unit of processor time in R_2 is available to both jobs (Lemma 4.1). Moreover, all jobs in B_1 must have finished when J_{n+2} is started, otherwise they will miss their deadline S. This means that J_{n+2} is the lowest priority job among all jobs in J since it has the latest starting time. Therefore, before $J'_{n+2,1}$ can be executed in R_2, $J'_{n+1,1}$ and all the second period tasks in B_1 must be started. Since all execution times of the jobs in B_1 are less than $S/2$ and the total execution time is equal to S, there will not be enough time for $J'_{n+2,1}$ to finish before its deadline $(3/2)S$. This is because if both $J_{n+1,1}$ and $J_{n+2,1}$ are finished before time $(3/2)S$ then at least one of the jobs in B_1 must be started after time $(3/2)S$. This contradicts the fact that $J'_{n+2,1}$ is the lowest priority job. □

Lemma 4.3. J_{n+1} cannot be the highest priority job, nor can it be the lowest priority job in J.

Proof: From Lemmas 4.1 and 4.2, we know that $J_{n+1,1}$ has exactly $S/2$ processor time in R_2. If J_{n+1} is the highest priority job it must be started before all other jobs at time 0. It will need at most $S/4$ in R_2. Thus J_{n+1} cannot be the highest priority job. If J_{n+1} is the lowest priority job, all second period tasks of B_1 must be started before $J_{n+1,1}$ can continue in R_2. $J_{n+1,1}$ will not have $S/2$ processor time available in R_2. □

In the following, we define B'_1 to be the jobs in B_1 which have higher priorities than J_{n+1}, and B''_1 to be the rest of jobs in B_1.

Lemma 4.4. The total execution time of jobs in B'_1 must be less than or equal to $S/2$.

Proof: Note that $J'_{n+1,1}$ and the second period tasks of all jobs in B'_1 must be scheduled in R_2. Jobs in B'_1 can be finished in R_3. If the total execution time of jobs in B'_1 is larger than $S/2$ then at least one of the jobs in B'_1 must be continued in R_3, that means its start time will be later than the start time of J'_{n+1} (since the execution times of all B_1 jobs are less than $S/2$). This contradicts the fact that all the jobs in B'_1 have priorities higher than that of J_{n+1}, and their start times must be earlier than that of J'_{n+1} (Figure 4.4). □

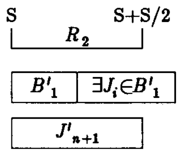


Figure 4.4.

Lemma 4.5. J_{n+2} has the lowest priority among all jobs.

Proof: If not, $J_{n+2,2}$ must be executed in time slot R_3 (since both J_{n+1} and J_{n+2} have priorities higher than some of the jobs in B''_1). Also, since J_{n+1} has priority higher than those of the jobs in B''_1 and the total execution time of jobs in B''_1 is $\geq S/2$ (Lemma 4.4), if J_{n+2} has priority higher than that of J_{n+1} then at least $S/2 - (S/4)/2 = (3/8)S$ of $J_{n+1,2}$ must be executed in R_3. In this case, the time left for the second period tasks in B_1 in time slots R_2 and R_3 is $\leq 2S - S/2 - S/4 - (3/8)S = (7/8)S < S$. If J_{n+2} has priority lower than that of J_{n+1} then at least $S/2$ of $J_{n+1,2}$ must be executed in R_3 (Figure 4.5). Again, the time left for the second period tasks in B_1 in time slots R_2 and R_3 is $\leq 2S - S/2 - S/4 - S/2 = (3/4)S < S$. In both cases, at least one of B_1 will miss its deadline. This contradicts with our assumption that a feasible schedule exists. □

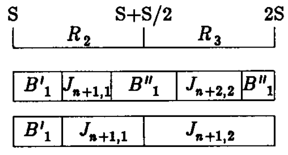


Figure 4.5.

Lemma 4.6. The total execution time of jobs in B'_1 must be larger than or equal to $S/2$.

Proof: From Lemma 4.5, we know that only the jobs in B'_1 have priorities higher than that of J_{n+1}. And, since the start time of J_{n+1} cannot be later than half of the total execution time of jobs in B'_1, if the total execution time of jobs in B'_1 is less than $S/2$, then the start time of J_{n+1} will be earlier than $S/4$ and this means that J_{n+1} will not require $S/2$ time in R_2, contradicting Lemmas 4.1 and 4.2. □

From Lemmas 4.4 and 4.6, we can conclude that the total execution time of jobs in B'_1 is exactly $S/2$. Therefore, A' does exist for the PARTITION problem if J is schedulable on two processors.

We now show the only if part: if the partition A' exists for the PARTITION problem, then there is a feasible schedule for J.

For each i such that $a_i \in A'$, we assign to J_i a priority higher than that of J_{n+1}. For each i such that $a_i \in A''$ ($A'' = A - A'$), we assign to J_i a priority lower than that of J_{n+1}. We assign to job J_{n+2} a priority lower than all others. Then, a feasible schedule for J is shown in Figure 4.6.

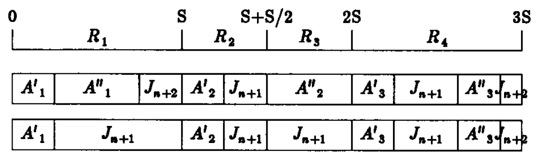


Figure 4.6.

Since the PARTITION problem is NP–complete, by the above reduction (note that the construction of the reduction can be carried out in polynomial time) we have shown that our schedulability problem is NP–hard [1,6]. □

Corollary. In a real–time system with n periodic jobs, the problem of deciding whether there exists a fixed–priority assignment such that all jobs are schedulable on m processors with parallel execution capability is NP–hard for $m > 2$. □

The corollary can be proved as in Theorem 4.1 by introducing m – 2 additional jobs all of which have execution times and periods equal to 3S. The proof is not presented in this paper due to space constraint.

5. A Heuristic Solution

In the following, we discuss a heuristic algorithm for the PARTITION problem which may be used to solve some of the problems discussed earlier. Consider the following problem.

Given a number set $A = \{a_1, a_2, ..., a_n\}$, find a subset A' of A to minimize

$$\left| \sum_{a \in A'} a - \sum_{a \in A - A'} a \right|$$

This is the minimization version of the PARTITION problem [6]. It can also phrased in terms of the following multiprocessor scheduling problem:

Given a job set of n jobs with execution times $t_i = a_i$, and ready times $r_i = 0$, for all i, schedule this job set on two processors so as to minimize the overall finish time.

Since the PARTITION problem is NP–complete, it is unlikely to find an exact solution for the problem in polynomial time. However, in some applications a sub–optimal solution may be satisfactory. We propose a heuristic algorithm that runs in polynomial time and produces an acceptable result.

Algorithm 5.1. Heuristic algorithm for partitioning.

```
Step 1 (sorting):
    Sort $a_i$ and reindex them s.t. $a_1 \geq a_2 \geq \cdots \geq a_n$.
Step 2 (initial partition):
    $A' = A'' = \emptyset$;
    for $k = 1$ to $n$ do
        if $\sum_{a \in A'} a \leq \sum_{a \in A''} a$ then $A' = A' \cup \{a_k\}$ else $A'' = A'' \cup \{a_k\}$;
Step 3 (refinement):
    In the following loop, let $A' = \{b_1, b_2, \ldots, b_{n_1}\}$, $A'' = \{c_1, c_2, \ldots, c_{n_2}\}$.

    $\Delta = \sum_{A'} b_i - \sum_{A''} c_i$;
    while there exists $i,j$, s.t. $\Delta > (b_i - c_j) > 0$ or $\Delta < (b_i - c_j) < 0$ do
        begin
            find $b_k$ and $c_l$ with $|2(b_k - c_l) - \Delta| = \min_{i,j} |2(b_i - c_j) - \Delta|$;
            $\Delta = \Delta - 2(b_k - c_l)$;
            exchange $b_k$ with $c_l$;
        end.
```

The idea behind this heuristic algorithm is the concept of *imprecise computation* [8]. The optimal solution for the PARTITION problem can be found by algorithms which need $O(2^n)$ time. But since

$$2^n = C(n,0) + C(n,1) + \cdots + C(n,n)$$
$$= 1 + O(n) + O(n^2) + \cdots + O(n) + 1$$

we can combine several polynomial time algorithms to approximate an exponential time algorithm. In Algorithm 5.1, only exchanges between two numbers of different partitions are considered. However, Algorithm 5.1 could have also tried to exchange two numbers in one partition with one number in another partition, or three numbers in one partition with one number in another partition, etc., which may be necessary to produce the optimal solution. The result produced by Algorithm 5.1 is usually nonoptimal, or *imprecise*. If more precise result is desired, we can conduct more comparisons (between a number from one partition and the sum of more than one number from the other partition) and have more exchanges to make $|\Delta|$ smaller.

One concern about Algorithm 5.1 is whether it will terminate. The answer is yes since $|\Delta|$ monotonically decreases in each iteration of Step 3. Another question is how many exchanges (iterations) are needed in Step 3. Theorem 5.1 answers this question.

Lemma 5.1. Once a number is exchanged to a new partition, it will not be exchanged back to the original partition.

Proof: Let $A' = \{b_1, b_2, \ldots, b_{n_1}\}$ and $A'' = \{c_1, c_2, \ldots, c_{n_2}\}$ be the original partitions. Assume that b_i has been exchanged with c_k earlier. So b_i is now in A'' while c_k is in A'.

If b_i is to be exchanged back to partition A', there are two possibilities:

(1) b_i is to be exchanged with c_l which is now in A'. c_l must have been exchanged with b_j earlier when $|\Delta|$ was reduced from D_1 to D_2. Since $|\Delta|$ monotonically decreases, if b_i is to be exchanged with c_l, the exchange must change $|\Delta|$ to a new value D_3 which is less than D_2. However, when b_j was exchanged with c_l, Algorithm 5.1 could have exchanged b_j with c_k and reduce $|\Delta|$ to D_3 directly. Since b_j was not exchanged with c_k, we know that D_3 is not less than D_2. We have a contradiction. So b_i cannot be exchanged with any c_l now in A'.

(2) b_i is to be exchanged with b_j in A'. Again, the new $|\Delta|$ after such an exchange must have a smaller value than that of after exchanging b_i and c_k. In other words, we would have exchanged b_j and c_k instead of b_i and c_k earlier. A contradiction shows that b_i cannot be exchanged with any b_j in A'.

From (1) and (2), we know that b_i cannot be exchanged back to A' once it is moved to A''. □

Theorem 5.1. The number of iterations in Step 3 of Algorithm 5.1 is $O(n)$.

Proof: By Lemma 5.1 we know that the number of iterations is less than or equal to $\min(n_1, n_2)$, the smaller of the numbers of elements in two initial partitions. In the worst case the number of iteration is $O(n/2) = O(n)$. □

The following example shows that the algorithm will require more than one exchange.

Example 5.1. $A = \{200, 194, 102, 100, 11, 10, 9\}$. After Step 2 $A' = \{200, 100, 10\}$ and $A'' = \{194, 102, 11, 9\}$. Step 3 will exchange pairs (102, 100) and (11, 10) (in that order). □

Actually, Step 1 (*sorting*) is not needed if Step 3 is executed until no more exchange is possible. In fact, sometimes the result is better if Step 1 is skipped.

Example 5.2. Given the sequence of a_i is (3, 52, 63, 99, 15, 39, 94, 83, 8, 89, 51, 55, 39, 79, 78, 21), Algorithm 5.1 without sorting will achieve a even partitioning, but will not if sorting is conducted first.

Without sorting, the following example shows that $O(n)$ exchanges are needed. Therefore, the $O(n)$ upper bound is tight for non-sorting case.

Example 5.3.
$A = \{l, l/2, k, k, \ldots, k, l/2 + k(k-1)/2, k-1, k-2, \ldots, 1\}$, $l \gg k$ and there are $(k-1)$ k's in the set A. After Step 2 $A' = \{l, k-1, k-2, \ldots, 1\}$ and $A'' = \{l/2, k, k, \ldots, k, l/2 + k(k-1)/2\}$. Then, Step 3 will exchange pairs $(1, k)$, $(2, k)$, ..., $(k-1, k)$ (in that

order). There are $O(k) = O(n/2) = O(n)$ exchanges. □

We have run some simulations to see how many exchanges are usually required using the algorithm. Two sets of simulations were conducted: one with manually arranged data sets and the other data sets were generated randomly. In each set, we executed the algorithm twice: once with the sorting step and the other without sorting. Tables I and II show the number of exchanges needed in each case and the value of Δ in final partitions. As can be seen from the tables, with sorting, we need only one or two exchanges to have a perfect partitioning even for 500 numbers.

Table I. Simulation with arranged data.

Data set	#data	Data range	Δ	#exchanges
With sorting				
1	21	[1,1000000]	0	0
2	7	[1,10]	0	0
3	13	[10,99]	0	1
4	16	[1,99]	2	1
5	7	[1,200]	0	2
No sorting				
1	21	[1,1000000]	0	9
2	7	[1,10]	0	1
3	13	[10,99]	0	1
4	16	[1,99]	0	1
5	7	[1,200]	0	1

Table II. Simulation with random data.

Data set	#data	Data range	Δ	#exchanges
With sorting				
1	100	[1,999]	0	1
2	100	[1,999]	0	0
3	100	[10,999]	0	1
4	100	[10,999]	0	1
5	100	[10,99]	0	1
6	300	[1,999]	0	1
7	300	[1,999]	0	1
8	300	[10,999]	0	1
9	300	[10,999]	0	0
10	500	[1,999]	0	1
No sorting				
1	100	[1,999]	0	1
2	100	[1,999]	0	3
3	100	[10,999]	0	7
4	100	[10,999]	0	2
5	100	[10,99]	0	2
6	300	[1,999]	0	1
7	300	[1,999]	0	11
8	300	[10,999]	0	12
9	300	[10,999]	0	4
10	500	[1,999]	0	9

We can use Algorithm 5.1 to find a sub-optimal solution for some of the multiprocessor scheduling problems discussed earlier. For example, given a set of periodic jobs with total utilization factor ≤ 2, and we want to execute it on two processors using the EDF algorithm. The job set is schedulable if we can partition the job set into two subsets such that each has a utilization factor smaller than 1. The heuristic algorithm may be used to produce a vague partition such that both A' and A'' have utilization factors close to one. If both are smaller than one, we have a feasible schedule. If one utilization factor is greater than one, we can parallelize one of the jobs in the partition to move part of the work to the other partition.

6. Conclusions

We have proposed a model of job scheduling with parallelizable jobs. Traditional scheduling problems assume that the execution time of a job is always fixed, and the number of processors required is not dynamic. In our model, we suggest that schedulers may determine the degree of parallelism for a job execution. In this way, jobs with high priorities can use more than one processor to reduce response time. With recent progress in multiprocessor technology and parallel computing, programs with such models can be easily implemented. Although the problem of static priority assignment using this model is NP-hard, heiristic algorithms with good performance can be found.

However, even if an optimal priority assignment is available, the scheduling algorithm still must decide the number of processors to be used by a job during its execution. Deciding an optimal processor assignment is another difficult problem that must be solved in order to completely handle the parallelizable job scheduling problem.

Acknowledgement

The authors wish to thank Jane Liu for her many insightful and detailed comments. We would also like to thank Phil McKinley and Wei-Kuan Shih for their careful reading on our earlier draft.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] J. Blazewicz, M. Drabowski, and J. Weglarz , "Scheduling multiprocessor tasks to minimize schedule length," *IEEE Trans. on Computers*, Vol. C-35, No. 5, May 1986.

[3] J. Blazewicz, J. Weglarz, and M. Drabowski, "Scheduling independent 2-Processor tasks to minimize schedule length," *Inform. Processing Letter*, Vol. 18, 1984.

[4] Z. Cvetanovic, "The effects of problem partitioning, allocation, and granularity on the performance of multiple-processor systems," *IEEE Trans. on Computers*, Vol. C-36, No.4, April 1987.

[5] J. Du and J. Y.-T. Leung, "Complexity of scheduling parallel task systems," Technical Report UTDCS 6-87, Uinv. Texas at Dallas, 1987.

[6] M.R. Garey and D.S. Johnson, *Computer and Intractability: A Guide to the Theory of NP–completeness,* Freeman, San Francisco, CA, 1979.

[7] J. Y.–T. Leung and J. Whitehead, "On the complexity of fixed–priority scheduling of periodic, real–time tasks," *Performance Evaluation,* 2, 1982.

[8] K.J. Lin, S. Natarajan, and J. W.–S. Liu, "Imprecise result: Utilizing partial computations in real–time systems," *RTSS 87,* San Jose, CA, Dec. 1987.

[9] C.L. Liu and J.M. Layland, "Scheduling algorithms for multiprogramming in a hard–real–time environment," *J. ACM,* Vol. 20, No. 1, Jan. 1973.

[10] J.D. Ullman, *Computational Aspects of VLSI,* CSP, 1984