**LETTER**

**Optimal Scheduling for Real-Time Parallel Tasks**

Wan Yeon LEE∗† and Heejo LEE††, Members

**SUMMARY** We propose an optimal algorithm for the real-time scheduling of parallel tasks on multiprocessors, where the tasks have the properties of flexible preemption, linear speedup, bounded parallelism, and arbitrary deadline. The proposed algorithm is optimal in the sense that it always finds out a feasible schedule if one exists. Furthermore, the algorithm delivers the best schedule consuming the fewest processors among feasible schedules. In this letter, we prove the optimality of the proposed algorithm. Also, we show that the time complexity of the algorithm is \(O(M^2 \cdot N^2)\) in the worst case, where \(M\) and \(N\) are the number of tasks and the number of processors, respectively.

**key words:** optimal algorithm, real-time scheduling, feasible schedule, bounded parallelism

1. **Introduction**

Multiple processors can be allocated for the execution of a single real-time task [1]. Examples of real-time systems making use of such parallel tasks include GISs (geographic information systems), flight simulators, particle simulators and systems dealing with atmospheric chemistry. Many studies have been conducted on the subject of scheduling real-time parallel tasks on multiprocessor systems [2]–[7]. However, most of these studies [2]–[5] used heuristic scheduling approaches, mainly due to the heavy complexity of the optimal scheduling approach.

In this letter, we propose an algorithm designed to find a feasible schedule of parallel tasks in real-time systems. A feasible schedule guarantees that all tasks complete their execution before their respective deadlines by making use of processors available in the system. The proposed algorithm can always find a feasible schedule of real-time parallel tasks with the principle of consuming as few processors as possible. The proposed algorithm is referred to as optimal in the sense that it always finds a feasible schedule consuming the fewest processors among feasible schedules. If the algorithm cannot find any feasible schedule, it implies that there is no way to guarantee the deadlines of all tasks using the given processors. The parallel tasks considered in this study have the properties of flexible preemption, linear speedup, bounded parallelism, and arbitrary deadlines. In flexible preemption, it is allowed to suspend a task and restart the task with a different number of processors without incurring any additional costs. In linear speedup, the speedup is linearly proportional to the number of allocated processors. In bounded parallelism (or parallelism bound), the speedup of parallel tasks can be maintained only up to some bounded number of processors [8], [9].

Drozdowski [6] and Burchard et al. [7] studied a similar scheduling problem but they assumed more severe constraints. Drozdowski’s algorithm [6] works for parallel tasks with linear speedup, flexible preemption, bounded parallelism, and arbitrary release constraints, but not for the tasks with arbitrary deadlines. An extension of Drozdowski’s algorithm can solve our problem [6], however, it is applicable only to parallel tasks with continuous-time execution but not applicable to those with discrete-time execution. Contrarily, our algorithm works for parallel tasks with discrete-time execution. Burchard’s algorithm [7] works for non-parallel tasks but not for parallel tasks. Hence, our algorithm can be more practicable than the previous algorithms.

The proposed algorithm can be utilized in low-power multiprocessor systems [10]. In the dynamic power management (DPM), unused components are turned off to reduce the power consumption. Whenever a set of real-time tasks are given, our algorithm reserves the minimum number of powered-on processors, even though there are more available processors. As well, the proposed algorithm is useful for on-line systems with a fixed number of processors [11]. When the task arrives dynamically, the scheduler is invoked to decide whether the new task can be scheduled, along with the old tasks which were previously accepted, so that the deadlines of all tasks are satisfied. Only when the deadlines of all previous tasks are satisfied along with the new task, the task is accepted. Then our algorithm can be utilized to estimate the upper bound of the acceptance ratio of real-time tasks in the on-line system.

In this letter, we deal with the problem of scheduling a set of \(M\) tasks on \(N\) identical processors. To formulate the problem, processor \(n\) is denoted as \(P_n\) and task \(m\) is denoted as \(T_m\). The deadline of \(T_m\) is denoted as \(d_m\), the total amount of computation of \(T_m\) to be executed before \(d_m\) is denoted as \(c_m\), and the parallelism bound of \(T_m\) is denoted as \(b_m\). Then, the parallel execution time of \(T_m\) with \(c_m\) on \(n\) processors is \([c_m/n]\) where \(n \leq b_m\), and it is assumed that \([c_m/b_m]\) \(\leq d_m\). This letter is organized as follows: In Sect. 2, we describe the proposed algorithm with at most \(O(M^2 \cdot N^2)\) steps. In
Sect. 3, we prove its optimality and conclude this paper in Sect. 4.

2. Proposed Algorithm

The proposed algorithm uses the Earliest Deadline First (EDF) rule when determining the scheduling order of tasks. Tasks are sorted in increasing order of their deadlines and stored in a list \( T = \{T_1, T_2, \ldots, T_M\} \). For each task, the algorithm should determine the starting time, the suspending time, and the number of processors used for its execution whenever it starts or restarts. When several processors are available for the execution of a task, the algorithm first allocates the processor with the smallest index to the task. The algorithm prefers to use the processors with a smaller index, which is similar to the one-dimensional packing algorithm using the minimum bins [12], [13]. Only after fully utilizing \( P_n \) from the time when it becomes available up to the deadline, does the algorithm use \( P_{n+1} \) to allocate the remaining computation. Then, the algorithm executes \( T_1, T_2, \ldots, T_M \) before their respective deadlines using the minimum number of processors for each \( m \). We refer to the proposed algorithm as Opt-Algorithm and the following notation is used for its formulation.

- \( \Theta \): the remaining amount of computation after the partial scheduling of each task
- \( \Phi \): the time upper bound after which the processor cannot be utilized for the execution of each task due to the deadline or the parallelism bound
- \( \eta_m \): the number of processors allocated for the execution of \( T_m \) at the time instant \( \tau^s \)
- \( \eta^s = \sum_{m=1}^{M} \eta_m \): the total number of processors allocated for the execution of \( T_1, T_2, \ldots, T_M \) at the time instant \( \tau^s \)

In order to describe Opt-Algorithm precisely, we use another algorithm described in Fig. 1, called Scheduling-Algorithm, which finds a feasible schedule using \( N \) processors for the tasks in \( T \). This Scheduling-Algorithm allows Opt-Algorithm to find the feasible schedule using the fewest processors, which is described in Fig. 2. Scheduling-Algorithm initializes the values of \( \eta \) and \( \eta_m \) in line 2 and schedules tasks one by one during the FOR loop in lines 3-20. For the scheduling of each task \( T_m \), the algorithm first tries to use \( P_1 \) to allocate the total computation of \( T_m \) during the WHILE loop in lines 5-19 (\( n = 1 \)). If the total computation is not allocated completely at the end of the WHILE loop, the algorithm next uses \( P_2 \) to allocate the remaining computation of \( T_m \) within the same WHILE loop from lines 5 to 19 (\( n = 2 \)). This procedure is repeated until the remaining computation is allocated completely (\( \Theta = 0 \)). In the WHILE loop, the algorithm searches for the start time \( \tau^s \) and the end time \( \tau^e \) of \( P_n \) in lines 6-7. \( \tau^s \) is the scheduled time for \( P_n \) to start the execution of \( T_m \). The time when \( P_n \) becomes available after executing other tasks is assigned to the variable \( \tau^s \). \( \tau^e \) is the time at which \( P_n \) is scheduled to finish the execution of \( T_m \). If \( P_n \) can completely allocate the remaining computation of \( T_m \), then (\( \Theta + \tau^s \)) is assigned to \( \tau^e \), otherwise \( \Phi \) is assigned to \( \tau^e \). After the time \( \Phi \), \( P_n \) cannot be used anymore for the execution of \( T_m \), because the deadline or the parallelism bound of \( T_m \) would be violated after this point. Because \( P_n \) is allocated for the execution of \( T_m \) from time \( \tau^s \) to time \( \tau^e \), the values of \( \eta_m \) and \( \eta \) during this period increase by one in lines 9-10. If this increment of \( \eta_m \) makes it equal to the value of \( b_m \) starting from some point, then the value of \( \Phi \) is updated in lines 11-13 by replacing it with the time value corresponding to this time point. The value denoting the remaining amount of computation after the reservation of \( P_n \) is updated in line 14. If \( n = N \) and \( \Theta > 0 \) in lines 15-17, the algorithm determines that scheduling \( T_m \) has failed, because all of the processors are occupied but there is still computation remaining to be done. Otherwise, the WHILE loop is performed again after increasing the value of \( n \) by one in line 18.

The values of \( \eta \) and \( \eta_m \) with regard to each \( \tau^s \) are maintained in linked lists, such as shown in Fig. 3. The values of \( \eta \) and \( \eta_m \) are recorded along with the time point, \( \tau^s \), in the lists, only when they are found to have changed. The elements in the lists consist of the number of allocated processors and the time point, and are sorted in increasing order of the time point, \( \tau^s \). These lists are initialized with an element, \([0, 0]\), in line 2 of Fig. 1. Then, the update of \( \eta_m \) along time \( \tau^s \) or the insertion of a new \( \eta_m \) in a linked list can be
performed in $O(M)$ steps, because each list stores the numbers of processors allocated to at most $M$ tasks. Thus, the operations in line 2 require $O(M)$ steps and the operations in lines 9-10 require $O(1)$ steps. The FOR loop in lines 3-20 performs $M$ iterations and the WHILE loop in lines 5-19 performs at most $N$ iterations. The sorting operation of $M$ tasks requires $O(M \cdot \log M)$ time complexity. Therefore, the total time complexity of Scheduling-Algorithm is $O(M^2 \cdot N)$ in the worst case.

Figure 4 shows a working example of Scheduling-Algorithm. The total computation of $T_1$ is first allocated to $P_1$ from time 0 to $d_1$, which is denoted as $t_{1}$. Next, the remaining computation is allocated to $P_2$ from time 0, which is denoted as $t_{2}$. The total computation of $T_2$ is first allocated to $P_1$ from time $d_1$ to $d_2$, which is denoted as $t_{3}$. Next the remaining computation of $T_2$ is allocated to $P_2$, denoted as $t_{2}$, and allocated to $P_3$ from time 0 to $d_1$, because $b_2 = 2$, which is denoted as $t_{4}$. Then, $T_3$ can be scheduled in a similar manner. In this example, Scheduling-Algorithm fails to find a feasible schedule. After $d_1$, the remaining amount of computation of $T_3$, denoted as $t_{5}$, cannot be assigned to $P_3$, $P_4$ or $P_5$, since assigning $t_{5}$ to these processors exceeds the parallelism bound of $T_3$.

Opt-Algorithm described in Fig. 2 increases the number of processors, $N'$, by one during the FOR loop in lines 2-6 until Scheduling-Algorithm finds a feasible schedule using $N'$ processors. When the algorithm finds a feasible schedule using $N'$ processors and stops its operation, $N'$ is the minimum number of processors to satisfy the deadlines of the tasks in $T$. Since there is always a feasible schedule when $N' = \sum_{i=1}^{M} b_i$, Opt-Algorithm must be finished even if $N \approx \infty$. The FOR loop requires at most $O(N)$ iterations ($N' \leq \min(N, \sum_{i=1}^{M} b_i)$) and thus the total time complexity of Opt-Algorithm is $O(M^2 \cdot N^2)$ in the worst case.

3. Proof of Optimality

We first show that there is no feasible schedule whenever Scheduling-Algorithm fails to find a feasible schedule. This means that Scheduling-Algorithm always finds a feasible schedule if there are feasible schedules to be found. Next, we prove the optimality of Opt-Algorithm by showing that there is no feasible schedule using fewer processors than the schedule found by Opt-Algorithm. This implies that Opt-Algorithm always finds a feasible schedule using the fewest processors.

**Lemma 1:** $\eta^x \geq \eta^y$ if $t^x < t^y$.

**Proof:** Scheduling-Algorithm prefers to use $P_n$ before using $P_{n+1}$ for any $n$. In order to schedule $T_1$, the algorithm first utilizes $P_1$ starting from the time when $P_1$ becomes available to the time $d_i$. After using $P_1$, the algorithm next tries to use $P_2$ starting from the time when $P_2$ becomes available. Similarly, after using $P_n$, the algorithm next utilizes $P_{n+1}$ starting from the time when $P_{n+1}$ becomes available, which is shown in Fig. 4. Thus, $\eta^x \geq \eta^y$ when $t^x < t^y$. \hfill $\Box$

**Lemma 2:** If $0 < \eta^x$ and $0 \leq \eta^y < b_i$ such that $t^x < t^y$ after Scheduling-Algorithm successfully completes the scheduling of $T_i$, then $\eta^x = \eta^y$ or $\eta^x = (\eta^y + 1)$ for each $i$.

**Proof:** Scheduling-Algorithm prefers to use $P_n$ before using $P_{n+1}$ for any $n$. Only after Scheduling-Algorithm has utilized $P_n$ fully, does it use the next processor $P_{n+1}$ to allocate the remaining computation of $T_i$. Therefore the algorithm uses each processor $P_n$ until time $t^x$, such that $t^x = d_i$, and $n$ is assigned to $\eta^x$ once the use of $P_n$ is completed. If there is no remaining computation, then $\eta^x = \eta^y$. Otherwise, $P_{n+1}$ executes the remaining computation up to some time point, $t^x$ ($t^x < t^y$). Then, $\eta^x = (\eta^y + 1)$. Hence, $\eta^x = \eta^y$ or $\eta^x = (\eta^y + 1)$ if $0 < \eta^x$ and $0 \leq \eta^y < b_i$ such that $t^x < t^y$ after Scheduling-Algorithm successfully completes the scheduling of $T_i$. This situation is illustrated in Fig. 5 (a), as an example of the scheduling of $T_i$. \hfill $\Box$

**Lemma 3:** If $\eta^x = N$, $\eta^y < N$, $\eta^x < b_k$, and $\eta^y = b_k$ such that $t^x < t^y$ when Scheduling-Algorithm fails to schedule $T_k$, then $\eta^y_i = b_i$ such that $\eta^y_i > 0$ and $i < k$ for each $i$.

**Proof:** Scheduling-Algorithm may fail to schedule $T_k$ after successfully scheduling $T_{k-1}$. Among many possible failure cases, we consider the failure case in which some two
time points $\tau^i$ and $\tau^j$ such that $\tau^i < \tau^j$ have the conditions of $\eta^i = N$, $\eta^j < N$, $\eta^i_k = b_k$, and $\eta^j_k = b_k$. In this case, let us assume that $\eta^i \leq \eta^j \leq (\eta^i + 1)$ after scheduling $T_{k-1}$ successfully (before starting to schedule $T_k$). If $\eta^i = \eta^j$ before scheduling $T_k$, then $\eta^i_k = \eta^j_k$ after scheduling $T_k$, because $P_{\eta^i+1}, P_{\eta^i+2}, \cdots$ and $P_N$ are used sequentially to schedule $T_k$ both at time $\tau^i$ and at time $\tau^j$. If $\eta^j = (\eta^i + 1)$ before scheduling $T_k$, then $\eta^i_k = (\eta^j_k - 1)$ after scheduling $T_k$ because $P_{\eta^j+1}$ is used to schedule $T_k$ at time $\tau^i$ but not at time $\tau^j$. $P_{\eta^j+2}, P_{\eta^j+3}, \cdots$ and $P_N$ are used sequentially to schedule $T_k$ both at time $\tau^i$ and at time $\tau^j$. Thus, $\eta^i_k$ or $\eta^j_k$ is not available processors to compensate for the additional processors required for the execution of $T_k$. In summary, the conditions at time $\tau^i$ are $\eta^i_k \leq b_k$, $\eta^j < N$, and $\tau^i < \tau^j$. Under these conditions, however, Scheduling-Algorithm does not fail to schedule $T_k$ at time $\tau^i$, because there are available processors and the deadline or the parallelism bound of $T_k$ is not violated. Hence, the assumption that $\eta^i \leq \eta^j \leq (\eta^i + 1)$ after successfully scheduling $T_{k-1}$ is a contradiction of the other assumption, namely that Scheduling-Algorithm fails to schedule $T_k$. Also, $\eta^i < \eta^j$ is not allowed by Lemma 1. Therefore, if $\eta^i = N$, $\eta^j < N$, $\eta^i_k < b_k$, and $\eta^j_k = b_k$ when Scheduling-Algorithm fails to schedule $T_k$, then $\eta^i_k \geq (\eta^j_k + 2)$ after scheduling $T_{k-1}$ successfully. Figure 5(b) shows this case.

By the contraposition of Lemma 2, if $\eta^i \geq (\eta^j + 2)$ after scheduling $T_k$, successfully, then $\eta^i_k = b_i$ or $\eta^j_k = 0$ such that $i < k$ for each $i$. Consequently, if $\eta^i = N$, $\eta^j < N$, $\eta^i_k < b_k$, and $\eta^j_k = b_k$ such that $\tau^i < \tau^j$ when Scheduling-Algorithm fails to schedule $T_k$, then $\eta^i_k = b_i$ such that $\eta^i_k > 0$ and $i < k$ for each $i$.

□

**Theorem 1:** There is no feasible schedule if Scheduling-Algorithm fails to find a feasible schedule on $N$ processors.

**Proof:** When all processors are occupied but there is still some computation of $T_k$ remaining to be scheduled, Scheduling-Algorithm fails to schedule $T_k$ at some time point $\tau^i$ ($\eta^i = N$). The failed conditions of Scheduling-Algorithm in line 11 of Fig. 1 are $\eta^i = N$ and $\tau^i = d_k$ or $\eta^i_k = b_k$. When Scheduling-Algorithm fails to schedule $T_k$ at time $\tau^i$, we assume that there is a feasible schedule which satisfies the deadlines of $T_1, T_2, \cdots, T_k$ simultaneously. We refer to this feasible schedule as New Schedule and the failed schedule of Scheduling-Algorithm as Original Schedule. Compared with the Original Schedule, the New Schedule must use some additional processors in order to schedule $T_k$ successfully. Hence, $T_k$ must additionally use some processors reserved for the execution of a previously scheduled task $T_i$ ($i < k$ and $d_i < d_k$).

In this case, let us check whether both $T_i$ and $T_k$ can be scheduled in the New Schedule when Scheduling-Algorithm fails to schedule $T_k$ at the time $\tau^i$ (when $\eta^i = N$ and $\tau^i = d_k$ or $\eta^i_k = b_k$ such that $\tau^i < \tau^j < \tau^i$). If $T_k$ uses some processors reserved for the execution of $T_i$ when $\eta^i = N$ and $\tau^i = d_k$, then $T_i$ cannot satisfy its deadline because there are no available processors to compensate for the additional processors required for the execution of $T_k$ before $d_k$ ($d_i < d_k$, $\tau^i = d_k$ and $\tau^i = N$ such that $\tau^i < \tau^j$). If $T_k$ uses some processors reserved for the execution of $T_i$ when $\eta^i = N$, $\eta^i_k = b_k$ and $\eta^i = N$, then $T_i$ cannot satisfy its deadline, because there are no processors available to compensate for the additional processors required for the execution of $T_k$ either before or after $d_i$.

If $T_k$ uses some processors reserved for the execution of $T_i$ when $\eta^i = N$, $\eta^i_k = b_k$ and $\eta^i = N$, then $T_i$ cannot satisfy its deadline, because there are no processors available to compensate for the additional processors required for the execution of $T_k$ from time $\tau^i$ to $d_i$. Unless $\eta^i_k < b_k$ such that $\tau^i < \tau^j$, it is not possible to allocate some additional processors previously reserved for the execution of $T_i$ to $T_k$. In other words, only when $\eta^i < b_k$ such that $\tau^i < \tau^j$, is it possible to allocate some additional processors previously reserved for the execution of $T_i$ to $T_k$. In summary, the conditions in this case are $\eta^i = N$, $\eta^i < N$, $\eta^i_k < b_k$, and $\eta^i_k = b_k$ such that $\tau^i < \tau^j < \tau^i$. Figure 6 shows this case. In order to execute $T_k$ before $d_k$, $T_k$ must use some of the processors reserved for the execution of a previously scheduled task $T_i$ at time $\tau^i$ ($\eta^i_k > 0$), because no other processors are available before $\tau^i$. Therefore, $T_i$ has to be executed with fewer processors than the original allocation at the time $\tau^i$. In order to compensate for the loss of its previously scheduled processors, $T_i$ must use more processors than the original assignment after the time $\tau^i$, if such processors are available. However, the additional processors available at time $\tau^i$ cannot be allocated to $T_i$, because $\eta^i_k = b_k$ such that $\eta^i_k > 0$ and $i < k$ by Lemma 3. If $\eta^i_k = b_k$, the additional available processors cannot be allocated to $T_i$ at the time $\tau^i$ due to the parallelism bound of $T_i$.

From the above-mentioned facts, the New Schedule cannot satisfy the deadlines of both $T_i$ and $T_k$. Thus, the assumption on feasibility of the New Schedule is a contradiction. Hence, there is no feasible schedule that satisfies the deadlines of $T_1, T_2, \cdots, T_k$ simultaneously when Scheduling-Algorithm fails to schedule $T_k$. This means that there is no feasible schedule if Scheduling-Algorithm fails to find a feasible schedule.

□

**Theorem 2:** Opt-Algorithm always finds a feasible schedule using the fewest processors.

**Proof:** When Opt-Algorithm stops its operation after find-
ing a feasible schedule using \( N' \) processors, let us assume that there is a feasible schedule using \( N'' \) processors such that \( N'' < N' \). Then, it is a contradiction of Theorem 1 since it means that Scheduling-Algorithm may not find the feasible schedule using \( N'' \) processors before finding the feasible schedule using \( N' \) processors. Consequently, Opt-Algorithm always finds out a feasible schedule using the fewest processors.

\[ \square \]

4. Conclusion

This paper presents two polynomial-time algorithms, called Scheduling-Algorithm and Opt-Algorithm, for the real-time scheduling of parallel tasks on multiprocessors, where the tasks have the properties of flexible preemption, linear speedup, bounded parallelism, and arbitrary deadlines. We prove that Scheduling-Algorithm always finds a feasible schedule if there are feasible schedules on given processors, and Opt-Algorithm always finds a feasible schedule using the fewest processors. We also show that the time complexities of Scheduling-Algorithm and Opt-Algorithm are \( O(M^2 \cdot N) \) and \( O(M^2 \cdot N^2) \) in the worst case respectively, where \( M \) is the number of tasks and \( N \) is the number of processors.

References


