Real-time scheduling of linear speedup parallel tasks*

Maciej Drozdowski
Institute of Computing Science,
Poznań University of Technology,
Poznań, Poland

Abstract

In this paper a problem of deterministic scheduling parallel applications in a multiprogrammed multiprocessor system is considered. We address the preemptive case. The number of processors used by a task can change over time. Any task can be executed with linear speedup on a number of processors not greater than some task-dependent constant. This problem can be solved by a low-order polynomial time algorithm for the makespan optimality criterion and tasks with different release times. The algorithm can be executed on-line.

Keywords: Parallel processing, parallel tasks, preemptive scheduling.

1 Introduction

Parallel computer systems are becoming common in scientific and industrial applications [4]. However, the advantage of parallelism can only be taken if the computations are efficiently organised. Hence, there is a demand for good scheduling algorithms in parallel environment.

A parallel application uses more than one processor at a time. Yet, a concept of the tasks using more than one processor at a time is quite recent in the scheduling theory. It has been introduced in [1] as a *multiprocessor* task system in which any task can be executed by only one given number of processors. This has been extended to a *parallel* task system [6] in which any task can be executed on any subset of processors, but the execution time depends on the number of used processors. In the literature (see [2, 10] for surveys) it was assumed that the number of processors used by a task does not change during the task execution. This assumption is not so obvious in

*This research has been partially supported by KBN grant 3P40600106/PI

the context of long numerical applications where the load on processors is balanced and the task can adapt itself to different numbers and speeds of available processors [11]. In this work it is assumed that the number of used processors may change during the execution.

Let us define our problem. We consider a set of n tasks to be scheduled on m parallel identical processors. Each task j requires p_j of processing. It is typical of parallel applications that linear speedup can be maintained only up to some bounded number of processors [9]. Assigning more processors is not very efficient. Hence, we assume that task j can be executed with linear speedup on the number of processors in the range $[1, \delta_j]$, while using more than δ_j processors is not allowed. In this work we assume that $\delta_j \leq m$. The number of used processors may change during execution of the task. Tasks are preemptive, i.e. their execution can be suspended and restarted later (probably on a different number of processors) without incurring additional costs. Hence, the schedule has h intervals where the assignment of tasks to processors is constant. Task j uses $proc_{kj}$ processors in k-th such interval of length t_k. In a feasible schedule $\sum_{k=1}^{h} t_k proc_{kj} = p_j$ for $j = 1, \ldots, n$. Task j can be executed only after its release at time r_j. The optimality criterion is schedule length (makespan) $C_{max} = \max_j c_j$, where c_j is task j completion time.

To denote scheduling problems we will use standard three-field notation with its extensions [3, 10]. Since the previously existing classification does not match our problem perfectly, we will use any^* to denote that speedup is linear, and δ_j that it is linear up to $\delta_j \leq m$ processors. The problem of scheduling tasks requiring more than one processors at the time has already been considered in the literature (see [2, 10] for surveys). In [6] it was shown that when speedup is an arbitrary function, nonpreemptive scheduling on five processors ($P5 \mid any \mid C_{max}$) and preemptive scheduling on an arbitrary number of processors ($P \mid any, pmtn \mid C_{max}$) are strongly NP-hard. Observe, that when $\forall_j \delta_j \geq m$ and speedup is a nondecreasing function, for C_{max} criterion the problem boils down to scheduling on one processor because it is always advantageous to use all processors. For the preemptive case, $\forall_j \delta_j \geq m$, and precedence constraints it has been shown in [5] that the problem is strongly NP-hard even for $n < m$ and chains of three tasks only. However, when an optimal schedule is known for the chains longer than the m-th longest chain, it is possible to merge in polynomial time all the remaining tasks with such a schedule to obtain an optimal schedule for all the tasks [5].

In this work we consider problem $P \mid any^*, \delta_j, r_j, pmtn \mid C_{max}$, i.e. the case where δ_j may have value smaller than m, tasks are ready for processing at various moments of time and makespan is the optimality criterion. The next section presents the optimisation algorithm for our problem.

2 The algorithm

The method presented here uses some ideas of the Muntz-Coffman's algorithm [8] for problem $P \mid pmtn, in-tree \mid C_{max}$. This algorithm schedules tasks according to their level which is the time required to finish all the tasks along the path from the given task to the root of the tree. In [8] a concept of *processing capabilities* has been introduced. Processing capabilities are real numbers which represent a share of all processors that is assigned to process a task for some period of time. Processing capabilities can be also considered as speeds of processing tasks. We will use analogous method to assign processors to tasks. We use also McNaughton's [7] wrap-around rule to schedule pieces of tasks. The basic idea behind the algorithm we propose for problem $P \mid any^*, \delta_j, r_j, pmtn \mid C_{max}$ is to build a schedule starting from the interval where only one task is present and ending with the last interval where all tasks are ready. The more tasks are executed before this last interval, the smaller C_{max} is. Now, we introduce some useful notation.

Height $h(j)$ of task j is the shortest time required to complete j. $h(j)$ is the remaining required processing divided by δ_j. $h(j)$ is $\frac{p_j}{\delta_j}$ initially, it decreases while processing j, and $h(j) = 0$ means that j is completed. We say that two tasks are equal if their heights are equal. Let us assume that there are $l \leq n$ different values of release times, and $r_1 = 0 < r_2 < \ldots < r_l$. We introduce also $r_{l+1} = \infty$. In the algorithm we will denote by:

k - index of interval $[r_k, r_{k+1}]$, $k = 1, \ldots, l$,
R_k - the set of tasks ready in interval k,
τ - the length of the current processing capabilities assignment,
t - the beginning time of the current processing capabilities assignment;
$\overline{\beta}$ - a vector of n processing capabilities for tasks $1, \ldots, n$.

The algorithm

1: $t := 0$; group tasks with release time r_k in set R_k, order tasks in R_k according to nonincreasing heights, $k = 1, \ldots, l$;

2: **for** $k := 1$ **to** l **do**
begin
2.1: order tasks in R_k according to nonincreasing values of $h(j)$ for $j \in R_k$;
2.2: **while** $(r_{k+1} > t)$ **and** $(\exists_{j \in R_k} h(j) > 0)$ **do**
begin
2.2.1: find_capabilities$(R_k, \overline{\beta})$;
2.2.2: calculate times:
if $\exists_{j,j+1 \in R_k} h(j) > h(j+1)$ **then**
$\tau' := \min_{j,j+1 \in R_k} \{ \frac{h(j)-h(j+1)}{\frac{\beta_j}{\delta_j} - \frac{\beta_{j+1}}{\delta_{j+1}}} : \frac{\beta_j}{\delta_j} \neq \frac{\beta_{j+1}}{\delta_{j+1}}, h(j) > h(j+1) \}$ **else** $\tau' := \infty$
- the shortest time required for two tasks $j, j+1$ with different heights to become equal;
$\tau'' := \frac{h(|R_k|)}{\delta_{|R_k|}}$ - the time to the earliest completion of any task;
2.2.3: $\tau := \min\{\tau', \tau'', r_{k+1} - t\}$;
2.2.4: schedule $\tau\beta_j$ piece of task j in interval $[t, t+\tau]$ according to McNaughton's wrap-around rule for $j \in R_k$;
2.2.5: $h(j) := h(j) - \frac{\tau\beta_j}{\delta_j}$ for $j \in R_k$;
2.2.6: $t := t + \tau$;
end;
2.3: **if** $(\exists_{j \in R_k} h(j) > 0)$ **then** $R_{k+1} := R_{k+1} \cup \{j : j \in R_k, h(j) > 0\}$;
end; (* end of the algorithm *)

procedure find_capabilities(**in:**X;**out:**$\overline{\beta}$); (* X - a set of tasks *)
begin
3.1: $\overline{\beta} := \overline{0}$; $avail := m$; (* $avail$ is the number of free processors *)
3.2: **while** $avail > 0$ **and** $\mid X \mid > 0$ **do**
begin
3.2.1: construct set T of the highest tasks in X with $h(j) > 0$;
3.2.2: **if** $\sum_{j \in T} \delta_j > avail$ **then**
begin
3.2.3: $\beta_j := \delta_j \frac{avail}{\sum_{j \in T} \delta_j}$ for task $j \in T$; $avail$:=0;
end
else (* tasks in T can use at most $avail$ processors *)
begin
3.2.4: $\beta_j := \delta_j$ for $j \in T$; $avail := avail - \sum_{j \in T} \delta_j$;
end;
3.3: $X := X - T$;

end; (* of **while** loop *)
end; (* of procedure find_capabilities *)

High level description. Intervals $[r_k, r_{k+1}]$ are considered consecutively in lines 2-2.3. In these intervals, subintervals are created in lines 2.2-2.2.6 where processing capabilities assignment is not changing. Tasks are assigned processors in line 2.2.1 in a way analogous to the one proposed in [8]. High tasks are given preference (line 3.2.1). When there is more processors than can be simultaneously required by the ready tasks, a maximal possible number of processors is assigned in line 3.2.4. Otherwise, processors are shared (line 3.2.3) by equal tasks such that their heights decrease at the same pace (cf. line 2.2.5). The length of the current assignment is calculated in line 2.2.3. The assignment of processors to tasks changes in three cases: $h(j)$ for some initially higher task becomes equal to $h(j+1)$ of some initially lower task and processing capacities must be recalculated such that equal tasks decrease their heights with the same speed (calculated as τ' in line 2.2.2), the lowest task in R_k finishes (τ''), or the end of interval is encountered and tasks in R_{k+1} must be considered. In line 2.3 tasks from R_k not completed by the end of interval k are added to R_{k+1} to be considered also in the next interval.

Lemma 1 *The algorithm for* $P \mid any^*, \delta_j, r_j, pmtn \mid C_{max}$ *is correct.*

Proof. First we prove that the algorithm stops. Procedure find_capabilities stops because in each execution of while loop in lines 3.2-3.3 at least one task is removed from X. Equal tasks reduce their heights with the same speed (cf. line 2.2.5). Hence, once two tasks become equal they remain equal until their completion. Height of initially higher task cannot fall below the height of an initially lower task, which is guaranteed by calculation of τ' in line 2.2.2. Thus, two tasks can become equal at most $n-1$ times. We conclude that while loop of lines 2.2-2.2.6 can be executed only a limited number of times. As a result of this, the algorithm stops.

Now, we consider feasibility of the obtained schedule. Tasks are not scheduled before their release times because any task released at r_k can be considered in sets $R_k, \ldots, R_l$, not $R_1, \ldots, R_{k-1}$. No task j is ruled out from consideration as long as $h(j) \neq 0$. This means that j is no longer considered only when it received required processing. In each subinterval built in lines 2.2-2.2.6:

(i) no task j is assigned more than δ_j processing capabilities which is result of lines 3.2.3 and 3.2.4 in procedure find_capabilities. McNaughton's wrapping-around procedure uses a new processor for a task only if the previous processor is filled completely. Hence, no more than δ_j processors can be used by any task.

(ii) the sum of processing requirements of tasks is $\sum_{j\in R_k} \tau\beta_j = \tau(\sum_{j\in R'_k} \delta_j + \sum_{j\in R_k - R'_k} \frac{\delta_j(m-\sum_{i\in R'_k}\delta_i)}{\sum_{i\in R_k - R'_k}\delta_i}) = \tau\, m$, where $R'_k \subseteq R_k$ is a set of tasks which received processing capabilities in line 3.2.4 of procedure find_capabilities. Thus, the sum of processing requirements for the subinterval does not exceed its capacity.

(i) and (ii) ensure that in the subinterval created in lines 2.2-2.2.6 a feasible schedule can be obtained by McNaughton's wrap-around rule. □

Theorem 1 *A2 is an optimisation algorithm with complexity* $O(n^2)$.

Proof. In each subinterval created in lines 2.2-2.2.6 either all m processors are occupied, or as many processors are occupied as possible. Hence, capacity of interval $[r_k, r_{k+1}], k = 1, \ldots, l-1$, is maximally exploited. Thus, the total processing requirement moved to R_{k+1} is minimal possible. Since tasks with the longest expected execution time are preferred, also $\max_{j\in R_k} h(j)$ is maximally decreased. The above arguments hold inductively for intervals $[r_k, r_{k+1}]$ $(k = 1, \ldots, l-1)$. Thus, also R_l has tasks with the lowest possible $\max_{j\in R_l} h(j)$ and their total processing requirement $\sum_{j\in R_l} h(j)\delta_j$ is minimal.

Now, we will show that $\max\{\max_{j\in R_l} h(j), \frac{1}{m}\sum_{j\in R_l} h(j)\delta_j\}$ is the length of the schedule in the last interval. Note, that no shorter schedule in the last interval can exist than $\max_{j\in R_l} h(j)$ - the longest time required to complete some task, and $\frac{1}{m}\sum_{j\in R_l} h(j)\delta_j$ - the time required to process all tasks when processors are equally loaded. We can distinguish two cases. If $\sum_{j\in R_l}\delta_j \le m$ then tasks are always assigned processing capacity δ_j in line 3.2.4. Hence, also δ_j processors are used by the tasks. This leads directly to a schedule of length $\max_{j\in R_l} h(j)$. Consider the case of $\sum_{j\in R_l}\delta_j > m$. Let us remind that tasks are ordered according to nonincreasing height. There must exist some task j' such that $\sum_{j=1}^{j'-1}\delta_j < m$ and $\sum_{j=1}^{j'}\delta_j \ge m$.

Tasks with height greater than $h(j')$ are assigned processing capacity δ_j and are executed with the greatest possible speed. As soon as the height of

some of them drops to the level of $h(j')$ (detected by calculation of τ' in line 2.2.2) it is treated in the same way as tasks with height $h(j')$.

Tasks with the height equal to $h(j')$ share processors (line 3.2.3).

Tasks with height smaller than $h(j')$ are not executed until the height of $h(j')$ drops to their level. From such moment on these tasks share processors with other tasks of height $h(j')$. Consequently, all tasks of height $h(j')$ (and lower initially) are completed in the same subinterval created in lines 2.2-2.2.6. Until such a moment the schedule in the last interval has no idle time. If after that there are still tasks with height greater than $h(j') = 0$ then such tasks have been always assigned δ_j processors, the schedule cannot be shorter, and its length is determined by $\max_{j \in R_l} h(j)$. In the opposite case, there is no more tasks with height greater than $h(j')$ and a schedule without idle time, i.e. of the length $\frac{1}{m} \sum_{j \in R_l} h(j)\delta_j$, has been built. Thus, we conclude that the schedule in the last interval has a minimal possible length. Consequently, the whole schedule is optimal.

The complexity of the algorithm can be estimated as follows. Grouping tasks according to their release times in line 1 can be implemented in $O(n \log n)$ time. There are $O(n)$ values of k considered in loop 2-2.3. Ordering tasks according to their heights is equivalent to sorting and requires $O(n \log n)$ time in line 1 and $O(n)$ time in line 2.1 (merging of R_k and R_{k-1}). Procedure find_capabilities can be executed in $O(n)$ time because it assigns processing capabilities to at most n tasks. Lines 2.2.2-2.2.6 require $O(n)$ time. Thus, the total complexity is $O(n^2)$. □

Note, that no information about tasks in $R_{k+1}, \ldots, R_l$ is necessary to schedule tasks with release times smaller than r_{k+1}. Hence, the above algorithm can be run on-line i.e. it builds optimal schedules using only the information about tasks that have been already released. In the above algorithm it was assumed that the cost of preemption (or a contex switch) is negligible. In practical situations the schedule loses its optimality when the context switch lasts some time. Yet, we are able to estimate the worst case lengthening of the schedule from the number of preemptions. The number of preemptions can be determined by the number of subintervals where processing capabilities are constant and the preemptions within such subintervals. There can be at most $3n$ subintervals and at most two context switches are associated with each task within the subinterval. Hence, there are at most $6n^2 + 3n$ context switches on any processor in the schedule.

Consider a problem of scheduling parallel tasks for maximum lateness criterion i.e. $P \mid any^*, \delta_j, pmtn \mid L_{max}$. There are also l different due-dates: $d_1 \leq d_2 \leq \ldots \leq d_l$. This problem can be solved by a modification of the above algorithm. For problem $P \mid any^*, \delta_j, pmtn \mid L_{max}$ we have to guarantee that task j is feasibly executed in interval $[0, d_j + L_{max}]$ and L_{max} is minimal possible. For problem $P \mid any^*, \delta_j, r_j, pmtn \mid C_{max}$ we have to schedule task j in interval $[r_j, C_{max}]$ and minimise C_{max}. Thus, for instance I of $P \mid any^*, \delta_j, pmtn \mid L_{max}$ we can construct equivalent instance I' of problem $P \mid any^*, \delta_j, r_j, pmtn \mid C_{max}$ by assuming $r'_j = d_l - d_{l-j+1}$ for $j = 1, \ldots, l$. Schedule S' for I' should be read from the end at C'_{max} to the beginning to be schedule S for $P \mid any^*, \delta_j, pmtn \mid L_{max}$, i.e. each time instant t' in S' has equivalent $t = C'_{max} - t'$ in S.

3 Conclusions

In this work we analysed a problem of preemptive scheduling parallel applications which preserve linear speedup only up to some limited number $\delta_j \leq m$ of processors. Tasks were assumed to be ready for processing at various moments of time, makespan is optimality criterion. Low order polynomial-time on-line algorithm has been proposed for this problem.

References

[1] J.Błażewicz, M.Drabowski, J.Węglarz, Scheduling multiprocessor tasks to minimize schedule length, *IEEE Trans. Comput.* **C-35** (1986) 389-393.

[2] J.Błażewicz, M.Drozdowski, J.Węglarz, Scheduling multiprocessor tasks - a survey, *International Journal of Microcomputer Applications* **13** (1994) 89-97.

[3] J.Błażewicz, K.Ecker, G.Schmidt, J.Węglarz, *Scheduling in Computer and Manufacturing Systems* (Springer Verlag, New York, 1993).

[4] W.J.Camp, S.J.Plimpton, B.A.Hendrickson, R.W.Leland, Massively Parallel Methods for Engineering and Science Problems, *Comm. ACM* **37** (1994) 31-41.

[5] M.Drozdowski, W.Kubiak, Scheduling parallel tasks with sequential heads and tails, Faculty of Business Administration, Memorial University of Newfoundland, working paper, April 1995.

[6] J.Du, J.Y-T.Leung, Complexity of scheduling parallel task systems, *SIAM J.Discrete Math.* **2** (1989) 473-487.

[7] R.McNaughton, Scheduling with deadlines and loss functions, *Management Science* **6** (1959) 1-12.

[8] R.R.Muntz, E.G.Coffman Jr., Preemptive scheduling of real-time tasks on multiprocessor systems, *Journal of ACM* **17** (1970) 324-338.

[9] K.C.Sevcik, Application scheduling and processor allocation in multiprogrammed parallel processing systems, *Performance Evaluation* **19** (1994) 107-140.

[10] B.Veltman, B.J.Lageweg, J.K.Lenstra, Multiprocessor scheduling with communication delays, *Parallel Computing* **16** (1990) 173-182.

[11] R.D.Williams, Performance of dynamic load balancing algorithms for unstructured mesh calculations, *Concurrency: Practice and Experience*, **3** (1991) 457-481.