A Parallelizing Algorithm for Real-Time Tasks of Directed Acyclic Graphs Model

Manar Qamhieh, Serge Midonnet, Laurent George

A Parallelizing Algorithm for Real-Time Tasks of Directed Acyclic Graphs Model

Manar Qamhieh, Serge Midonnet
Université Paris-Est, France
{manar.qamhieh,serge.midonnet}@univ-paris-est.fr

Laurent George
ECE-Paris, France
lgeorge@ece.fr

Abstract

In this paper, we consider parallel real-time tasks following a Directed Acyclic Graph (DAG) model. This task model is classical in embedded and industrial system applications. Each real-time task is defined by a set of subtasks under precedence constraints. With each subtask being associated a worst case execution time and a maximal degree of parallelism. We propose a parallelizing algorithm based on the critical path concept, in which we find the best parallelizing structure of the task according to the response time and the required number of processors, considering the worst case execution time of the subtasks.

1. Introduction

Multi-core processors are widely produced nowadays in order to cope with the physical constraints of the manufacturing process, which makes the development of parallel softwares more and more important. The same concept can be applied to real-time systems. Those systems have been thoroughly studied over the last forty years but were mostly focused on sequential processing. In order to get advantage of the new hardware developments in real-time systems, a new challenge towards integrating parallelism in real-time systems has appeared.

Many models of parallelism have been used in programming languages and APIs, but few of them have been studied in real-time systems. In our work, we propose a more general model of parallel tasks following a Directed Acyclic Graph (DAG) with 2 levels of parallelism. This type of model can be used to represent real industrial applications like video surveillance network and complex 3D games, in which many images are processed at the same time. Then, we propose a parallelism algorithm in order to execute the DAG task with the best parallel structure of the task while taking the response time in consideration.

In this paper, we start by presenting other related parallel task models already studied in the literature in section 2. Then we present our task model in section 3. Section 4 describes the parallelizing algorithm applied to a DAG. Finally, we finish the paper with conclusion and perspectives in section 5.

2 Related Work

Parallelism in real-time systems is a new domain with many open issues to be studied. There exist many programming APIs which support parallelism like OpenMP [1], Cilkplus, Go, *etc*. The fork-join parallelism model is used in OpenMP. It is defined as a sequence of sequential and parallel segments, since the main thread of the task forks into many parallel threads during the execution, and when they finish their execution, they join the main thread again. This model is studied in [2] and a stretching algorithm has been proposed to transform parallel tasks into sequential tasks when possible.

However, a more general model of parallel tasks has been recently proposed in [4] which overcomes the restrictions of the fork-join model. It replaces the sequential-parallel segment ordering with parallel segments. Those segments have an arbitrary number of threads, but the execution time of all the threads in the same segment is the same. In their work, they propose a decomposition algorithm to assign local deadlines for the different parallel segments and to transform them into sporadic sequential tasks. They also provide a resource augmentation bound for this decomposition algorithm of 4 when the tasks are scheduled using global EDF and of 5 for partitioned deadline monotonic scheduling, respectively.

The latter model has been generalized in [3], they propose sporadic parallel real-time tasks with constrained deadlines. It differs from the model in [4] by allowing threads in the same segment to have different worst case execution time. As a result of the paper, they optimize the number of processors needed to schedule this model of tasks while applying the same resource augmentation bounds performed before.

3. Task Model

In this paper, we deal with parallel implicit-deadline real time tasks, where the deadline of a task equals to their period, and each task of this model is represented by a directed acyclic graph (DAG), which is a collection of subtasks and directed edges, which represents the execution flow of the task and the precedence constraints between the subtasks. Precedence constraint means that each node can start its execution when all of its predecessors have finished theirs. If there is an edge from subtask $\tau_{i,u}$ to $\tau_{i,v}$, then we can say that $\tau_{i,u}$ is a parent of $\tau_{i,v}$, and $\tau_{i,v}$ has to wait for $\tau_{i,u}$ to finish its execution before it can start its own. Each vertex in the graph may have multiple parents, and multiple child vertices as well, but each graph should have single source and sink vertices.

Each parallel real-time task τ_i consists of a set of q_i subtasks, $\tau_i = \{\tau_{i,1}, \tau_{i,2}, ..., \tau_{i,q_i}\}$, and each subtask $\tau_{i,k}$ is represented as the following:
$\tau_{i,k} = \{e_{i,k}, m_{i,k}\}$, where $e_{i,k}$ is the worst execution time of the subtask, and $m_{i,k}$ is the maximal degree of parallelism of $\tau_{i,k}$, which means that the subtask $\tau_{i,k}$ can be scheduled on $m_{i,k}$ parallel processors at most.

Figure 1 shows an example of a parallel real-time task τ_i of 6 subtasks. $\tau_1 = \{\tau_{1,1}, \tau_{1,2}, ..., \tau_{1,6}\}$. τ_1 has a single source subtask $\tau_{1,1}$ and a single sink subtask $\tau_{1,6}$. Each subtask is characterized by an ordered pair, the first is the total execution time of the subtask and the second is the maximal degree of parallelism. As shown in Figure 1, $\tau_{1,4}$ for example, has a total execution time of 3, and it can be parallelized at most on 3 processors with 1 execution time unit on each processor.

This parallel real-time tasks of graph model have 2 levels of parallelism; a inter-subtask parallelism and intra-subtask parallelism. The inter-subtask parallelism is caused by the precedence constraints between the different subtasks in the task. Subtasks sharing the same parent means that they are activated at the same time (when the parents finish their execution), allowing them to execute in parallel on multiple processors. $\tau_{1,2}$ and $\tau_{1,3}$ in Figure 1 are an example of this subtask parallelism. The intra-subtask parallelism is denoted by the possibility of parallelizing each subtask $\tau_{i,k}$ on x number of processors, where $1 \leq x \leq m_{i,k}$. A parallel subtask $\tau_{i,k}$ with maximal degree of parallelism $m_{i,k}$ equals to 1 can be considered as a sequential subtask.

In this paper, we chose to work with a generalized model of parallel tasks, a model that describes the industrial and embedded systems applications. We are different from the parallel models of [4] and [3] by proposing the intra-subtask parallelism within the real-time task model of graphs. While in the other models, a parallel real-time task starts as a collection of segments and then they propose the possibility of transforming it into a DAG, while the same decomposition algorithm and feasibility analysis remain applied.

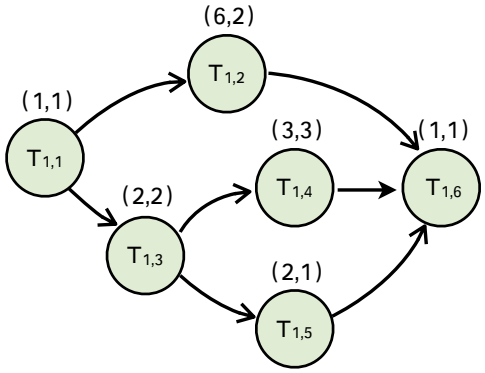

Figure 1. Example of the task model.

4 Parallelizing algorithm for a parallel real-time task

4.1 Critical Path Calculations:

This general parallel task model of graphs has many possibilities regarding the execution flow of the subtasks in the same task due to the intra-subtask parallelism described before in 3. Task τ_1 from Figure 1 has 3 parallelizable subtasks ($\tau_{1,2}, \tau_{1,3}, \tau_{1,4}$), they can either be parallelized or executed sequentially. The simplest solution will be to parallelize them all up to their maximum degree of parallelism. However, this solution will achieve the minimum response time of τ_1 when compared with other parallelizing structures, but with no consideration for the precedence constraints within the subtasks or the number of processors needed.

In this section, we propose a parallelizing algorithm that uses the critical path of a graph technique which is based on the depth-first search algorithm, and finds the best parallel structure of the task with minimum response time and number of processors. We assume a univocal task to processor assignment (the processors assigned to a task will only run this task). This leads to consider that the number of processor is high compared to the number of tasks. We plan to remove this restriction as a further work.

The algorithm we propose considers first a system with unlimited number of processors. We finally obtain with the algorithm the exact number of processors required for a given task.

Definition. *Critical path P_i of a parallel real-time task is the path through task τ_i with the longest sequential execution time when τ_i is executed without intra-subtask parallelism on a system with infinite number of processors.*

For a real-time graph task, P_i can be considered as the maximum execution time of τ_i, that τ_i will need at least P_i

units of time to finish its execution when all its subtasks execute sequentially without parallelism. Subtasks in the critical path are called the critical subtasks.

According to task τ_1 shown before in Figure 1, Figure 2(a) shows its critical path, $P_1 = \{\tau_{1,1}, \tau_{1,2}, \tau_{1,6}\}$. We can notice that this path has the longest consecutive execution time of the task which is 8, while the other possible paths $\{\tau_{1,1}, \tau_{1,3}, \tau_{1,4}, \tau_{1,6}\}$ and $\{\tau_{1,1}, \tau_{1,3}, \tau_{1,5}, \tau_{1,6}\}$ have execution time of 7 and 6 respectively.

According to Algorithm 1, we can calculate the critical path and the slack time of the non-critical subtasks by performing forward and backward calculations, the forward calculation of a subtask x is denoted as $F(x)$, for each subtasks in the graph starting from the source of the graph, $F(x)$ is the maximum sum of the execution time of its preceding subtasks. $F(\tau_{i,sink})$ is the response time of τ_i.

Backward calculation $R(x)$ is performed on each subtask starting from the sink of the graph, we calculate the minimum available time for the path of subtasks to execute from the x until the source of the graph. For each subtask $\tau_{i,j}$, the difference between the backward and forward calculations is its slack time, if it equals to 0, then $\tau_{i,j}$ is a critical subtask.

In order to perform the calculations proposed in Algorithm 1, we need to find the subtask flow in the graph, by determining depth levels of each subtask. The source subtask will be in the first level, and its children are in 2, and so on...

For any subtask $\tau_{i,k}$ in the graph, its depth is denoted as $h(k)$ and is calculated as the following:

$$h(k) = \max_{\text{u parent of k}} h(u) + 1$$

If a subtask $\tau_{i,k}$ has multiple parents, it will follow the parent subtask with the maximum depth. The maximum depth of the graph is denoted by $H = h(\tau_{i,sink})$.

In our algorithm and while calculating the critical path of a graph, we give a higher priority for the parallel nodes, that if we have 2 different critical paths, we choose the one with the highest probability of parallelism. In this case, we increase the number of parallelized subtasks while keeping the same response time or reduce it.

The critical path method is not new, it is used in operation analysis of graph tasks and based on depth-first search algorithm, in order to choose the sequence of actions that will define the execution of the task as whole, and any delay in these actions will delay the total execution time of the operation.

4.2 Parallelizing algorithm:

By using the critical path algorithm and as shown in Figure 2, we can find the critical subtasks in τ_i that determine the response time of the task, where the rest of the "non-critical subtasks" has certain amount of slack time calculated by Algorithm 1. Since we are concerned with parallelizing the parallel subtasks of the graph while keeping the best possible response time of the task, we will start by parallelizing the critical subtasks.

Algorithm 1 Calculating the critical path of a graph

for *depth* $h = 1 \rightarrow H$ **do**
  for *each subtask k in h* **do**
    $F(k) = \max_{\text{u parent of k}} F(u) + e_{i,k}$
  end for
end for
$R(H) = F(H)$
for $h = (H-1) \rightarrow 1$ **do**
  for *each subtask k in h* **do**
    $R(k) = \min_{\text{u child of k}} F(u) - e_{i,u}$
  end for
end for
for $i = 1 \rightarrow s_i$ **do**
  if $F(i) = R(i)$ **then**
    subtask *i* is a critical subtask.
  else
    Slack(i) = R(i) - F(i).
  end if
end for

Figure 2 shows the parallelizing process of the task τ_1 shown before. The first step is to find the critical path of the task, which is $P_1' = \{\tau_{1,1}, \tau_{1,2}, \tau_{1,6}\}$, which has a single parallel critical subtask $\tau_{1,2}$, and it can be executed on 2 processors ($m_{1,2} = 2$). So, we will divide it into 2 sequential subtasks of execution time = $6/2 = 3$. This parallelizing process will modify the structure of the graph and its response time, so we can calculate a new critical path $P_1'' = \{\tau_{1,1}, \tau_{1,3}, \tau_{1,4}, \tau_{1,6}\}$ which can be also parallelized as shown in Figure 2(b). This process will be repeated until we get a graph with critical path that can't be parallelized any more, like in Figure 2(c).

This parallelizing process will change the structure of the graph and reduce the response time of the task as well. As shown in Figure 2, τ_1 had a response time of 8 when executed on 3 processors in the first iteration, but in the final iteration, it has 5 units of response time when executed on 6 processors. We believe that reducing the response time of the task on the behalf of the number of processors is acceptable since multi-processor systems are widely manufactured. However, in the next section we will optimize the number of processors resulted from the parallelizing algorithm.

4.3 Optimization:

As mentioned before, the previous parallelizing algorithm tends to parallelize the real-time subtasks to their

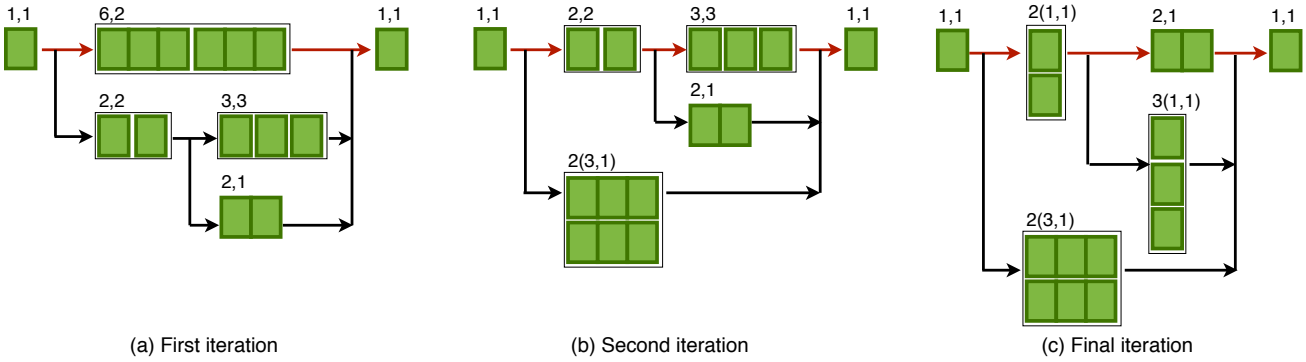

Figure 2. Example of the parallelizing algorithm.

maximal degree of parallelism in order to reduce the response time of the task, without considering the placement of these parallel subtasks or the number of processors needed for scheduling. As a final step of the algorithm we tend to find a better placement of the generated parallelized subtasks so as to reduce the number of processors without affecting the calculated response time. Optimizing the placement of the non-critical subtasks depends on 2 factors; the execution time of the subtask and the slack time.

According to algorithm 1, critical subtasks have no slack time, and they have to execute without delay in order to get the best response time of the task. But this is not the case for the non-critical subtasks, their paths through the graph will have strictly an execution time equal to the critical path at most, and algorithm 1 can calculate the slack time of all the non-critical subtasks in the graph.

Figure 3(a) shows the final placement of the subtasks of τ_1 after applying the parallelizing algorithm, and we can notice that task τ_1 can be fully executed on 6 processors with 5 units of time. However, this placement of subtasks is not the optimal, since there exists a non-critical subtask in the graph with slack time $S_{1,4} = 1$. So, the 3^{rd} thread of the subtask can be placed in the slack time of the subtask without increasing the response time of the τ_1.

The advantage of this optimizing process is to occupy the idle time of processors with non-critical subtasks with sufficient slack time. This optimizing step will reduce the overall number of processors needed by the task in order to execute within the same response time.

5 Perspective and Conclusion

In this paper, we have introduced a parallel real-time tasks graph model, and we have proposed a parallelizing algorithm for this model which gives the best parallelizing structure of the task according to the response time when executed on a specific number of processors. Until now, we only considered a univocal task to processor association, but we aim to extend our work to take into account

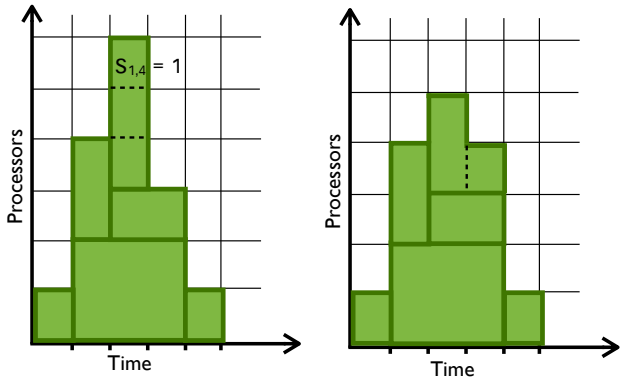

Figure 3. Optimizing the parallelized graph using slack time.

several parallel graph tasks on a processor, and study their schedulability and interference.

The parallelized task can be seen as a set of parallel segments with arbitrary number of threads execute on multiple processors, feasibility analysis studies used before in [4] and [3] can be adapted on our task model, and in the future we will work on proposing feasibility analysis for the schedulability of tasks parallelized using the proposed DAG model.

References

[1] Openmp, http://www.openmp.org.

[2] K. Lakshmanan, S. Kato, and R. (Raj) Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *IEEE RTSS, 2010*.

[3] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Optimizing the number of processors to schedule multi-threaded tasks. In *IEEE RTSS WiP Session, 2011*.

[4] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *IEEE RTSS, 2011*.