A Heuristic Real-Time Parallel Scheduler Based on Task Structures[*]

Qiegang Long and Victor Lesser
Department of Computer Science
University of Massachusetts
Technical Report 95-92

Abstract

The development of networks and multi-processor computers has allowed us to solve problems in parallel. The task of efficiently coordinating parallel processors is formidable. This paper presents a heuristic parallel real-time scheduler that analyzes the interactions among the tasks, and builds a parallel schedule that tends to take advantage of those interactions.

[*]This work is supported by NSF grant IRI9321324.

1 Introduction

The development of networks and multi-processor computers has allowed us to solve problems in parallel. The task of efficiently coordinating parallel processors is formidable. It requires a scheduler to specify which processor to allocate for what problem and when. Existing parallel schedulers can be divided into three categories [1]: graph-theoretic [2] [3], integer 0-1 programming approach [4] and the heuristic approach [5] [6]. Many of these methods use the cost of computation and communications as the schedule objective function, assuming that the duration of executing a task and its result quality are independent of the order of task execution. However, this assumption does not hold if there are complex interactions among tasks.

It has been observed that the tasks needed to solve a problem are often related to each other. The two most obvious relationships are *subtask* and *dependency*. A reasonable parallel schedule can be generated only if the scheduler has taken into account these relationship. For example, if there is a dependency relationship between two tasks, the dependent task cannot be executed before the other is done. Decker and Lesser [7] have identified several kinds of other task relationships which also affect the quality of a schedule. Among them, there are two that are most relevant here:

Facilitates Task A facilitates Task B, if execution of Task A will decrease the duration of executing Task B, or increase its quality.

Hinders This is the negative form of *Facilitates* relationship. Task A hinders Task B, if execution of Task A will increase the duration of executing Task B, or decrease its quality.

These kinds of relationships exist frequently in real-life applications. For example, to construct a track from a set of acoustic signals caught by a sensor monitoring traffic, we can start building and extending the track from any signal. But since the signal qualities vary due to noise or the distance between the vehicle and the sensor, it may be preferable that we interpret the strong signals first so that we can quickly get a rough track. This rough track can help us to identify what weak signals are pure noise (like those far away from the draft track), and what are the real signals from the vehicle with poor quality. In this case, it is clear that interpreting strong signals facilitates the task of interpreting weak ones by reducing the needed interpretation time. Exploiting these task relationships is crucial to building high quality schedules, but it makes scheduling even more challenging and computationally expensive for these reasons:

- *Facilitates* and *hinders* task relationships introduce the variable duration of executing a task and its result quality.
- With more task relationships taken into account, any small change in the schedule can affect many tasks.
- How to choose task relationships to exploit becomes an important issue. For example, it is not always possible or desirable to take advantage of all the available *facilitates* relationships to get higher schedule quality. This is especially true for parallel scheduling. Suppose that two tasks can be executed in parallel, but if they are scheduled to be executed sequentially to benefit from the *facilitates* relationship, the beneficiary may not be able to finish before its deadline. Thus, the scheduler has to decide which task relationship can be exploited in view of the overall schedule quality.

In this paper, we present a heuristic parallel real-time scheduler which is based on the design-to-time algorithm developed by Garvey and Lesser [8]. Our scheduler builds schedules incrementally. It uses a *near-greedy* algorithm to construct a draft schedule first, and then employs a iterative repairing procedure to focus on the task relationships that have not been exploited. This parallel scheduler demonstrates two important issues: how to exploit available task relationships and how to do it without excessive computational effort.

In the following section, we begin by introducing our scheduling task environment and our assumptions. Section 3 presents the overview of our algorithm, followed by a detailed description of the parallel scheduling algorithm in section 4 and the approach we use to incrementally refine the schedule in section 5. Throughout the paper, we will use examples to explain how each algorithm works. Section 7 reviews related work, and section 8 analyzes our approach and offers suggestions for future research.

2 Environment and Assumptions

Our scheduler works in a modified TÆMS environment [7]. The scheduler is given a set of task groups that are represented in TÆMS task structure, as shown in Fig.

1. The leaves in a task structure are the executable methods. The other non-leaf nodes act as quality accumulation functions; they represent tasks that achieve quality from their subtasks (e.g., *sum* function denotes that the quality of TG_2 is the total quality of its subtasks). Each task group denotes an independent problem that needs to be solved. *Subtask* relationship is represented in black line, it links a parent node to its children. Any other links (dotted lines with arrows) represent task interactions like *enables*, *facilitates* or *hinders*.

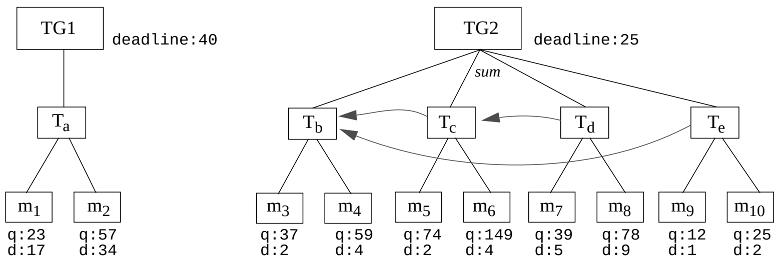

Figure 1: Task groups and TÆMS task structure.

There are a fixed number of agents (or processors) existing in the environment. The objective of the scheduler is to choose appropriate methods, and to allocate them to the agents so that the agents as a whole can achieve a higher total result quality for the given task groups in a shorter duration. We use three criteria to evaluate the parallel schedule:

1. the deadline of every task group is satisfied; and
2. the higher qualities of the task results are, the better the parallel schedule is; and
3. if two schedules achieves the same quality, the shorter the schedule duration is the better the schedule.

3 General Algorithm Description

The design-to-time algorithm consists of three stages. Since there may be more than one method to achieve an individual task (e.g., there are two methods m_1 and m_2 for task T_a in Fig. 1), the scheduler first identifies a set of *alternative* ways to achieve all independent task groups. Each *alternative* is a set of methods, all the task groups (independent problems) are achieved if every method in an alternative is successfully executed. Thus, an alternative represents what needs to be done. After generating alternatives, the scheduler then proceeds to actually schedule each alternative for the computational agent to execute. This stage decides when to execute what method. After this stage, the scheduler uses an iterative process to repair and refine the generated schedule. The best schedule is then given to the computational agent to execute.

The design-to-time algorithm is a heuristic approach. At each stage, there are domain-independent heuristics that are used to limit the number of options explored. These heuristics are mainly based on the task relationships, quality accumulation functions and task parameters, such as deadline or earliest-start-time. To schedule the alternative which is generated at the first stage, the scheduler uses a set of heuristics which includes *enforce enables*, *enforce hard deadlines*, *enforce earliest start time*, *prefer facilitators*, *delay facilitates*, *prefer increased quality*, *avoid violating commitments*, *prefer satisfying commitments* and *prefer earlier deadlines*. The detailed description of these heuristics can be found in [8]. With these heuristics, the scheduler uses a simple evaluate-choose loop: rate each method in the schedule against the heuristics, select the one with the highest nonnegative rating and then add it to the existing schedule. If there is no nonnegative rating, add some idle time at the end of the schedule. This process continues until all the methods in the alternative have been added to the schedule, or a failure has been recognized because some remaining method cannot meet its deadline.

Our parallel scheduler uses a similar three stages. The alternative generation stage is borrowed directly from the design-to-time scheduler. At the second stage, we also use the set of rating heuristics but extend the scheduling process so that it can generate parallel schedules for multiple processors/agents. The number of processors to schedule for is an external parameter to this process[1]. Due to the nature of parallel schedule, we also use a new iterative process to refine the schedule in the third stage.

[1] One possible future direction of this work is for the scheduler to decide how many processors are cost-effective for the given set of task structures. In order to make this decision, some model is necessary to relate cost of using a processor to value achieved by task structure.

4 Generating parallel scheduling

One of the most important factors that distinguishes one schedule from another is the order that a method appears in the schedules. This order is determined by the heuristics which are based on the task interactions, quality accumulation functions and some task parameters. When there is more than one computation agent, the order of a method in the parallel schedules may be different. For example, if there are two independent methods, and both can be started now, in parallel, we can allocate a free agent for each method, thus both method orders are 1. In the single agent case, it is clear that one method can be started only after another has been done. However, since the heuristics encourage or tend to take advantage of task relationships to increase schedule quality, it can be expected that using the same set of heuristics in the parallel scheduler would result in a compatible parallel schedule. By compatible schedules, we mean that the method order in one does not violate that of the others. For example, the parallel schedule $\{(m_1, m_4)(m_2, m_3)\}$ for two agents is compatible with a single agent's schedule (m_1, m_2, m_3, m_4), but $\{(m_2, m_1)(m_3, m_4)\}$ is not since method m_2 is executed before m_1 in one agent's schedule.

With the scheduling heuristics from the design-to-time algorithm, our scheduler uses this process to generate a parallel schedule:

1. Initialize an empty schedule for each computation agent.

2. Use all the heuristics to rate each method which has not yet been scheduled against every agent. This rating procedure results in a set of tuples in the form of $(agent, method, rating)$. If there is a tuple whose rating is non-negative, go to the next step. Otherwise, skip step 3.

3. Select the best tuple, add the method to the end of the agent's schedule. Go to step 2.

4. If there are methods that are not yet scheduled but none of them gets non-negative rating, signal failure and stop. Negative rating for a method means that it is inappropriate to add the method to the schedule because its execution adds no value to the computation.

To rate a method against an agent, the scheduler considers what change of the parallel schedule quality can be expected if appending that method directly to the end of the agent's schedule. It should be noted that at this step the scheduler is searching only in a two-dimension space of methods and agents to find the "best" match. For each pair, the scheduler does not try to figure out when is the best time for the agent to execute the method. The method is simply added to the agent's schedule so that it can be executed immediately after the previous one is done. In some situations, however, it might be better to add some idle time at the end of the schedule before adding the method so it can benefit from the task relationships like *enables* or *facilitates* resulting from the execution of a method at another agent. The reason that the scheduler does not allocate any idle time at this moment is because it is hard to make the decision about what the effect of this idle time would be on the ability to schedule other methods necessary for completing the whole computation[2]. If too much idle time is allocated, the scheduler may later find that some methods cannot be scheduled to meet their deadlines. There are at least two variances that make such predication difficult:

1. If a method is enabled by some other method, the scheduler cannot guarantee that method will be executed eventually since the enabling one may not get scheduled. Thus the number of the methods that will be actually executed is uncertain.

2. The execution time of a method may be uncertain since the scheduler does not know whether it will be facilitated and to what extend.

We use the task groups in Fig. 1 to illustrate our schedule generation process. Assume there are two computational agents A_1 and A_2, and all the gray lines in the figure represent *facilitates* relationship. For simplicity, we further assume that the earliest-start-time of each method is *NOW* (which means no time requirement on when a method can be executed), and the deadline of each method is that of the task group it belongs to. The scheduler starts by generating a set of alternatives. Among them, alternative $m_2, m_4, m_6, m_8, m_{10}$ is very promising since it returns maximum quality. Now consider this alternative. Methods m_4 and m_6 will get a low rate (0) because of heuristic *delay facilitates* (meaning that they cannot benefit from the facilitate relationships). The scheduler will find that method m_8 gets the highest rating for both agents (they have the same empty schedule at this time) since it has a high quality and it facilitates m_6 (heuristic *prefer facilitators* increases the ratings of facilitating methods). Thus the scheduler randomly assigns m_8 to an agent, say

[2]However, in the case the *facilitates* relationship saves some execution time, it would seem that this saved time should be allocated.

A_1. Now the scheduler considers the remaining four methods m_2, m_4, m_6, m_{10}. Since m_8 is already in A_1's schedule, heuristic *delay facilitates* will no longer give low rating to method m_6 if it is assigned to agent A_1 this time (but obviously it will still give low rating for m_6 and agent A_2). Heuristic *prefer facilitators* also increases the rating of m_6 for agent A_1. Thus, this pair is chosen and the scheduler puts m_6 at the end of A_1's schedule. There are three methods left. Again due to the same heuristic *delay facilitates*, method m_4 will not get high rating with either of the agents. Heuristic *enforce hard deadline* will give the pair m_2 and A_1 negative rating since the deadline cannot be met. If method m_{10} dramatically increases the quality of method m_4 because of the *facilitates* relationship, m_{10} and agent A_2 will get a higher rating than method m_2 and A_2. But let's assume that facilitation is not that strong. So m_2 is put into A_2's schedule. Next, method m_{10} will be assigned to A_1, and then method m_4 (clearly neither will be assigned to A_2 since A_1 can start executing them earlier). The schedule is shown in Fig. 2 (dotted lines with arrows represent those *facilitates* relationships that have been taken advantage of), and Table 1 shows the method ratings at each step. It should be noted that in this case the schedule we generated is the optimal. In other cases, the scheduler must use a third stage to improve schedule quality.

scheduled methods (A_1/A_2)	m_2	m_4	m_6	m_8	m_{10}
	$57/57$	$0/0$	$0/0$	$85/85$	$28/28$
$\{m_8\}/\{\}$	$< 57/57$	$0/0$	$165/0$		$< 28/28$
$\{m_8, m_6\}/\{\}$	$< 57/57$	$0/0$			$< 28/28$
$\{m_8, m_6\}/\{m_2\}$		$0/0$			$28/ < 28$
$\{m_8, m_6, m_{10}\}/\{m_2\}$		$59/ < 59$			
$\{m_8, m_6, m_{10}, m_4\}/\{m_2\}$					

Table 1: Method ratings. A method is rated against each agent, thus $< 57/57$ for method M_2 means it gets rating of 57 if executed by agent A_2, and *less* if executed by A_1. *Less* is used when the quality against each agent is the same, but the method will be executed *later*. For simplicity, a method gets 10 percent increase of quality for each method it facilitates. The actual heuristic *prefer facilitators* used in our implementation is a bit more complicated.

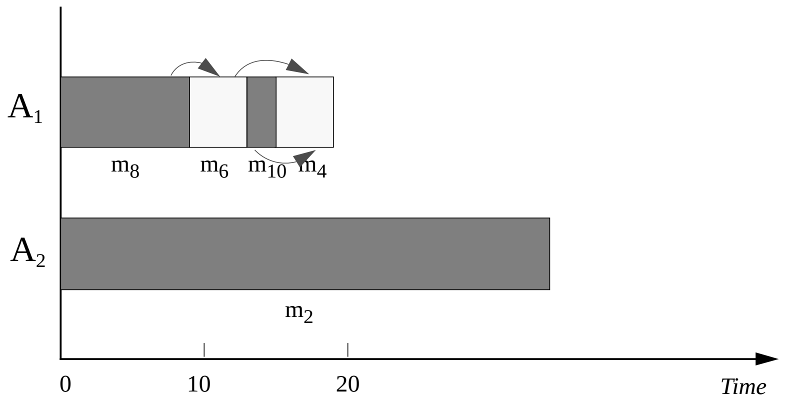

Figure 2: A schedule.

5 Repairing schedule

The objective function of our parallel schedule is based on *schedule result quality* and *schedule duration*. Informally, this function describes two ordering preferences:

1. As long as all the task deadlines are met, the schedule which derives higher overall task qualities is preferred.

2. If two schedules derive the same task qualities, the one with shorter total execution time is preferred.

After a schedule is created, the parallel scheduler uses a repairing stage to improve its quality. Some improvement can be expected in most cases since the scheduler at the generation stage was mainly concerned with how to distribute methods among agents, and did not try to find the optimal time to execute a method. A new method was simply appended to an agent's schedule so that the agent would execute it as soon as possible. The scheduler did not consider what would happen if the agent waits some time and then starts the method to take advantage of the possible *facilitates* relationship.

The parallel scheduler uses two mechanisms to repair a schedule: *postpone* and *switch*. Postponing a method in an agent's schedule means to insert some idle time so that there is a delay before the agent executes that method. Switching a method means that to locate another agent so that the method can execute earlier.

The scheduler uses these two mechanisms to search for a good starting time for each method.

The repair procedure is the following.

1. Find all methods in the parallel schedule, put them into a list L. Sort the list by each method's start-time in non-increasing order, thus the one which has the last starting time is at the beginning of list L.

2. For each method M in L, the scheduler performs one of the three actions:

   (a) If method M can be postponed for some time to increase the parallel schedule quality, *postpone* it. Then consider the next method in list L. We will describe how the scheduler determines whether to postpone a method in section 5.1.

   (b) If method M cannot be postponed in one agent's schedule, but if switching it to another agent would increase the parallel schedule quality, try to *switch* it. Then consider the next method in list L. It should be noted that *postpone* and *switch* represents two distinctive ways of refining the parallel schedule. The former adjusts the starting time of some methods from the local view of each agent. During this process, the method order of each agent's schedule will not be changed. The latter one, on the other hand, considers the parallel schedule as a whole. Particularly, it may switch a method from one agent to another so that this method is executed before the one which was in front of it in the old agent's schedule is finished. In this case, the method order is changed but only if the overall quality of the parallel schedule increases. We will discuss *switch* repairing approach in section 5.2.

   (c) Otherwise, skip the current method M and consider the next one in list L.

3. If step 2 results in a change of parallel schedule quality, go back to step 1. Otherwise, stop repairing procedure.

It is clear that our repairing algorithm is an incremental process. Step 2a and 2b guarantees that the new parallel schedule has a higher quality than the old one. As a matter of fact, the algorithm is a form of hill-climbing, it stops whenever it reaches a local maximum. This seems rather disappointing, and especially if the original parallel schedule had very poor quality. However, since the parallel schedule produced at the generation stage is compatible with that from the design-to-time algorithm, we believe that its quality is already acceptable. Thus any improvement by reaching a local maximum is still statistically significant (i.e., the improvement is not due to a floor effect). In fact, since the *postpone* and *switch* mechanisms directly address the key problem which may affect the parallel schedule quality, a good improvement can be expected.

In the following two subsections, we will describe how *postpone* and *switch* mechanisms work, and how the scheduler chooses one from the two.

5.1 Postpone a method

The reason to postpone a method in an agent's schedule is simple — it starts too early to take advantage of *facilitates* task interactions. Thus, for a method M in list L, the scheduler first decides whether it can be facilitated by any other methods in the parallel schedule. This can be easily done in the TÆMS environment since all the task relationships are explicitly represented. If there exists *facilitates* relationship from which M can be benefited, the scheduler then determines how much delay D is needed. The calculation is simple, since for each method the scheduler knows its current scheduled start-time and its execution duration. This information allows the scheduler to find out the earliest start-time for method M, which is the time when all the facilitating methods are expected to have finished. The delay D of M is just the difference between the earliest start-time and its current scheduled start-time.

Before the scheduler actually postpones M for D time, it must decide whether such a change will increase the parallel schedule quality. This requires the scheduler to find out what methods in the parallel schedule would be affected by the delay of M. Clearly, the methods that are directly affected are those in L which take advantage of the *facilitates* relationships from M, and those that are in the same agent's schedule with M and but start after it finishes. Finding those facilitated methods is not difficult, the scheduler can check each of the *facilitates* relationships from M and see whether its beneficiary is in list L. However, to identify all the methods in L which are affected by the delay of M requires the scheduler to recursively consider each that has been identified. This might be very computationally expensive. Furthermore, for each affected method the scheduler wants to know whether it can be postponed accordingly so that it can continue to benefit from the *facilitates* relationship, or at least to meet its deadline if there is no *facil-*

itates relationship between it and method M. Clearly, if there are some methods that benefited by M but are not any more, or if some can no longer meet their deadlines, the quality loss resulted should be subtracted from the gain by the delay M to get the parallel schedule quality. To reduce the computation cost, we use this algorithm to identify those methods which are affected by the delay of method M, and more importantly, to split them into two groups — one that can be delayed and continue to benefit from method M, and one that cannot delayed.

1. Initialize two lists T and F. T is a set of methods that can be delayed with M, while F is a set which cannot be delayed with M. C is the other list used, it denotes a set of methods in consideration and is initialized with all the methods which are directly affected by M in list L.

2. Consider each method N in C.

   (a) If N is also in list T, return the status that it can be delayed.

   (b) If N is also in list F, return the status that it cannot be delayed.

   (c) If N currently does not facilitate any method (i.e., there is no *facilitates* relationships from it, or all the beneficiaries are expected to start before N finishes), put N into list T if delay D will not violate its deadline, or put N into list F if the delay will violate its deadline. Return the corresponding status, and the changed set T or F.

   (d) If N is expected to facilitate some method in the parallel schedule, put all those that are directly affected and are neither in list T nor F at the beginning of list C. Then recursively consider each of them.

   (e) If N is expected to facilitate some method in the parallel schedule, and all of those methods that are directly affected are either in T or F, then consider if the gain of delaying M will offset the loss of facilitating benefits of those in F. If so, and if N will meet its deadline even with the delay D, put N into list T. Otherwise, put it into list F. Return the status of whether it can be delayed or not, and the new list of T or F.

This algorithm can be implemented in two procedures, with step 1 and 2 constituting the first, and $2a$ to $2e$ making up another. Both can be implemented recursively, with the second one always passing back the status and the changed new list C, T or F. This algorithm is like a depth-first search, but to avoid redundant computations, the nodes that have been visited are kept in list T or F so that the search can later reuse any previous results. Since the top level repairing procedure may apply this algorithm to the same method many times, we also build and use a table to remember the maximal delay a method can bear without decreasing the whole parallel schedule during this process. Thus if the scheduler later needs to know whether a method can be further postponed, it can quickly get the answer by comparing the maximal and the desired delay time. Caching results can greatly reduce the computation cost in our application domain, since one method (or task) can be facilitated by many others. For example, task T_b is facilitated by both T_c and T_e.

After the scheduler has determined that it is beneficial to postpone method M, the scheduler will use a similar algorithm to visit all the affected methods in the parallel schedule. But this time it will postpone the start-time of each methods in list T to reflect the delay of M.

We still use the task groups in Fig. 1 as our example to illustrate this *postpone* repairing mechanism. Since the scheduler gets the optimal schedule (see Fig. 2) at the generation stage, we make a small change here. We assume that method m_2 only needs 12 instead of 34 time units for execution. This change will result in a new schedule (shown in Fig. 3), since now m_{10} can be finished earlier if it is executed by agent A_2 than by A_1. The postpone mechanism starts by identifying

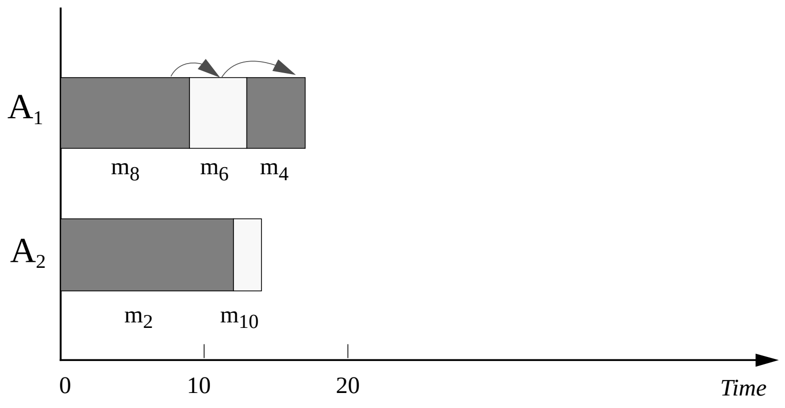

Figure 3: Method M_4 needs to be postponed.

those methods that do not benefit from all of the facilitates relationships. It can be seen from Fig. 3 that method m_4 is the one since it is executed before m_{10} is finished. Clearly, 1 time unit delay is what m_4 needs. The scheduler then checks if such delay would affect the other methods in the schedule. Since m_4 is the last one to be executed, the scheduler cannot find any. Thus a 1 unit idle time is inserted into agent A_1's schedule, as shown in Fig. 4. Clearly, this is also an optimal quality parallel schedule.

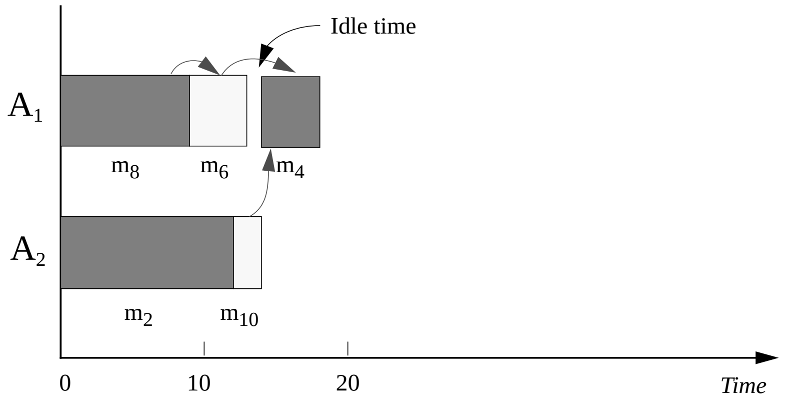

Figure 4: Postponing a method.

5.2 Switch agent

From Section 5.1, it can be seen that postponing a method M in one agent's schedule to take advantage of the task relationships is not always beneficial to the parallel schedule quality. The reasons can be any combination of the following two cases:

1. Some methods which benefited from the *facilitates* task relationships from method M no longer do so due to the delay of M, or they can no longer meet their deadlines.

2. Some methods which do not benefit from the execution of method M can no longer meet their deadlines due to the delay. These methods are those behind method M in the agent's schedule.

In any case, if the quality loss is greater than the gain, the scheduler will not try to postpone M.

But if the sole reason not to postpone method M is some methods behind it cannot meet their deadlines any more, and these methods are not expected to take advantage of the *facilitates* relationships from M in the current schedule of the agent, switching M to another agent so that it will start later may be possible. In this way, there will be no change to any method behind M in the same agent's schedule.

The simplest way to switch another agent for a method M is to let the parallel scheduler to reschedule the method. That is, the scheduler considers M in the presence of the current parallel schedule with M removed. If the new parallel schedule results higher quality, replace the current one with the new.

To illustrate how the two repairing mechanisms work together, we modify the previous example shown in Fig. 1 and Fig. 3 (see Section 5.1). But increasing the duration to execute method m_{10} to 6 time units. Furthermore, we add a new task T_f to the task group `TG2` in Fig. 1, and assume there is only one method m_{11} (with $q = 25$, $d = 5$) to achieve this task. Suppose T_f is *enabled* task T_e, meaning that it can be executed only after T_e is done. Fig. 5 shows the initial parallel schedule before any repairing is done. Method m_{11} is assigned to agent A_1 instead of A_2

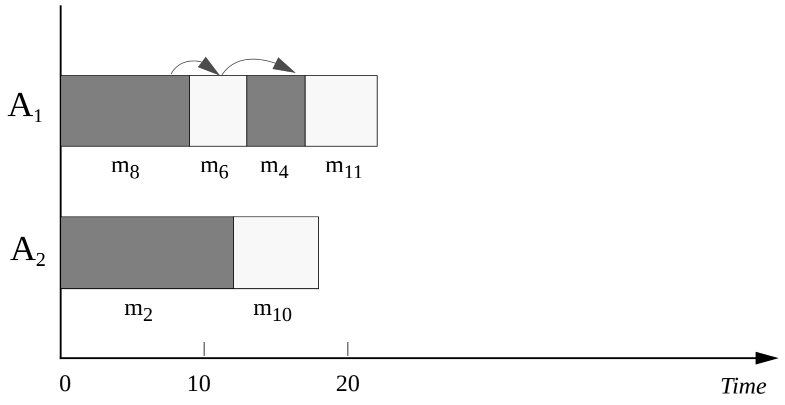

Figure 5: Needs switch agent and postpone.

since it can be executed earlier. Similar to the previous example, the scheduler will try to postpone method m_4 for 5 time units to take advantage of the facilities relationship from m_{10}. But by checking the schedule of agent A_1, the scheduler finds method m_{11} won't be finished in time if such delay happens. Since m_{11} is not required or desired to be executed after m_4 because of task relationships, the scheduler will try to assign it to another agent. A_2 is the only choice. After the switching, the scheduler can then postpone m_4. The final schedule is shown in Fig.6.

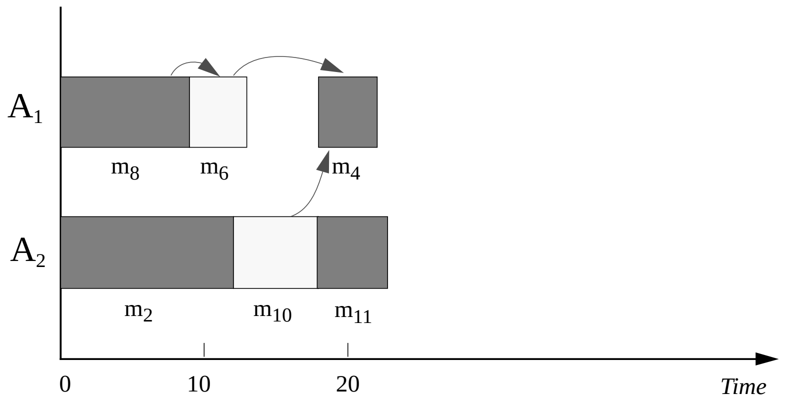

Figure 6: After switching m_{11} and postponing m_4.

5.3 Compact schedule

When a method is switched from one agent to another, the time slot it occupied in the schedule of the former agent becomes idle time. In order to keep schedule duration short, the scheduler needs to check whether it is possibile to utiltize this idle time by shift some methods forward to execute them earlier. It is clear that only the method that immedately follows the idle time should be considered at first. This method M is then moved forward as far as it does not lose any benifit already taken from the task relationships. Any movement of this method M, however, can result in a list L of method that starts after the original idle time slot, and follows some amount of idle time (so it is move-able) in the parallel schedule. To consider how to shift foward the methods in list L, three heuristics are used in order.

1. Any shift of a method cannot result in any lose of the benefits it has taken from any task relationships.
2. If more than one can be shifted forward, the one that causes the largest movement in the longest schedule is preferred. That is, the scheduler perfers to reduce the duration of the whole parallel schedule, instead of a local schedule of some agent.
3. The method that starts earlier is preferred.

Of course, the new method is put into L whenever its predecessor is shifted forward. The scheduler repeatedly use the three heuristics to move methods in L till no shift can be made.

6 Experiment

We have implemented our parallel scheduler on TÆMS testbed. The testbed allows us to adjust various parameters, such as the number of computational agents, the number of the task groups, the chance that each kind of task relationship can occur. Since we haven't implemented an exclusive-search scheduler to generate optimal parallel schedules, it is unclear at this moment how our scheduler performs statistically. In a number of (relatively simple) randomly generated tests whose optimal schedules are known, our scheduler performs very well. We are undergoing more experiments to analyze the performance of the scheduler with or without the repairing stage.

7 Related work

Our scheduling problem would be quite like the ones that are widely studied if there are no task relationships that would affect task execution durations or their result qualities[1, 5, 6]. The presence of these task relationships imply that not only all the methods need to be scheduled so that they can be executed before their deadlines, they should also be ordered in a way to achieve shorter schedule duration and better overall result qualities. Below we describe the related work that does consider some relationships between tasks, but none as realistic and the ones we use.

Fox and Smith treats scheduling as a constraint-directed search. Their system ISIS [9] implements a hierarcial scheduling approach. Orders (tasks) are selected one by one to have their operations (methods) scheduled according to their priority. Decisions regardsing the selection of resources (like computational agents) simply attempt to satisfy the constraints relating to the order (to properly sequence the operations and meet the deadlines). Their constraints are different from the ones that result from the *facilitates* or *hinders* task relationships, in the sense theirs must be satisfied in a good schedule while ours can be violated if it is impossible to satisfy them. If not all the constraints can be satisfied, the flexibility from task relationships thus requires the scheduler to decide what constraints should pursue.

Hildum implements a knowledge-based system for solving dynamic resource-constrained scheduling problems in [10]. His scheduling algorithm expliots the *flexibility* properties (like *earliest-start-time*) of tasks to allow the schedule to be adaptable to the changing environment (like the arrival of some new tasks). It uses least-commitment decision making technique to preserve maneuverability by explicitly incorporating slack time into the developing schedule. The preserved slack time can be used (e.g., shifting task) later to adjust the schedule. Our scheduler does not preserve slack time in the schedule, it inserts them only when necessary.

GERRY [11] developed by Zweben *et al.* uses constraint-based iterative repair to schedule and reschedule the tasks of a plan according to temporal constraints, milestones, resource constraints and state constraints. Some of their constraints are similar to or can be represented as our task relationships. Their scheduling and repairing algorithm iteratively modifies the schedule via some basic actions like inserting an achiever, shifting a task forward or backward. Our work employs a constructive method to build draft schedule. This reduces the amount of iterative modifications needed at the repairing process.

Ramamritham and Stankovic in [12] uses a heuristic incremental approach to build schedule for multiprocessor systems. Their work focuses on how to guarantee that all the tasks meet their deadlines. Their assumption that tasks are independent is different from ours, though we believe that their heuristics (based on task arrival time, deadline, worst case processing time) can be used to generate some meta-level information (like what is the maximal load for an agent) for our scheduler.

8 Summary

This paper presented a heuristics parallel scheduler based on task structures. We believe identifying task relationships and exploiting them is crucial in building high quality schedules for real-life applications. Since in most cases task relationships can be rather complicated, a scheduler can be easily trapped into a large amount of backtracking in order to find a good schedule. Our scheduler avoids such traps by using a conservative non-backtrack heuristic search at the generation stage. It is conservative, since at this stage it only *encourages* the benefits of task interactions but does not aggressively *enforce* (*enables* relationship is an exception) them, like the examples illustrated in Fig. 3 and Fig. 5. This allows our scheduler to quickly generate a draft schedule. The two mechanisms *postpone* and *switching agent* that our scheduler uses to refine a draft schedule focus on how to take advantage of those task relationships that have not yet been taken.

We did not specify what to do if a given set of methods (alternative) cannot be scheduled to meet the desired quality. It is expected that the repairing approach developed by Garvey and Lesser in [8] can be used in this parallel scheduler. Their approach allows the scheduler to regenerate the alternative by switching some methods that causes the failure or the low quality of the schedule. For example, if the scheduler finds that if method m_{10} (in Fig. 1) had used fewer computation time the overall quality of the parallel scheduler could be increased by 50. Then the scheduler will use method m_9 to replace m_{10} since they achieve the same task T_e.

The performance of our scheduler can also be improved by exploiting meta-level information. More specifically, we expect that the scheduler can estimate what is the most promising change (or what is the worst point) in the schedule before it actually starts the repairing process. It is clear such information will allow the scheduler to quickly get a good schedule, and to compare it with other schedulers working in a similar environment.

9 Acknowledgment

We appreciate the valuable reviews from Kathryn McKinley. Our experiment is based on the testbed developed by Keith Decker and Alan Garvey. Alan Garvey has also helped us to form and implement the ideas.

References

[1] Wesley W. Chu and Lance M-T. Lan. Task allocation and precedence relations for distributed real-time systems. *IEEE Transactions on Computers*, c-36(6):667–679, June 1987.

[2] S. H. Bokhari. Dual processor scheduling with dynamic reassignment. *IEEE Transactions on Software Eng.*, SE-8:401–412, July 1979.

[3] G. S. Rao, H. S. Stone, and T. C. Hu. Assignment of tasks in a distributed processing system with limited memory. *IEEE Trans. Comput.*, C-28:291–299, April 1979.

[4] P. Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Trans. Comput.*, C-31:41–47, January 1982.

[5] Gilbert C. Sih and Edward A. Lee. Declustering: A new multiprocessor scheduling technique. *IEEE transactions on Parallel And Distributed Systems*, 4(6):625–637, June 1993.

[6] Gilbert C. Sih and Edward A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–186, February 1993.

[7] Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex computational task environments. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 217–224, July 1993.

[8] Alan Garvey and Victor Lesser. Design-to-time scheduling with uncertainty. Technical Report TR95-03, Department of Computer Science, University of Massachusetts, January 1995.

[9] Mark S. Fox and Stephen F. Smith. Isis—a knowledge-based system for factory scheduling. *Expert Systems*, 1(1):25–49, 1984.

[10] David W. Hildum. Flexibility in a knowledge-based system for solving dynamic resource-constrained scheduling problems. Technical Report Umass CMPSCI TR94-77, University of Massachusetts, September 1994.

[11] Monte Zweben, Brian Daun, and Michael Deale. *Scheduling and Rescheduling with iterative repair*, pages 241–256. Morgan Kauffman, 1994.

[12] Krithi Ramamritham and John A. Stankovic. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions of Parallel and Distributed Systems*, 1(2):184–194, April 1990.