Scheduling Dynamic Workflows onto Clusters of Clusters using Postponing

Sascha Hunold, Thomas Rauber, Frédéric Suter

Scheduling Dynamic Workflows onto Clusters of Clusters using Postponing

Sascha Hunold [#], Thomas Rauber [#], Frédéric Suter [*]

[#] *Department of Mathematics and Physics*
University of Bayreuth, Germany

[*] *Nancy Université / LORIA*
UMR 7503 CNRS - INPL - INRIA - Nancy 2 - UHP, Nancy 1

Abstract

In this article, we revisit the problem of scheduling dynamically generated directed acyclic graphs (DAGs) of multi-processor tasks (M-tasks). A DAG is a basic model for expressing workflows applications where each node represents a task of the workflow. We present a novel algorithm (DMHEFT) for scheduling dynamically generated DAGs onto a heterogeneous collection of clusters. The scheduling decisions are based on the predicted runtime of an M-task as well as the estimation of the redistribution costs between data-dependent tasks. The algorithm also takes care of unfavorable placements of M-tasks by considering the postponing of ready tasks even if idle processors are available. We evaluate the scheduling algorithm by comparing the resulting makespans to the results obtained by using other scheduling algorithms, such as RePA and MHEFT.

1. Introduction

PC clusters have become mainstream for running very time-consuming parallelized applications. Nowadays, there are many installations of those PC clusters available to developers and researchers. The computational power of the parallel platform is the limiting factor in obtaining a certain result quality (short time, high detail) of simulation applications. An efficient utilization of multiple clusters can help to improve the quality of the results significantly.

Many applications can be represented as workflows which can be modeled as DAGs. Each node in the DAG represents an executable task and each directed edge represents a data or a control dependency. The mixed-parallel approach for parallelizing an application where data-parallel nodes are executed concurrently often leads to faster execution times than pure data-parallel applications [3]. A task which is assigned to a number of available processors is called a multi-processor task (M-task). The scheduling of such mixed-parallel applications has been studied for homogeneous platforms [8]. More recently, heterogenous platforms (grid environments, clusters of clusters) have become the target for scheduling mixed-parallel applications [1, 6]. The previous approaches assume that the entire DAG is known at all time. However, DAGs can also be created dynamically during execution time. For instance, in a recursively implemented algorithm a task may create new sub-tasks if certain criteria are satisfied and so, the workflow is not known beforehand.

Scheduling dynamically generated DAGs on heterogenous clusters of clusters is challenging. Assigning too many processors to one task may prevent other tasks from being executed, and moreover, the scalability of tasks being limited, assigning many processors does not necessarily lead to a high performance efficiency.

A primary goal of scheduling algorithms is to achieve a small makespan that is the time required to execute all the nodes of the DAG. The reduction of the communication costs plays an important role for achieving small makespans. In general, a task receives input data, performs some computation, and then produces output data. Input and output data (e.g., matrices or vectors) are distributed across processors. Data redistribution is required when data produced by a source task is required as input for a target task. Depending on the processors assigned to source and target tasks, the communication costs vary tremendously, especially when the tasks are mapped onto different clusters.

In previous work, we have introduced the TGrid framework which is an runtime environment for the execution of M-tasks in clusters of clusters [4]. It supports the dynamic creation of M-tasks, leading to aforementioned dynamically created DAGs. M-task programs can be used to implement scientific applications which are formulated recursively, e.g., pre-defined convergence criteria decide whether new tasks are generated or not. A first scheduling algorithm that produces

978-0-7695-3156-4/08 $25.00 © 2008 IEEE
DOI 10.1109/CCGRID.2008.44

satisfying makespans for dynamic DAGs (ReP algorithm) was proposed in [5]. For RePA, it is assumed that the data distribution of tasks is unknown to the scheduler and therefore the redistribution costs cannot be estimated. Since RePA does not account for redistribution costs and attempts to use all idle processors in a heterogeneous system, long running tasks (not malleable) could get scheduled to a small set of processors which delays the execution of successor tasks.

In this article, we extend the RePA approach by considering the time for performing data redistribution between subsequent tasks and also propose a postponing strategy to avoid assigning a small number of processors to computation intensive tasks. We evaluate the performance of the new scheduling algorithm by running a number of simulations using SimGrid [2].

This paper is organized as follows. Section 2 points out the problem of scheduling dynamically generated DAGs and introduces the ReP algorithm. In Section 3 the DMHEFT algorithm is proposed which extends the ReP algorithm by accounting for redistribution costs and adding postponing strategies. Section 4 presents the experimental evaluation of DMHEFT. Section 5 discusses related work and Section 6 concludes the article and outlines future work.

2. Scheduling dynamic DAGs

In this section, we state the scheduling problem for multi-processors tasks in a heterogeneous environment and outline our abstraction model. We also recall the ReP algorithm and discuss performance related drawbacks of this approach in certain situations.

We consider a computational platform which consists of k clusters $C_i, i = 1,\ldots,k$. Each cluster C_i contains p_i processors of the same type and speed. All clusters are interconnected to a backbone via a single network link and each cluster may have a different connection topology. This is a reasonable setup since todays larger grid environments are mostly a collection of internally homogeneous clusters.

Mixed-parallel applications can be modeled as a DAG $G = (N,E)$, where $N = \{t_i | i = 1,\ldots,n\}$ denotes the set of nodes which represents M-tasks. E denotes the set of edges $\{e_{i,j} | (i,j) \in \{1,\ldots,n\} \times \{1,\ldots,n\}\}$, representing the data-dependencies between the nodes in N. A node is associated with computational costs which are modeled as the number of operations that this task has to execute. Each edge $e_{i,j}$ is weighted by the amount of data (in bytes) that task t_i has to send task t_j.

An execution environment for mixed-parallel application such as TGrid does not have the notion of DAGs. The mixed-parallel application is simply a sequence of M-tasks which are generated during the execution of an application. The arising scheduling problem is similar to the scheduling of batch jobs in grids. The main difference is the existence of data-dependencies between M-tasks. Therefore, the scheduler has to take special care of the communication costs which are involved in the redistribution of data. After the mixed-parallel application has finished its execution in the runtime environment, the call sequence of M-tasks can be represented as a DAG.

Algorithm 1 RePA

```
1: queue (ready tasks) sorted by decreasing computation cost and
   node depth
2: while queue is not empty and clusters not saturated do
3:     node = queue.pop()
4:     c = find target cluster for node
5:     processor nb p = get number of processors(node, c)
6:     processor list l = select processors(node, c, p)
7:     schedule node on c and l
```

Another requirement of our model is the fact that M-tasks must be executed within a single cluster. Assigning an M-task to processors of several different clusters implies an overhead that is practically never regained by shortening the execution time. The computational complexity of the scheduler increases significantly which leads to a slower decision making process.

In previous work, we have introduced the ReP algorithm (*Re*use *P*rocessors *A*lgorithm) as a possible scheduling method to be used within the TGrid framework. Algorithm 1 shows the algorithmic structure of RePA which is extended in this article. The algorithm is always executed when M-tasks have finished execution and new tasks become ready. The queue of ready tasks is sorted by the computation amount and node depth (line 1). The node depth is the number of nodes (predecessors) from the entry task to the ready node. Considering the node depth often avoids the problem of starvation that nodes with low computational costs will be delayed too long and thus, successors of this node will not become ready leading to a smaller degree of task parallelization. Sorting by the computational costs of nodes is a common heuristic for most schedulers (LPT strategy). The 'big' tasks have to be scheduled first and the other tasks are used to 'fill the holes'. This heuristic avoids the case that larger tasks which require multiple processors to get done in an acceptable amount of time will not get executed since the fragmentation of the clusters is too high.

RePA uses three steps to find a suitable assignment of processors to an M-task (lines 4-6). At first, the algorithm determines which cluster is the most suitable for executing the currently considered task. The cluster with the most available computational power is selected, i.e., the cluster which minimizes the finish time

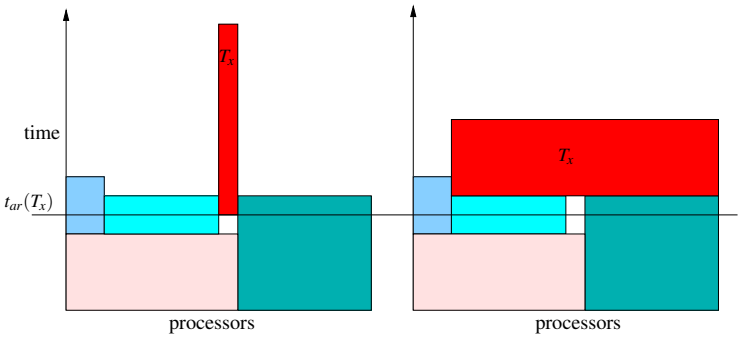

Figure 1. Common problem for dynamic schedulers: placing and executing a task T_x at time $t_{ar}(T_x)$ may lead to an unfavorable schedule if the number of available processors is small.

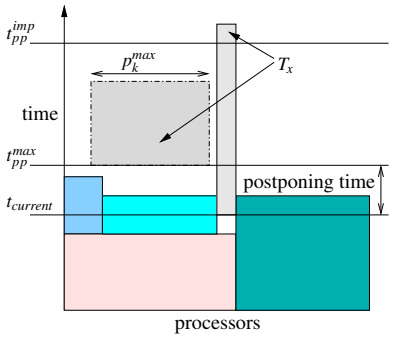

Figure 2. DMHEFT – postponing heuristic for M-task T_x which becomes executable at $t_{current}$.

of the task (EFT strategy). In the second step, the algorithm determines the number of processors which are assigned to this task. This number is computed by taking into account the computational power of the cluster and the computational costs of the other unscheduled tasks. As a result, a task is assigned to a fair share of the available processors which leaves enough room for the other unscheduled tasks. In step three, the algorithm selects this number of processors from the list of available processors. In the selection phase, RePA favors the processors which were assigned to a parent task to reduce the communication overhead for data redistribution.

The experimental evaluation of RePA showed good results in terms of the resulting makespan. Nonetheless, our testing methodology and the simulation tools have evolved significantly since then. A newer simulation testbed, described in Section 4, allowed us to obtain more accurate and realistic scheduling results. Moreover, we have separated and further investigated the cases in which the schedules produced by RePA had most potential for improvements compared the schedules produced by an algorithm with a static DAG approach such as MHEFT.

It clearly turned out that neglecting the communication costs for data redistribution when selecting a cluster can lead to a big communication overhead, especially when data has to be moved across cluster borders. We could also observe that the strategy of using all processors at all time has a big impact on the overall makespan. As we consider moldable tasks, the processors which are assigned to a task are determined before starting the task and cannot be changed during the execution. An illustration of a problematic setup is depicted on the lefthand side of Figure 1. At time t_{ar}, the task T_x becomes ready. In this case, RePA attempts to schedule the biggest task (most operations to perform) on the available processors. As one can see, as a consequence of this decision T_x will be executed on this cluster for a long time. If the mixed-parallel program does not contain a high degree of task parallelism and many tasks are dependent on the result of T_x, the schedule will contain large holes. A possible solution to this problem might be the utilization of a postponing strategy, e.g., task T_x can be postponed until more processors become available, see Figure 1 (right).

In the context of this work, we highlight two essential objectives for reducing the average makespan of schedules of dynamically generated DAGs: (1) accounting for the redistribution costs when assigning a task to a particular cluster and, (2) using postponing strategies to find a good mapping of computational intensive M-tasks to a cluster.

3. Scheduling algorithm using postponing

In this section, we give a detailed description of DMHEFT. The algorithm follows similar principles as RePA and matches the objectives stated above. The name DMHEFT (dynamic MHEFT) was chosen since the algorithm schedules **d**ynamically generated DAGs of **M**-tasks by using an **EFT** approach onto a **h**eterogeneous collection of clusters.

As mentioned, the general ideas behind DMHEFT and RePA are similar, i.e., DMHEFT proceeds in the same sequence of steps as RePA which are: (1) selecting a cluster, (2) determining the number of processors on this cluster, (3) selecting the processors of this cluster. The pseudo code of DMHEFT is shown in Algorithm 2. The algorithm (line 1-4) is always executed when a node becomes executable.

The first improvement is to consider the communication costs involved in the data redistribution process. To achieve this, we have to consider all clusters having at least one idle processor which are potential candidates for the task assignment. For each cluster,

DMHEFT estimates the computation time of the considered task and also estimates the communication time for the data redistribution between this possible task allocation and parent task allocations. The algorithm selects the cluster with the smallest total time consisting of the task execution time and the maximum of all redistribution operations.

Algorithm 2 DMHEFT

```
1: while not done do
2:    node_schedules = schedule( dag, get_ready_nodes() )
3:    for each schedule in node_schedules do
4:       run_task( schedule )
function schedule (dag dag, list ready_nodes)
1: Q= { ready_nodes } ∪ PQ // PQ contains postponed tasks
2: sort Q by incr. depth and decr. computation amount
3: node_schedules = {}
4: while Q is not empty and clusters not saturated do
5:    T_i= pop(Q)
6:    [C_best, lp_best, t_best] = get_best_cluster(T_i)
7:    if C_best > −1 then // clusters not saturated
8:       task_schedule = (T_i, lp_best, t_current, t_best)
9:       if T_i is postponed then
10:         if t_current + t_best < t_pp^imp or t_current ≥ t_pp^max then
11:            // force scheduling
12:            add task_schedule to node_schedules
13:         else append T_i to PQ
14:      else if postponable(T_i, t_best) then
15:         append T_i to PQ
16:      else add task_schedule to node_schedules
17: return node_schedules
function get_best_cluster( task T, queue Q )
1: C_best = −1, t_best = −1, lp_best={}
2: for each cluster C_j do
3:    p_Cj = get_nb_of_processors_for_cluster(T,C_j,Q)
4:    lp_avail = get_processors_for_cluster(C_j, p_Cj)
5:    t_j^est = estimate_time( T_i, lp_avail )
6:    if t_j^est < t_best or -1 then set C_best,t_best,lp_best
7: return [C_best,t_best,lp_best]
function postponable( task T , t_T^est )
1: t_pp^max = t_current + f_pp · t_T^est // max postponing time span
2: t_pp^imp = t_current + f_pi · t_T^est // improvement minimum
3: for each cluster C_k do
4:    p^(t_pp^max) = get_free_processor(t_pp^max, C_k)
5:    p_k^max = f_cu · p^(t_pp^max)
      // only a fraction of the processors might be available
6:    t_i,k^pp = estimate_time(T, p_k^max)
7:    if t_pp^max + t_i,k^pp ≤ t_pp^imp then
8:       return true // postponing will probably work
9: return false
```

The second optimization based on postponing strategies requires more changes. Similar to RePA, the node with the highest computation amount (or smallest node depth) is selected and assigned to the cluster which minimizes the finish time of the task (accounting redistribution costs). As seen in Figure 1, executing a computational intensive task on a small set of processors might increase the resulting makespan of the schedule. Thus, a postponing strategy (function *postponable*) can help to overcome this problem. The herein proposed postponing heuristic uses three parameters depicted in Figure 2. Their values can be computed from the current time $t_{current}$ (the time when a task T becomes executable) and the estimated finish time of this task t_T^{est}. We defined as $t_{pp}^{max} = t_{current} + f_{pp} \cdot t_T^{est}, 0 \leq f_{pp} < 1$, the maximum amount of time that a task can be postponed. We expressed by $t_{pp}^{imp} = t_{current} + f_{pi} \cdot t_T^{est}, 0 < f_{pi} \leq 1 \wedge t_{pp}^{imp} \geq t_{pp}^{max}$, the time by which a postponed task must be completed to justify postponing. And finally $p_k^{max} = f_{cu} \cdot p^{t_{pp}^{max}}, 0 < f_{cu} < 1$ represents the number of processors which can be assigned the a task at time t_{pp}^{max}, where $p^{t_{pp}^{max}}$ is the number of free processors at this time. f_{pp}, f_{pi} and f_{cu} are adjustment variables whose values will be given later.

After the best cluster and the processor list for the currently active task T has been determined (lines 1-8), the scheduler calls the *postponable* function (line 14) to estimate the likeliness that postponing T can reduce the execution time of T. The rationale behind the three postponing parameters is as follows. The value of t_{pp}^{max} denotes the time by which a postponed task must be executed. For this time, the scheduler tries to predict the number of free processors $p^{t_{pp}^{max}}$ on this cluster by scanning the estimated finish time of the currently running tasks. Since it is unlikely that at time t_{pp}^{max} only T will be ready for execution the number of free processors which can be assigned to T is limited to a fraction of $p^{t_{pp}^{max}}$ and is denoted by the variable p_k^{max}. The values help the scheduler to predict the execution time of T at time t_{pp}^{max} if p_k^{max} processors could be assigned to T. Due to the fact that the number of executable tasks at time t_{pp}^{max} is unknown, the number of idle processors is hard to predict. Therefore, a task is only postponed if it could lead to a significant performance gain. This performance gain is expressed by the time t_{pp}^{imp}, i.e., only if the postponing, the redistribution, and the execution of T is predicted to be done by t_{pp}^{imp} the scheduler will postpone T.

A *postponable* task is removed from the list of ready nodes and appended to a list of postponed nodes (line 15). The algorithm in function *schedule* outlines the postponing strategy with an upper postponing bound (line 10). If time t_{pp}^{max} is up task T is scheduled on its currently best known allocation. We experimented with two cases: (1) postponing using a hard upper bound, and (2) the postponing is not bounded by a threshold (lines 9-13 removed). In case (1), a task is executed when t_{pp}^{max} is up. In the second case, the time t_{pp}^{max} is only used to estimate if further postponing would provide a benefit to the execution time. If so, a task can be postponed multiple times, as long as the postponing condition (function postponable) evaluates to true.

4. Experimental results

To evaluate the quality of DMHEFT, we performed a series of simulations using different configurations of platforms and DAGs. We consider randomly generated application graphs to model a variety of application classes. We have used the same DAG generation framework as in [6]. Each of the generated edges is associated with communication costs which are specified by the data size of the nodes. That pinpoints the size of the data but not the communication pattern used to simulate the data redistribution. Therefore, we assume for the data redistribution that data (matrices) is distributed in a block-partitioned fashion onto the processors of the tasks. The communication pattern (the number of messages which must be transferred) is created by computing the overlapping of the output and input matrices of the nodes. In total, 144 DAGs consisting of 25, 50, 75, or 100 nodes were generated. These DAGs are scheduled onto 40 different grid platforms. Each grid platform consists of either 4 or 8 clusters, and each cluster contains processors has a computation performance between 1 GFLOPS and 1.5 GFLOPS. For the cluster interconnection network, a 10 Gbit Ethernet network with a latency of 100 μs is considered.

All experiments have been carried out using the SimGrid toolkit [2] (revision 4988) which provides a testbed with all necessary functionality to simulate mixed-parallel programs in a heterogeneous distributed environment. We have used the `ptask_L07` workstation model which considers network contention and also takes network latencies into account.

Before DMHEFT can be compared to the other algorithms, we first have to obtain suitable values for the three postponing parameters f_{pp}, f_{pi}, and f_{cu} introduced in Section 3. Experiments have been performed with all possible combinations of pre-defined values for each parameter. The following values have been chosen for the each postponing parameter for the simulations: $f_{pp} = 0.2$, $f_{pi} = 0.8$, and $f_{cu} = 0.4$.

As mentioned before, we have evaluated several strategies for DMHEFT which are: (1) consider the redistribution costs only, no postponing, (2) use a postponing strategy with a threshold (hard limit), and (3) use a postponing strategy without threshold. Not surprisingly, the version that does not use a postponing strategy produced the longest schedules in the average. However, the scheduling method without using a postponing threshold showed better results in the average and was therefore used in the comparison experiments.

Figure 3 shows the average makespan of the schedules produced by DMHEFT, RePA, and MHEFT. Similar trends were noticed for 4 and 8 clusters. The schedules produced by MHEFT are shorter in the average which could be expected since MHEFT follows a static approach where the entire DAG is known at all times. More importantly, the results for DMHEFT show a huge improvement compared to RePA as its average makespan is at least 34% better.

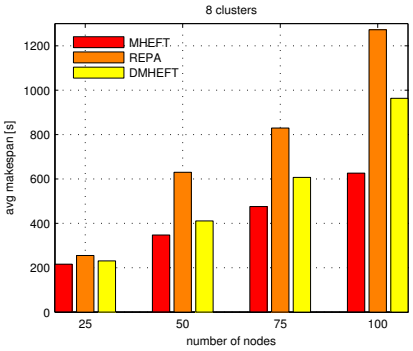

Figure 3. Average makespan produced by MHEFT, RePA, and DMHEFT.

Table 1. Pair-wise comparison of the scheduling algorithms for a total of 5760 tests.

		MHEFT	REPA	DMHEFT	combined
MHEFT	better	xxx	4932	4242	79.64 %
	equal		26	54	0.69 %
	worse		802	1464	19.67 %
REPA	better	802	xxx	1533	20.27 %
	equal	26		185	1.83 %
	worse	4932		4042	77.90 %
DMHEFT	better	1464	4042	xxx	47.80 %
	equal	54	185		2.07 %
	worse	4242	1533		50.13 %

Table 1 presents another view of the experimental data. This table gives a one to one comparison for each pair of scheduling algorithms. The use of MHEFT results in better schedules in about 79% of all cases. When comparing only the dynamic scheduling approaches, DMHEFT clearly outperforms RePA in 4042 tests (70%). The relative high number of better schedules obtained using RePA (1533) is the result of different strategies for placing the first task on each cluster. If a cluster is empty, RePA allocates all available processors for the first scheduled task. In contrast, DMHEFT allocates only the 'fair share' for a task leading to a lower communication overhead for small DAGs (25 nodes) compared to RePA. In comparison to MHEFT, DMHEFT produces better schedules in more than 25% of all cases which is noteworthy, especially since DMHEFT can only predict future allocations.

It is also interesting to examine the relative quality of the schedules produced by an algorithm using the degradation from best metric. The first line of Table 2 presents the average over the total number of experi-

Table 2. Average degradation from best.

	MHEFT	REPA	DMHEFT
avg over all exp.	5.2%	98.7%	36.2%
# not best	1888	5200	4338
avg over # not best	16.0%	109.4%	48.1%

ments (5760) of the percent relative difference for each experiment between the makespan achieved by an algorithm and the best makespan achieved. We can see that MHEFT produces schedules which are very close to the best schedule and that for dynamic DAGs, the schedules produced by DMHEFT are in the average more than 60% shorter than those of RePA. It could by criticized that this averaging method favors an algorithm which is often the best. Thus, we provide a second averaging method in which the sum is divided by the number of experiments where a given algorithm did not produced the shortest schedule. This number is shown in the second line of Table 2 while the third line presents the consequent degradation from best. We can see that MHEFT is still close to the best schedule produced as MHEFT knows the entire DAG before making the first decision. Knowing that MHEFT produces often best schedules or at least very close schedules, the comparison of the degradation from the best of RePA and DMHEFT has a significant meaning: a schedule produced by DMHEFT is in average about 50% slower than a static approach, and since RePA produces schedules which are twice as long as the best, it can be stated that DMHEFT clearly improves the schedule quality for dynamic DAGs.

5. Related work

Existing algorithms for scheduling mixed-parallel programs are based on static DAGs. A number of scheduling algorithms have been designed for the case of homogeneous platforms [8]. An overview and discussion of mixed-parallel task scheduling algorithms for heterogenous environments is provided in [6]. Recently, the Δ-CTS [9] has been proposed which attempts to increase the effective degree of task-parallelism of a DAG by relaxing the critically classification of ready tasks (bottom levels). An algorithm for dynamic scheduling of directed graph-based workflows has been presented in [7]. Even though the workflow model used in this paper is similar to the M-task graphs of TGrid, the main objective of both approaches is different. Instead of scheduling a static workflow representation on a highly heterogeneous and dynamic grid environment, the TGrid program itself can be treated as a dynamic description of a mixed-parallel workflow which is executed in a static environment consisting of several homogeneous clusters.

6. Conclusions

In this article we have presented the DMHEFT algorithm for scheduling dynamic workflows. Our approach takes redistribution costs between dependent tasks into account. To achieve a small makespan the algorithm also features several postponing heuristics which help to avoid unfavorable task placements. An evaluation of the scheduling algorithm has shown that DMHEFT outperforms RePA in most cases and is competitive to static approaches like MHEFT even with decisions based on the task scheduling history. In future work, we will investigate the performance of DMHEFT in a dynamic environment in which the predicted and actual execution times of an M-task may differ. We also plan to extend the approach to multiple DAGs using fairness policies as suggested in [10] and finally to implement DMHEFT and RePA within the TGrid scheduling component for a real world evaluation.

References

[1] H. Casanova, F. Desprez, and F. Suter. From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling. In *the 10th Int. Euro-Par Conf.*, volume 3149 of *LNCS*, 2004.

[2] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th International Conference on Computer Modeling and Simulation*, Mar. 2008.

[3] S. Chakrabarti, K. Yelick, and J. Demmel. Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs. *JPDC*, 47(2):168–184, 1997.

[4] S. Hunold, T. Rauber, and G. Rünger. TGrid – Grid Runtime Support for Hierarchically Structured Task-parallel Programs. In *the 5th Int. HeteroPar Workshop*, 2006.

[5] S. Hunold, T. Rauber, and G. Rünger. Dynamic Scheduling of Multi-Processor Tasks on Clusters of Clusters. In *the 6th Int. HeteroPar Workshop*, 2007.

[6] T. N'Takpé, F. Suter, and H. Casanova. A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms. In *the 6th Int. Symp. on Parallel and Distributed Computing*, 2007.

[7] R. Prodan and T. Fahringer. Dynamic Scheduling of Scientific Workflow Applications on the Grid: A Case Study. In *the 2005 ACM Symposium on Applied Computing*, pages 687–694. ACM Press, 2005.

[8] A. Rădulescu and A. J. C. van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *the 15th International Conference on Parallel Processing (ICPP)*, pages 69–76, 2001.

[9] F. Suter. Scheduling Δ-Critical Tasks in Mixed-Parallel Applications on a National Grid. In *8th Int. Conf. on Grid Computing (GRID 2007)*, 2007.

[10] H. Zhao and R. Sakellariou. Scheduling Multiple DAGs onto Heterogeneous Systems. In *the 15th Heterogeneous Computing Workshop (HCW)*, 2006.