Comparison of Batch Scheduling for Identical Multi-Tasks Jobs on Heterogeneous Platforms

Sékou Diakité, Jean-Marc Nicod and Laurent Philippe

***Abstract*—In this paper we consider the scheduling of a batch of the same job on a heterogeneous execution platform. A job is represented by a directed acyclic graph without forks (intree) but with typed tasks. The execution resources are distributed and each resource can carry out a set of task types. The objective function is to minimize the makespan of the batch execution. Three algorithms are studied in this context: an on-line algorithm, a genetic algorithm and a steady-state algorithm. The contribution of this paper is on the experimental analysis of these algorithms and on their adaptation to the context. We show that their performances depend on the size of the batch and on the characteristics of the execution platform.**

***Index Terms*—Batch scheduling, grid computing, heterogeneous platform, on-line scheduling, steady state scheduling, genetic algorithm.**

I. Introduction

In this paper we are interested in scheduling one batch of identical jobs on a heterogeneous computing grid. Each job of the batch is an application described by a Directed Acyclic Graph (DAG) without forks (intree) and executed independently from the others jobs. The originality of the work is that each host is able to perform only a subset of the application tasks but several hosts are able to carry out the same task. An example is a work-flow to process a set of data in several steps –each step could be a filter applied to an image– on a grid where the softwares installed on the hosts differs –only some filters are available on a host. Our objective is to minimize the makespan of the batch execution, but, as there is no direct optimal solution to the problem, we evaluate different solutions by simulating them.

In this context, the main contribution of this paper is the comparison of three scheduling techniques for medium sized batches: a simple on-line scheduler, a steady-state scheduling technique and a standard heuristic based algorithm, designed for DAG scheduling. We also show that the platform architecture may affect the performances of the schedule.

The organization of the paper as follow. The second section gives a formal definition of the context and of the scheduling problem. The third section is dedicated to the related works. Then, in the forth section, we select three algorithms and we explain how we adapt them to our context. The algorithms comparison is done experimentally, so the simulation implementation is described in the fifth section. The results and the comments on the different scheduling techniques are given in the sixth section, before the conclusion.

LIFC/ INRIA GRAAL, 16 route de Gray 25000 Besançon, France, [diakite,nicod,philippe]@lifc.univ-fcomte.fr

II. Context definition

The platform is composed by a set of processors which communicate through network links. It is represented by an undirected graph: $PF = (P, L)$, where the vertices are the n processors p_i ($p_i \in P : i \in [1..n], n = |P|$) and the edges are the network links l_i between these processors.

On this platform, we execute one batch of identical jobs. A batch B of length m is defined as a set of m identical instances J_j of the job J such as $B = \{J_j : j \in [1..m]\}$. The job J is composed of several tasks with dependency constraints and is represented by a DAG: $J = (T, D)$ where the vertices T are the tasks T_k and the edges D are the dependency constraints between the tasks. Note that there will be several instances of task T_k belonging to different instances J_j of job J.

Tasks cannot be executed on every processor. Each processor P_i implements a limited set of functions (libraries) to carry out the tasks. We define the set of function F_i as the set of tasks T_k that the processor P_i is able to perform. Let F be the set of all the functions used by the job J thus $\cup_i F_i = F$ where $i \in [1..N]$. The time T needed to execute a task T_k on different processors P_i is not uniform: $T(T_k, P_i) \neq T(T_k, P'_i)$.

The targeted applications on the grid are very time consuming. For this reason, we make the assumption that the bandwidth of PF allows us to neglect the communication time for all the computation steps of each instance J_i of the job J.

According to the $\alpha|\beta|\gamma$ [5] classification of scheduling problems, this problem is defined by $U_r|$batch of intrees$|C_{max}$: the platform is heterogeneous thus the execution times are unrelated and we optimize the makespan C_{max} of a batch of intree. Finding an optimal schedule for a batch of jobs on a heterogeneous platform with limited resources is known to be an NP-complete problem, so there is no direct method. Two solutions may be used: either use a heuristic to compute a suboptimal schedule or use the results of an optimal solution to a problem close to ours.

III. Related works

Three approaches match this context.

Steady-state techniques achieve an optimal use of the resources for an infinite number of identical jobs [1]. The resulting schedule is composed of an initialization stage before entering in steady-state where it becomes optimal. This result will tend towards optimality when the size of the batch increases, as the weight of the initialization stage decreases in the global schedule. When the size of the batch is too small, the initialization stage overhead leads however to an

inefficient schedule. So, the question is "from what batch size steady-state scheduling becomes interesting?".

Classical solutions to optimize the makespan of a set of tasks relay on heuristics such as Earliest Finish Time [10] or Critical Path [7]. These heuristics approaches compute a schedule off-line with the assumption that no other jobs will be run on the platform. These techniques schedule the whole set of tasks, so they do not suffer from the initialization problem. However, if the number of tasks scales up, the computation time becomes too long due to the complexity of the algorithm.

On-line oriented techniques generate the schedule during the jobs execution. They take the state of the system into account to assign new tasks to processors. These techniques are very easy to implement and give rather good results. They give however no guarantee on the optimality of the schedule.

Other works exist on scheduling multiple DAGs. Most of them describe real-time schedulers for periodic tasks [8], [9]. Real-time schedulers are not adapted to our context because their objectives is to ensure that the tasks deadlines are met. Iverson and Özgüner [6] use a different technique where multiple DAGs compete for the available computational resources. In the paper, schedulers have limited informations on processors and no informations on future jobs to schedule. This is not the case in our context where a centralized entity schedules all the jobs.

IV. Selected solutions

In this section we present the selected algorithms and their adaptation to the context characteristics. represents one of the three possible approaches and provide good results in our context. The first algorithm is a static scheduling algorithms for graphs of aperiodic tasks on a heterogeneous platform. It allows us to evaluate the performances of an algorithm designed for a set of tasks to schedule middle size batches of jobs. The second algorithm is a simple list-based on-line scheduling algorithm that we use to obtain a reference makespan. The third algorithm is a steady-state oriented scheduling algorithm. This algorithm uses a linear program to optimize the number of jobs executed per time unit in steady-state, and produces an optimal schedule[1]. It allows us to evaluate the performance of an optimal steady-state algorithm, designed for an unlimited number of jobs, in the context of middle sized batch.

A. GATS

Genetic Algorithm for Task Scheduling (GATS) is a genetic algorithm that produces 7% to 10% shorter schedule than classical scheduling techniques for aperiodic tasks on a heterogeneous platform [4]. It uses the genetic metaheuristic to enhance the schedule obtained by a list-based scheduling heuristic. The first step is to create an initial population: the first individual represents the result of a list-based schedule which favors the tasks on the critical path of J. Then one individual per processor represents a random schedule where all the tasks are affected to the given processor and the remain of the population represents random schedules.

GATS individuals do not represent total schedules but only processors-tasks associations. A decoding step translates each individual to a schedule that respects all the dependencies. The decoded schedule is used to compute the fitness f of an individual as: $f = 1/Makespan(schedule)$. The representation without time informations allows fast computations of mutations and cross-overs since tasks are directly movable from one slot to another. Once the initial population is created, GATS performs a loop that computes the fitnesses of the individuals, a rank-based selection, mutations and cross-overs until the termination criteria are met. The termination criterion used is a fixed number of loops. The computed schedule is at least equal to the initial list-based solution.

The original GATS algorithm is not designed to schedule multiple instances of the same job. One simple method to make GATS able to solve this problem is to schedule all the job instances as a single job. This method increases however the time and the physical memory necessary for GATS to compute the schedule. So we use a hybrid method that schedules the tasks in successive intervals of x jobs and appends the set schedules.

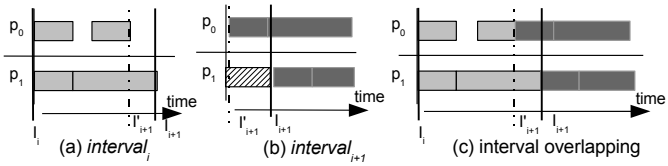

Fig. 1: Overlapping of $Interval_i$ and $Interval_{i+1}$.

This method introduces a new problem: the concatenation of two successive intervals leaves some processors idle so, we overlap intervals as shown on figure 1. The interval $Interval_i$ ends when the processor P_1 finishes its task (fig.(a) I_{i+1}). The processor P_0 is idle between $I^{'}_{i+1}$ and I_{i+1} (fig. (b)). The interval overlapping re-uses this idle time in the next interval. The interval $Interval_{i+1}$ starts at $I^{'}_{i+1}$ instead of I_{i+1} with the information that P_1 is not available until I_{i+1}. Figure (c) shows the overlapping of $Interval_i$ and $Interval_{i+1}$.

B. On-line Scheduling with knowledge

Algorithm 1 onlineScheduler(processors, job, count)

```
job ← job_multiply(job, count)
while not isEmpty(job) do
  free ← update_free_list(job, free)
  T_best ← remove_first(free)
  P_best ← smallest_EFT(processors, T_best)
  send(T_best, P_best)
  if isEmpty(free) then
    Wait for a task to be free
  end if
  Wait for a processor to be idle
end while
```

The algorithm 1 is a simple implementation of an on-line scheduler. When the on-line scheduler is asked to execute $count$ jobs, it creates a single job (job) that groups the $count$ jobs to be scheduled. The algorithm manages a $free$ tasks list that contains tasks with no dependency. At each loop

of the algorithm, the first task of the list is selected to be scheduled. This algorithm does not use a heuristic to select the task to schedule contrary to other list based scheduling algorithms [10], [11]. This mechanism shortens the traversal time of a job. T_{best} is used to select the best processor (P_{best}) using the Earliest Finish Time (EFT) heuristic [10]. The selected task is sent to the selected processor then the algorithm waits for, at least, one free task and one idle processor, until all the jobs are carried out.

C. Steady-state scheduling

The steady-state scheduling technique uses a linear program to compute an optimal schedule when the system enters a steady-state. The objective function of the linear program is to maximize the number of jobs computed per time unit. The constraints of the linear program force the solution to respect the computation abilities of each processor. These constraints also ensure that jobs are fully computed.

The solution of this linear program gives the ratio of time spent by each processor for each task of the jobs and the proportion of time spent by each network link to send task results for each inter-tasks dependencies. As the execution context is a steady-state, the proportion can be computed as a rational number. This solution is translated into a weighted sum of allocation, where an allocation represents the traversal of a job in the platform (task/processor associations). The steady-state scheduling computes a period in which the allocations are interleaved according to their weight. The steady-state stage is the concatenation of the adequate number of these periods to compute the number of desired jobs.

The scheduling needs an initialization stage before entering in steady-state and a termination stage to finish after the last period. The initialization stage computes every task needed to enter into the steady-state stage. In the original algorithm, the initialization is not optimized. The master computes itself every tasks without parallelism facility. As the initialization, the termination stage is performed by the master in a sequential way. Its role is to terminate every task staying in the platform after the last steady-state period.

V. Simulation

Building a mathematical model of the algorithms is not realistic due to the complexity of the problem. Their implementation on a grid cannot give reproducible results. So we use a grid simulator to evaluate the three algorithms.

The simulator was implemented above SimGrid and its MSG API [2], [3]. it is based on the master/slave paradigm. Algorithms are implemented in the master node and dispatch tasks to slave nodes according to the scheduling decisions.

Figure I shows the platforms and figure 2 shows the jobs used in the simulation discussed in the next section. For instance, in the platform PF_0, the processor p_1 needs 10 time units to perform a task A ($p_1(A) = 10$).

The processor and algorithms performances are expressed in time units as there are obtained by simulation and not by experiments. These results may be applied to different time units: seconds, minutes, etc.

Task	PF_0				PF_1					
	p_1	p_2	p_3	p_4	p_1	p_2	p_3	p_4	p_5	p_6
A	10	-	-	-	10	-	-	-	100	1000
B	-	10	-	-	-	10	-	-	10	-
C	-	-	10	-	-	-	10	-	10	10
D	-	-	-	10	-	-	-	10	-	-

Task	PF_2			PF_3			PF_4			
	p_1	p_2	p_3	p_1	p_2	p_3	p_1	p_2	p_3	p_4
A	10	50	40	100	100	100	20	-	-	20
B	100	-	-	20	-	-	10	10	-	-
C	-	-	-	-	-	-	-	10	10	-
D	-	-	-	-	-	-	-	-	10	10

TABLE I: Simulation platforms

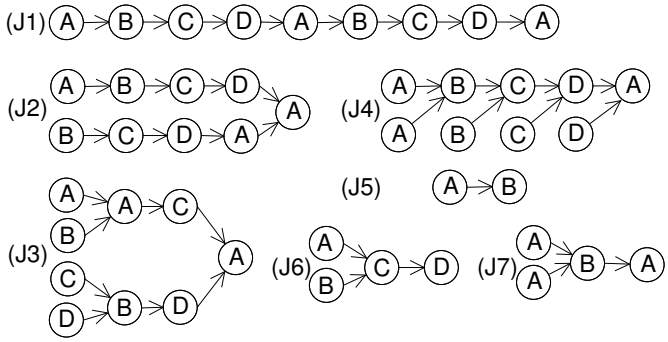

Fig. 2: Simulation jobs

VI. Results

The results presented in this section are obtained by scheduling the different batches of jobs (fig. 2 (J_1, ..., J_7)) on the different platforms (fig. I (PF_0, ..., PF_4)).

Table II gives the simulations results for small (50), medium (100) and big (500) batches. The capacity (Cap.) is a lower bound of the time needed to execute one job: the size of the period the steady-state algorithm divided by the number of jobs produced during the period. Since the schedule computed by the steady-state algorithm is optimal, this value is a lower bound for the time needed to execute one job. The results given in table II are obtained using the following expression: $performance = capacity \times batch\ size/makespan$. So, this value represents the ratio of their makespan and the lower bound.

A. Performances

GATS is the best algorithm for small and medium batches with an average performance of 0.99. On-line scheduling and the steady-state scheduling are the seconds with an average performance of 0.93. For big batches (500 jobs), the steady-state algorithm gives the best results, on-line scheduler is the second and GATS has very poor performances.

The on-line algorithm is always good and stable around 0.94 with a minimum value of 0.86 and the two others are less reliable as, in some cases, they have very bad performances with 0.46 for steady-state and 0.22 for GATS. Each of these two algorithms has however its own domain of stability: GATS in small and medium sizes and steady-state in larger sizes.

Another result not shown by this table, is that our adaptations on the algorithms – parallelizing the initialization and

			Small batch : 50 jobs			Medium batch : 100 jobs			Big batch : 500 jobs		
Plat.	Job	Cap.	On-line	Stead.	GATS	On-line	Stead.	GATS	On-line	Stead.	GATS
PF_0	J_1	30	1	0.99	1	1	1	1	1	1	1
PF_0	J_2	30	1	0.99	1	1	1	1	1	1	1
PF_0	J_3	30	1	1	1	1	1	1	1	1	1
PF_0	J_4	30	1	1	1	1	1	1	1	1	1
PF_0	J_5	10	0.98	1	0.98	0.99	1	0.99	1	1	1
PF_1	J_1	27.03	0.9	0.89	0.94	0.9	0.9	0.94	0.9	0.94	0.22
PF_1	J_2	27.03	0.9	0.89	0.97	0.9	0.9	0.97	0.9	0.89	0.45
PF_1	J_3	27.03	0.9	0.9	0.99	0.9	0.9	0.97	0.9	0.94	0.64
PF_1	J_4	27.03	0.9	0.82	0.99	0.9	0.83	0.96	0.9	0.94	0.3
PF_1	J_5	9.09	0.89	0.46	0.97	0.9	0.46	0.98	0.91	0.8	0.99
PF_2	J_5	100	0.91	1	1	0.91	1	1	0.93	1	0.93
PF_3	J_5	40	0.86	0.91	0.98	0.89	0.95	0.99	0.89	0.99	0.54
PF_4	J_6	12.5	0.89	0.8	0.95	0.91	0.89	0.96	0.91	0.98	1
PF_4	J_7	30	1	0.99	1	1	0.99	1	1	1	0.54
Average			0.93	0.93	0.99	0.94	0.94	0.99	0.94	0.97	0.61

TABLE II: Performance of On-line, Steady-state and GATS against platform capacity.

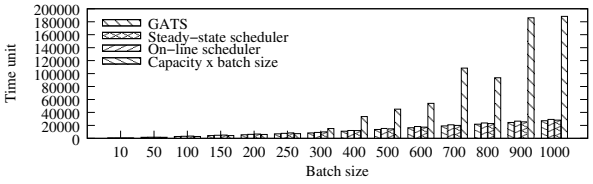

Fig. 3a: Simulation with platform PF_1 and job J_4

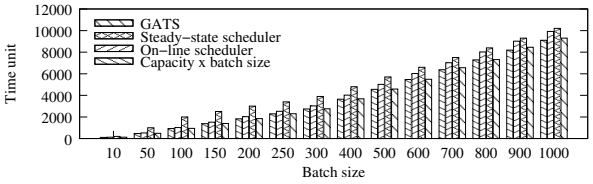

Fig. 3b: Simulation with platform PF_1 and job J_5

termination stages of the steady-state algorithm and using limited buffer size in the on-line algorithm – generate significant performances improvements. For instance, for on-line algorithm, the mean result for small batches is improved by 16% and, for steady-state algorithm, it has been improved by 52%.

B. Batch size

The main drawbacks of the steady-state scheduler are its initialization and termination stages. Figure 3a shows an experiment where the steady-state period takes 1000 time units (one period) to execute 37 jobs. The initialization executes 148 partial jobs (740 tasks) and the termination cleans up 148 partial jobs (592 tasks). In this experiment, the steady-state algorithm makes up its handicap for a batch of 500 jobs where it becomes better than the two other algorithms. However, figure 3b shows that this is not always true. In this experiment the steady-state period takes 1000 time units to execute 110 jobs, the initialization and the termination executes 110 partial jobs and the scheduling is not able to make up its handicap even for 1000 jobs. The difference between these two experiments is in the parallelism of the initialization and the termination stage. In (a) a high level of parallelism is possible as tasks of types A, B, C and D are scheduled in initialization and termination while, in (b), the initialization stage manages only tasks of type A and termination tasks of type B.

GATS is very sensitive to the batch size, for small and medium batches (50 and 100) GATS has an average performance of 0.99 but it falls to 0.61 for big batches (500). One of the reasons of this performance loss may be the scheduling intervals presented in IV-A: GATS does not enhance the whole schedule but just partial schedule intervals of 200 tasks. Note that some of the results, on PF_0 for example, are not affected by this, so intervals are note the only reason. We propose another explanation in the next section.

The on-line scheduler performances does not change with the batch size. This is not surprising as the algorithm takes jobs one after another and thus does not depend on the batch size.

C. Platform / Job

We can observe that the performances depend on the platform characteristics. On platform PF_0, where no parallelism is possible for a task type, the three algorithms perform well. On platform PF_1, the three algorithms choose where to schedule A, B, C tasks as several processors can execute them. This leads to poor performances for GATS on batches bigger than 500 because the search space is too large. For the steady-state scheduler, the number of jobs per steady-state period becomes larger leading to longer initialization and termination stages and so to bad performances.

The on-line scheduler performs well in general as shown on figure 4a. The only identified drawback is that the algorithm does not take the graph dependencies into account. For instance, in the case of the platform PF_2 (fig. 4b), the first processor is the fastest at producing A tasks and the only one able to carry out B tasks. For a set of jobs composed of two tasks of type A and B with B depending on A, the on-line scheduler first balances the load of A tasks between all

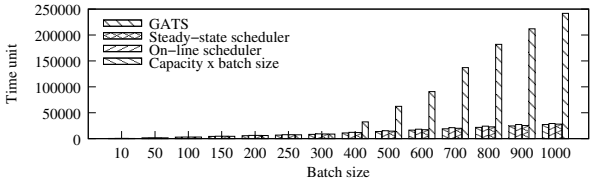

Fig. 4a: Simulation with platform PF_1 and job J_1

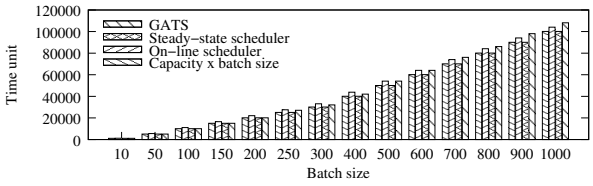

Fig. 4b: Simulation with platform PF_2 and job J_5

processors, as they are the only free tasks. Then, when tasks B are freed, the first processor has to finish the execution of all its A tasks before executing B tasks. This leads to a performance loss as shown on figure 4b or on table II when scheduling 50 jobs J_5 on PF_1.

As explained in the previous subsection the steady-state algorithm may suffer from its initialization and termination stages. These two stages closely depend on the period size and on the number of jobs in the period. The period depends on the platform and the job. For example, on table II, scheduling job J_5 on platform PF_1 leads to a period of 1000 time units for 110 jobs. On the opposite scheduling job J_5 on platform PF_0 leads to a period of 10 time units for 1 job. The steady-state algorithm performs poorly when the platform/job association leads to large number of jobs in a period, in particular for small to medium batches. This is shown by the results of job J_5: on platform PF_0 the schedule is always optimal, whereas, on PF_1, the performances increase with the batch size to make up the delay generated by the initialization and termination stages.

As a genetic algorithm, GATS explores the possible schedules space to find a good schedule. This space grows when processors are able to execute different types of tasks and when the jobs contain tasks of the same type like in the experiments involving platform PF_1 or PF_5 (Table II). In such big spaces, GATS is not able to converge to a good schedule and its performances fall when the size of the batch increases, as shown in figure 4a. This is also significant on table II when scheduling jobs on platform PF_1 compared to platform PF_0.

D. Computation time of the scheduling stage

The table III shows the average time needed on an Intel Core 2 Duo running at 1.6 GHz to compute the schedule of 1000 jobs using the three different algorithms. We can note that the steady-state scheduler is the fastest. The on-line scheduler is also very fast, but, on real experiments we must take care as it does not provide any guaranties on real time use. GATS is very slow, this is acceptable to schedule a batch, but the computation overhead is too important to launch an execution on the fly.

On-line scheduler	Steady-state scheduler	GATS
35.04s	0.08s	1799.68s

TABLE III: Average time spent by the micro-processor to compute the schedule of 1000 jobs.

VII. Conclusion and future works

In this paper, we have presented a performance comparison with three scheduling algorithms for bounded batches of jobs on heterogeneous platforms. On-line scheduling is a good algorithm, but it does not reach optimal performances when the size of the batches increases. Steady-state suffers from its initialization and termination stages. it could be improved by parallelizing these two stages. Another point to look at is the steady-state period, the number of jobs to perform in the initialization and the termination is directly linked to the number of jobs per steady-state period. Futures works should try to find a balance between the period jobs count and the number of jobs to schedule. GATS obtains excellent performances for small batches but degrades with the batch size. Future work should investigate the performances degradation and find solutions to avoid them. The computation time needed to obtain the schedule from GATS cannot be improved by a great magnitude, code optimization will reduce it, but the inner time complexity of GATS forces it to be computation intensive.

Future studies will generalize the problem by introducing communication costs and DAG with forks. The forks on DAG will increase the complexity of the steady-state scheduling, future experiments should tell us if it is significant. Fault tolerance is also an issue of interest : off-line algorithms (steady-state and GATS) will probably suffer from the unpredictable platform behavior while the on-line scheduler should not need any adaptations as it is based on dynamic informations.

References

[1] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. *Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms*. IEEE Computer Society Press, 2004.

[2] H. Casanova. Simgrid: A toolkit for the simulation of application scheduling. *ccgrid*, 00:430, 2001.

[3] H. Casanova, A. Legrand, and L. Marchal. Scheduling distributed applications: the simgrid simulation framework. In *3rd IEEE Intl Symposium on Cluster Computing and the Grid*, 2003.

[4] M. Daoud and N. Kharma. Gats 1.0: A novel ga-based scheduling algorithm for task scheduling on heterogeneous processor nets. In *Genetic And Evolutionary Computation Conference*, 2005.

[5] R.L. Graham and al. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Ann. Discrete Math.*, 4:287–326, 1979.

[6] M. Iverson and F. Özgüner. Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment. In *7th Heterogeneous Computing Workshop*, pages 70 – 78, 1998.

[7] Y. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multi-processors. In *IEEE Trans. on Parallel and Distributed Systems*, pages 506 – 521, 1996.

[8] Y. Li and W. Wolf. Hierarchical scheduling and allocation of multirate systems on heterogeneous multiprocessors. In *European conference on Design and Test*, pages 134 – 139, 1997.

[9] X. Qin and H. Jiang. A dynamic and reliability-driven scheduling algorithm for parallel real-time jobs executing on heterogeneous clusters. *Journal of Parallel and Distributed Computing*, 65(8):885–900, 2005.

[10] H. Topcuouglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. In *IEEE Trans. on Parallel and Distributed Systems*, pages 260 – 274, 2002.

[11] M. Wu, W. Shu, and H. Zhang. Segmented min-min: A static mapping algorithm for meta-tasks on heterogeneous computing systems. In *9th Heterogeneous Computing Workshop*, pages 375 – 385, 2000.