

Scheduling Malleable Applications in Multicenter Systems

Jérémy Buisson ^{#1}, Ozan Sonmez ^{*2}, Hashim Mohamed ^{*3}, Wouter Lammers ^{*4}, Dick Epema ^{*5}

#IRISA/INSA de Rennes

Campus de Beaulieu, 35042 Rennes CEDEX, France

¹jbuisson@irisa.fr

**Delft University of Technology*

P.O Box 5031, 2600 GA, Delft, The Netherlands

²O.O.Sonmez@tudelft.nl

³H.H.Mohamed@tudelft.nl

⁴wouter@w78.nl

⁵D.H.Epema@tudelft.nl

Abstract—In large-scale distributed execution environments such as multicenter systems and grids, resource availability may vary due to resource failures and because resources may be added to or withdrawn from such environments at any time. In addition, single sites in such systems may have to deal with workloads originating from both local users and from many other sources. As a result, application malleability, that is, the property of applications to deal with a varying amount of resources during their execution, may be very beneficial for performance. In this paper we present the design of the support of and scheduling policies for malleability in our KOALA multicenter scheduler with the help of our DYNACO framework for application malleability. In addition, we show the results of experiments with scheduling malleable workloads with KOALA in our DAS multicenter testbed.

I. INTRODUCTION

Application malleability, that is, the property of applications to use varying amounts of resources such as processors during their execution, is potentially a very versatile and beneficial feature. Allowing resource allocation to vary during execution, malleability gives a scheduler the opportunity to revise its decisions even after applications have started executing. Increasing the flexibility of applications by shrinking their resource allocations, malleability allows new jobs to start sooner, possibly with resources that are not going to be usable during their whole execution. Making applications able to benefit from the resources that appear during their execution by growing their allocations, malleability also helps applications terminate sooner. In addition to these very general advantages, malleability makes it easier to deal with the dynamic nature of large-scale distributed execution environments such as multicenter systems, and more generally grids. In this paper, we present the design of the support for malleability in our KOALA [1] multicenter scheduler by means of the inclusion of our DYNACO framework [2] for application malleability, and the design and analysis of two scheduling policies for malleable applications.

In execution environments such as multicenters and grids, the availability of resources varies frequently [3]. In addition to

failures, resources may be allocated (or released) by concurrent users, and organizations may add or withdraw (parts of) their resources to/from the resource pool at any time. In any of these cases, malleability allows applications to benefit from appearing available resources, while gracefully releasing resources that are reclaimed by the environment. Malleability thus holds a great promise in strategies to more performant execution in multicenters. Besides, malleable applications give schedulers the opportunity to increase system utilization.

On the one hand, despite that several approaches have been proposed to build malleable applications [2], [4], [5], [6], [7], [8], virtually no existing multicenter and grid infrastructures are able to benefit from this property. Consequently, many applications embed their own specific scheduler and submit bulks of jobs in order to build dynamic resource management on top of existing infrastructures. On the other hand, most of the previous work on scheduling malleable applications does not handle the challenges that appear in the context of multicenter systems. Specifically, issues such as the selection of a suitable cluster for each job and resilience to background load due to local users are often not taken into account. Furthermore, many proposed approaches have only been tested with simulations, and an assessment of the overhead due to the implementation of grow and shrink operations are commonly omitted.

Our contributions in this paper are the following. First, we present an architecture and an actual implementation of the support for malleability in grid schedulers, showing the benefits of the modular structure of KOALA as a by-product. Second, we present two policies for managing malleability in the scheduler, one which hands out any additional processor to the malleable jobs that have been running the longest, and one that spreads them equally over all malleable jobs; each of these policies can be combined with one of two approaches which either favour running or waiting jobs. Third, we evaluate these policies and approaches in combination with KOALA's worst-fit load-sharing scheduling policy with experiments in the DAS3 [9] testbed. These experiments show that a higher

utilization and shorter execution times can be achieved when malleability is used.

The rest of this paper is structured as follows. Section II states more precisely the problem addressed in this paper, and Section III reviews the state of the art of malleability in resource management in multicluster and grid systems. Section IV describes the KOALA grid scheduler and the DYNACO framework, which are the starting points of our work. Section V describes how we support malleability in KOALA, and details the malleability management approaches and policies that we propose. Section VI presents the experimental setup, and Section VII discusses our experimental results. Finally, Section VIII makes some concluding remarks and points to future work.

II. PROBLEM STATEMENT

In multicluster systems and more generally in grids, there may be various types of parallel applications that can benefit from being able to change their processor configuration after they have started execution. Malleability of parallel applications may yield improved application performance and better system utilization since it allows more flexible scheduling policies. In this paper, we propose and compare scheduling approaches that take into account malleable applications in order to assess such benefits. In this section, we first classify parallel applications in order to distinguish malleable applications, and then we address several aspects of malleable applications that should be taken into account by a resource management or a scheduling system.

A. Classification of Parallel Jobs

Following the well-known parallel job classification scheme presented in [10], we consider three types of jobs, namely, rigid, moldable, and malleable. A *rigid* job requires a fixed number of processors. When the number of processors can be adapted only at the start of the execution, the job is called *moldable*. Similar to rigid jobs, the number of processors for moldable jobs cannot be changed during runtime. Jobs that have the flexibility to change the number of assigned processors during their runtime (i.e., they can grow or shrink) are called *malleable*.

B. Specification of Malleable Jobs

A malleable job may specify the *minimum* and *maximum* number of processors it requires. The minimum value is the minimum number of processors a malleable job needs to be able to run; the job cannot shrink below this value. The maximum value is the maximum number of processors a malleable job can handle; allocating more than the maximum value would just waste processors. We do not assume that a *stepsize* indicating the number of processors by which a malleable application can grow or shrink is defined. We leave the determination of the amount of growing and shrinking to the protocol between the scheduler and the application (see Section V).

C. Initiative of Change

Another aspect that we consider is party that takes the initiative of changing the size of a malleable job (shrinking or growing). Either the application or the scheduler may initiate grow or shrink requests. An application may do so when the computation it is performing calls for it. For example, a computation can be in need of more processors before it can continue. On the other hand, the scheduler may decide that a malleable job has to shrink or grow based on the availability of free processors in the system. For example, the arrival of new jobs to a system that is heavily loaded may trigger a scheduler to requests currently running malleable jobs to shrink.

D. The Obligation to Change

Requests for changing the size of a malleable job may or may not have to be satisfied. A *voluntary* change means that the change does not have to succeed or does not necessarily have to be executed; it is merely a guideline. A *mandatory* change, however, has to be accommodated, because either the application cannot proceed without the change, or because the system is in direct need of the reclaimed processors.

III. RELATED WORK

Much of the previous research on the problem of scheduling and allocating resources to malleable jobs has focused on theoretical aspects [11], [12], [13]. Thus, the results have been obtained in simulated environments, often neglecting the issues that arise in real environments such as the effective scalability of applications and the cost of growing or shrinking. In this section, we discuss several implementations of malleable applications and their scheduling in real multicluster systems.

As noted in [14] and in our previous work [2], malleability helps applications perform better when resource availability varies. Several approaches have been used to make parallel and/or multicluster applications malleable. While GRADS [5] relies on the SRS [15] checkpoint library, APPLES [4] and ASSIST [6] propose to build applications upon intrinsically malleable skeletons. With AMPI [7], malleability is obtained by translating MPI applications to a large number of CHARM++ objects, which can be migrated at runtime. Utrera *et al.* [8] propose to make MPI applications malleable by folding several processes onto each processor.

A couple of works [4], [5] have studied how to schedule such applications in conjunction with building malleable applications. Among them, APPLES [4] and GRADS [5] are somewhat specific as they propose that applications are responsible to schedule themselves on their own. However, this approach raises the question of how the system behaviour and performance would be in case several concurrent malleable applications compete for resources. Furthermore, as those approaches rely on checkpointing, it is unclear how an application gets its resources back when it accepts to try a new allocation. System-wide schedulers do not suffer from these drawbacks.

Other approaches rely on a system-wide scheduler. Corresponding to the underlying execution model, AMPI uses an

equipartition policy, which ensures that all jobs get almost the same number of processors; while the policy in [8] is based on folding and unfolding the jobs (i.e., doubling or halving the number of allocated processors). However, those two approaches rely on the properties of their underlying execution model. For instance, equipartition assumes that any application can be executed efficiently with any number of processors, as it is the case with AMPI; while folding restricts the number of processes to be divisible by the number of processors (often a power of 2 for practical reasons), which is the only way to fold efficiently non-malleable applications. A more general approach such as the one we propose is more appropriate in the context of multiclusters.

McCann and Zahorjan [16] further discuss the folding and equipartition policies. According to their experiments, folding preserves well efficiency; while equipartition provides higher fairness. They have proposed in [16] a rotation policy in order to increase the fairness of the folding policy. However, rotation is almost impracticable in the context of multiclusters.

As fairness in terms of allocated processors does not imply efficiency, a biased equipartition policy is proposed in Hungershöfer *et al.* [17] such that the cumulative speedup of the system is maximized. It also considers both malleable and rigid jobs in a single system in [18], and it guarantees to allocate a minimum number of processors to each malleable job, such that they are not ruled out by rigid jobs. However, in multiclusters, it is common that some of the users bypass the multicluster-level scheduler. The problem of making the scheduler take into account that incurred background load is not addressed in the works of Hungershöfer *et al.*

In addition, most of those previous research works [8], [16], [17], [18] have not considered the combination of malleability management and load sharing policies across clusters, which is an issue specific to multiclusters.

IV. BACKGROUND

In this section, we summarize our previous work on a co-allocating grid scheduler called KOALA and on the implementation of malleable applications, which is the starting point of the work reported in this paper. Section IV-A presents the KOALA grid scheduler, followed by section IV-B that describes our DYNACO framework that we use to implement malleable applications.

A. The KOALA multicluster scheduler

The KOALA [1] grid scheduler has been designed for multicluster systems such as the DAS [9]. KOALA job submission tools employ some of the GLOBUS toolkit [19] services for job submission, file transfer, and security and authentication. On the other hand, KOALA scheduler has its own mechanisms for data and processor co-allocation, resource monitoring, and fault tolerance.

Figure 1 shows the architecture of KOALA. This architecture shows auxiliary tools called runners, which provide users with an interface for submitting jobs and monitoring their progress for different application types. Runners are built out of a

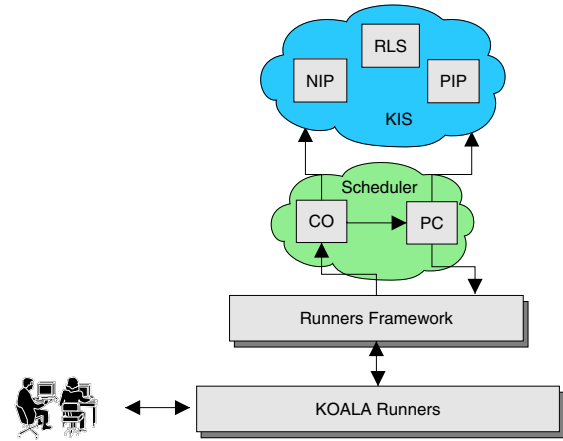


Fig. 1. Overview of the architecture of the KOALA multicluster scheduler.

framework, which serves as a frontend between each runner and the centralized scheduler. The latter is made of a co-allocator (CO), which decides of resource allocations, and of a processor claimer (PC), which ensures that processors will still be available when the job starts to run. If processor reservation is supported by local resource managers, the PC can reserve processors immediately after the placement of the components. Otherwise, the PC uses KOALA claiming policy [20], [21] to postpone claiming of processors to a time close to the estimated job start time. In its tasks, the scheduler is supported by the KOALA information service (KIS), which monitors the status of resources thanks to a processor information provider (PIP), a network information provider (NIP) and a replica location service (RLS). Providers connect KOALA with the multicluster monitoring infrastructure, which can be GLOBUS MDS or whatever else depending on the used resource managers.

Within the context of KOALA job model, a job comprises either one or multiple components that each can run on a separate cluster. Each job component specifies its requirements and preferences such as the program it wants to run, the number of processors it needs, and the names of its input files.

Upon receiving a job request from a runner, the KOALA scheduler uses one of the placement policies described below, to try to place job components on clusters. With KOALA, users are given the option of selecting one of these placement policies.

- The Worst-Fit (WF) [22] places each component of a job in the cluster with the largest number of idle processors. The advantage of WF is its automatic load-balancing behaviour, the disadvantage is that large (components of) jobs have less chance of successful placement because WF tends to reduce the number of idle processors per cluster.
- The Close-to-Files (CF) policy [20] uses information about the presence of input files to decide where to place (components of) jobs. Clusters with the necessary

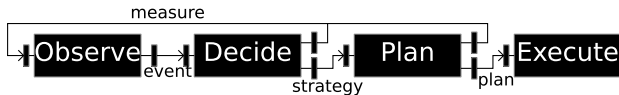


Fig. 2. Overview of the architecture of the DYNACO framework for adaptability.

input files already present are favoured as placement candidates, followed by clusters for which transfer of those files take the least amount of time.

- The Cluster Minimization (CM) and the Flexible Cluster Minimization (FCM) policies [23] have been designed especially for jobs that may use co-allocation in order to minimize the number of clusters to be combined for a given parallel job, such that the number of inter-cluster messages is reduced. The flexible version decreases in addition the queue time by splitting jobs into components according to the number of idle processors.

If the placement of the job succeeds and input files are required, the scheduler informs a runner to initiate the third-party file transfers from the selected file sites to the execution sites of the job components. If a placement try fails, KOALA places the job at the tail of a placement queue. This queue holds all the jobs that have not yet been successfully placed. The scheduler regularly scans this queue from head to tail to see whether any job is able to be placed. For each job in the queue we record its number of placement tries, and when this number exceeds a certain threshold value, the submission of that job fails.

B. The DYNACO framework and its use for malleability

In our previous work [2], we have proposed techniques to implement malleability as a special case of adaptability. Basically, adaptability is an approach for addressing the problem of the dynamicity of large-scale execution environments. It consists in the ability of applications to modify themselves during their execution according to constraints imposed by the execution environment. Our previous work on abstracting adaptability [24] has resulted in DYNACO¹, a generic framework for building dynamically adaptable applications.

As its architecture shows in Figure 2, DYNACO decomposes adaptability into four components, similarly to the control loop suggested in [25]: the *observe* component monitors the execution environment in order to detect any relevant change; relying on this information, the *decide* component makes the decision about adaptability. It decides when the application should adapt itself and which strategy should be adopted. When the strategy in use has to be changed, the *plan* component plans how to make the application adopt the new strategy; finally, the *execute* component schedules actions listed in the plan, taking into account the synchronization with the application code. Being a framework, DYNACO is expected

to be specialized for each application. In particular, developers must provide the decision procedure, the description of planning problems, and the implementation of adaptation actions. In addition, we have proposed AFPAC [26] as an implementation of the *execute* component that is specific to SPMD applications. Tools provided in IBIS [27], ASSIST [6], skeleton-based paradigms and similars can be used as well.

As reported in [2], DYNACO and AFPAC have been successfully used to make several existing MPI-based applications malleable. While not being restricted to this class of applications, DYNACO contributes to reduce the cost of transforming existing parallel applications into malleable ones when it is combined with tools such as AFPAC.

V. DESIGNING SUPPORT FOR MALLEABILITY IN KOALA

In this section, we present our design for supporting malleable applications in KOALA. First, we explain how we include the DYNACO framework into the KOALA multicluster scheduler, and then we present our approaches and policies for managing the execution of malleable applications, respectively.

A. Supporting DYNACO applications in KOALA

In order to support DYNACO-based applications in KOALA, we have designed a specific runner called the Malleable Runner (MRunner); its architecture is shown in Figure 3. In the MRunner, the usual control role of the runner over the application is extended in order to handle malleability operations. For that purpose a complete instance of DYNACO is included in the MRunner on a per-application basis. A frontend, which is common to all of the runners, interfaces the MRunner to the scheduler. We add a malleability manager in the scheduler, which is responsible for triggering changes of resource allocations.

In the DYNACO framework, the frontend is reflected as a monitor, which generates events when it receives *grow* and *shrink* messages from the scheduler. Resulting events are propagated throughout the DYNACO framework and translated into the appropriate messages to GRAM and to the application. The frontend catches the results of adaptations in order to generate acknowledgments back to the scheduler. It also notifies the scheduler when the application voluntarily shrinks below the amount of allocated processors.

GRAM is currently not able to manage malleable jobs. We further discuss this issue in [28]. In this paper, for the sake of simplicity and despite the poor reactivity of that solution, the MRunner manages the malleable job as a collection of GRAM jobs of size 1. Upon growth, the MRunner submits new jobs to GRAM. When it receives *active* messages from GRAM, it transmits the new collection of active GRAM jobs (i.e. the collection of held resources) to the application. In order to reduce the impact on the execution time, interactions with GRAM overlap with the execution of the application and suspension of the application does not occur before all the resources are held. To do so, GRAM submissions launch an empty stub rather than the application's program. The stub is turned into an application process during the

¹DYNACO is available at the following website: <http://dynaco.gforge.inria.fr>

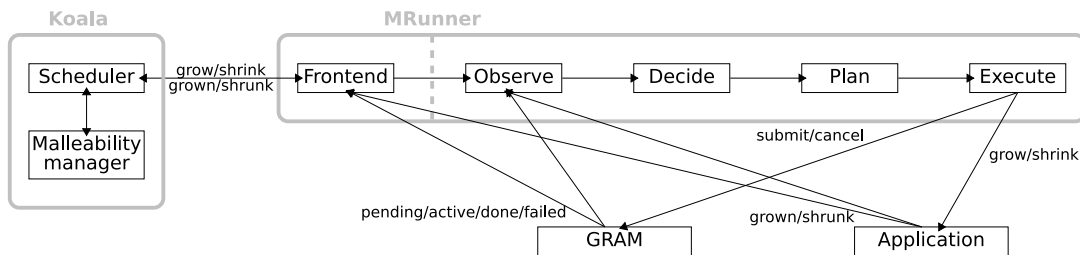


Fig. 3. The architecture of the Malleable Runner with DYNACO in the KOALA multicluster scheduler.

process management phase, when resources are recruited by the application. That latter operation is faster than submitting a job to GRAM as it is relieved from tasks such as security enforcement and queue management. Conversely, upon shrink, the MRunner first reclaims processors from the application; then when it receives *shrunk* feedback messages, it releases the corresponding GRAM jobs. Again, interactions with GRAM overlap the execution, which resumes immediately.

B. Job Management

Upon submission of an application to KOALA, whether it is rigid, moldable or malleable, the initial placement is performed by one of the existing placement policies as described in Section IV-A. In the placement phase of malleable applications, the initial number of processors required is determined considering the number of available processors in the system. Specifically, given a malleable application, the placement policies place it if the number of available processors is at least equal to the minimum processor requirement of the application.

In the job management context, the malleability manager is responsible for initiating malleability management policies that decide on how to grow or shrink malleable applications. Below, we propose two design choices as to when to initiate malleable management policies, which give Precedence to Running Applications over waiting ones (PRA) or vice versa (PWA), respectively.

In the PRA approach, whenever processors become available, for instance, when a job finishes execution, first the running applications are considered. If there are malleable jobs running, one of the malleability management policies is initiated in order to grow them; any waiting malleable jobs are not considered as long as at least one running malleable job can still be grown.

In PWA approach, when the next job j in the queue cannot be placed, the scheduler applies one of the malleability management policies for shrinking running malleable jobs in order to obtain additional processors. Those shrink operations are mandatory. If it is however impossible to get enough available processors in order to place job j taking into account the minimum sizes of the running jobs, then the running malleable jobs are considered for growing by one of the malleable management policies. Whenever processors become available, the placement queue is scanned in order to find a job to be placed.

In both approaches, in order to trigger job management, the scheduler periodically polls the KOALA information service. In doing so, the scheduler is able to take into account dynamically the background load due to other users even if they bypass KOALA. In addition, in order not to stress execution sites when growing malleable jobs, and therefore, in order to leave always a minimal number of available processors to local users, a threshold is set over which KOALA never expands the total set of the jobs it manages.

C. Malleability Management Policies

The malleability management policies we describe below determine the means of shrinking and growing of malleable jobs during their execution. In this paper, we assume that every application is executed in a single cluster, and so, no co-allocation takes place. Consequently, the policies are applied for each cluster separately.

1) *Favour Previously Started Malleable Applications (FPSMA)*: The FPSMA policy favours previously started malleable jobs in that whenever the policy is initiated by the malleability manager, it starts growing from the earliest started malleable job and starts shrinking from the latest started malleable job. Figure 4 presents the pseudo-code of the grow and shrink procedures of the FPSMA policy.

In the *grow* procedure, first, malleable jobs running on the considered cluster sorted in the increasing order of their start time (line 1), then the value of the number of processors to be allocated on behalf of malleable jobs (i.e. *growValue*) is offered to the subsequent job in the sorted list (line 3). In reply to this offer (the job itself considers its maximum number of processors requirement), the *accepted* number of processors are allocated (lines 4 – 5) on behalf of that job. Then the *growValue* is updated and checked whether there are more processors to be offered (lines 6 – 8).

The *shrink* procedure runs in a similar fashion; the differences with the *grow* procedure is that the jobs are sorted in the decreasing order of their start time (line 1), and rather than allocation, the *accepted* number of processors are waited to be released (line 5).

2) *Equi-Grow & Shrink (EGS)*: Our EGS policy attempts to balance processors over malleable jobs. Figure 5 gives the pseudo-code of the grow and shrink procedures of that policy. When it is initiated by the malleability manager, it distributes available processors (or reclaims needed processors) equally

```

procedure FPSMA_GROW(clusterName, growValue)
1. List  $\leftarrow$  malleable jobs running on
   clusterName, sorted in the increasing order
   of their start time
2. for each (Job in List) do
3.   send grow request (growValue) to Job
4.   get accepted number of processors from
   Job
5.   initiate processor allocation for Job
6.   growValue = growValue - accepted
7.   if growValue == 0 then
8.     break

procedure FPSMA_SHRINK(clusterName, shrinkValue)
1. List  $\leftarrow$  malleable jobs running on
   clusterName, sorted in the decreasing order
   of their start time
2. for each (Job in List) do
3.   send shrink request (shrinkValue) to Job
4.   get accepted number of processors from
   Job
5.   wait for Job to release the processors
6.   shrinkValue = shrinkValue - accepted
7.   if shrinkValue == 0 then
8.     break

```

Fig. 4. Pseudo-code of the FPSMA policy

(line 2) over all of the running malleable jobs. In case the number of processors to be distributed or reclaimed is not divisible by the number of running malleable jobs, the remainder is distributed across the least recently started jobs as a *bonus* (line 5), or reclaimed from the most recently started jobs as a *malus* (line 5).

The EGS policy is similar to the well-known equipartition one. The two policies however differ in the following points. While our EGS policy distributes equally available processors among running jobs, the equipartition policy distributes equally the whole set of processors among running jobs. Consequently, EGS is not expected to make at each time all of the malleable jobs have the same size, while equipartition does. But equipartition may mix grow and shrink messages, while EGS consistently either grows or shrinks all of the running jobs.

VI. EXPERIMENTAL SETUP

In this section we describe the setup of the experiments we have conducted in order to evaluate the support and the scheduling policies for malleable jobs in KOALA. We present the applications that have been used in Section VI-A, our testbed in Section VI-B, and the details of the workloads in Section VI-C.

A. Applications

For the experiments, we rely on two applications that we have previously made malleable with DYNACO. These applications are the NAS Parallel Benchmark FT [29], which is a benchmark for parallel machines based on a fast Fourier transform numerical kernel, and GADGET 2 [30], which is a legacy n -body simulator. Further details on how we have made

```

procedure EQUI_GROW(clusterName, growValue)
1. List  $\leftarrow$  malleable jobs running on
   clusterName, sorted in the increasing order
   of their start time
2. jobGrowValue  $\leftarrow$   $\lfloor$ growValue/size(List) $\rfloor$ 
3. growRemainder  $\leftarrow$  remainder(growValue, size(List))
4. for each (Job in List) do
5.   bonus  $\leftarrow$  1 if  $i <$  growRemainder;
   0 otherwise
6.   send grow request (jobGrowValue + bonus)
   to Job
7.   get accepted number of processors from
   Job
8.   initiate processor allocation for Job

procedure EQUI_SHRINK(clusterName, shrinkValue)
1. List  $\leftarrow$  malleable jobs running on
   clusterName, sorted in the decreasing order
   of their start time
2. jobShrinkValue  $\leftarrow$   $\lfloor$ shrinkValue/size(List) $\rfloor$ 
3. shrinkRemainder
    $\leftarrow$  remainder(shrinkValue, size(List))
4. for each (Job in List) do
5.   malus  $\leftarrow$  1 if  $i \geq$  shrinkRemainder;
   0 otherwise
6.   send shrink request (shrinkValue + malus)
   to Job
7.   get accepted number of processors from
   Job
8. for each (Job in List) do
9.   wait for Job to release the processors

```

Fig. 5. Pseudo-code of the EGS policy

malleable these two applications can be found in [2]. Figure 6 shows how the execution times of the two applications scale with respect to the number of machines on the Delft cluster (see table I). With 2 processors, GADGET 2 takes 10 minutes, while FT lasts 2 minutes. The best execution times are respectively 4 minutes for GADGET 2 and 1 minute for FT.

While GADGET 2 can execute with an arbitrary number of processors, FT only accepts powers of 2. As we have already stated, we propose that the scheduler does not care about such constraints, in order to avoid to make it implement an exhaustive collection of possible constraints. Consequently, when responding to grow and shrink messages, the FT application accepts only the highest power of 2 processors that does not exceed the allocated number. Additional processors are voluntarily released to the scheduler. In addition, the FT application assumes processors of equal compute power, while GADGET 2 includes a load-balancing mechanism.

B. The Testbed

Our testbed is the Distributed ASCI Supercomputer (DAS3) [9], which is a wide-area computer system in the Netherlands that is used for research on parallel, distributed, and grid computing. It consists of five clusters of 272 dual AMD Opteron compute nodes. The distribution of the nodes over the clusters is given in Table I. Besides using the 1 and 10 Gigabit/s Ethernet, DAS3 employs the novel local high-

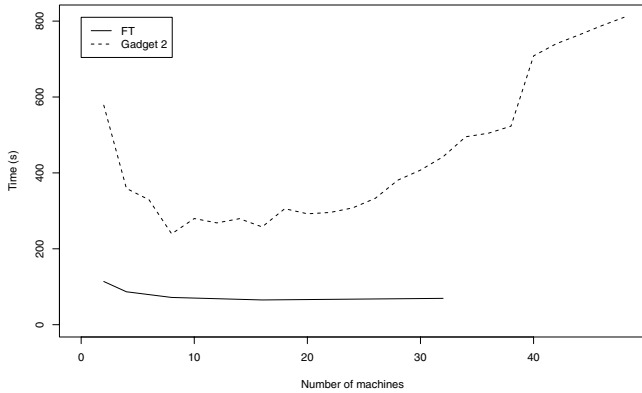


Fig. 6. The execution times of the two applications depending on the number of used machines.

TABLE I
THE DISTRIBUTION OF THE NODES OVER THE DAS CLUSTERS.

Cluster Location	Nodes	Interconnect
Vrije University	85	Myri-10G & 1/10 GbE
U. of Amsterdam	41	Myri-10G & 1/10 GbE
Delft University	68	1/10 GbE
MultimediaN	46	Myri-10G & 1/10 GbE
Leiden University	32	Myri-10G & 1/10 GbE

speed Myri-10G interconnect technology. On each of the DAS clusters, the Sun Grid Engine (SGE) [31] is used as the local resource manager. SGE has been configured to run applications on the nodes in an exclusive fashion, i.e., in space-shared mode. The granularity of allocation is the node.

C. The Workloads

The workloads that we employ in our experiments combine the two applications of Section VI-A with a uniform distribution. Their minimum size is set to 2 processors, while the maximum size is 46 for GADGET 2 and 32 for FT. In both cases, 300 jobs are submitted. Jobs are submitted from a single client site; no staging operation is ordered even when processors are allocated from remote sites.

Regarding Figure 6, the maximum sizes we have chosen are greater than the sizes for which we have observed the minimum execution times. This deliberate choice comes from the following. Applications are not supposed to scale the same in all of the clusters, which may be heterogeneous. In addition, users may not be aware of the speedup behavior of their applications. Hence, the maximum size of a malleable job should not be the size that gives to the best execution time of the application in any particular cluster.

For the PRA-based experiments, we have used two following workloads. Workload W_m is composed exclusively of malleable jobs, while workload W_{mr} is randomly composed of 50% of malleable jobs and 50% rigid jobs. In both cases, inter-arrival time is 2 minutes. Rigid jobs are submitted with a size of 2 processors, and malleable jobs with an initial size of 2. In our experiments, KOALA employs the WF policy.

Apart from workload W_m or W_{mr} , the only background load during the experiments is the activity of concurrent users. This background load does not disturb the measures.

When analysing the PWA approach, we have used two workloads W'_m and W'_{mr} , which derive respectively from W_m and W_{mr} . In these workloads, inter-arrival time is reduced down to 30 seconds in order to increase the load of the system.

VII. EXPERIMENTAL RESULTS

In this section we will present the results of our experiments for both the Precedence to Running Applications and the Precedence to Waiting Applications approaches.

A. Analysis of the PRA approach

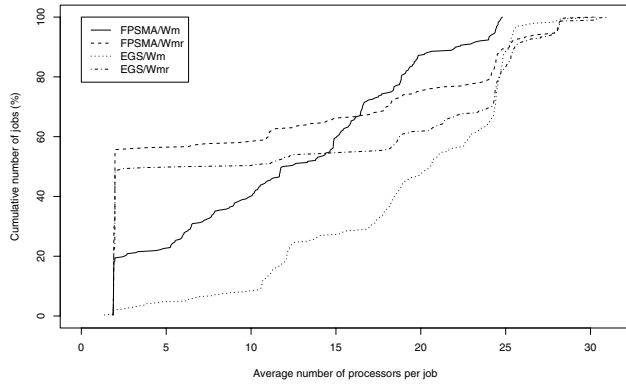
Figure 7 compares the FPSMA and EGS policies for malleability management in the context of the PRA approach for job management, i.e., when jobs are never shrunk. For this experiment, we have done 4 runs for each combination of a malleability management policy (one of FPSMA or EGS) and a workload (either W_m or W_{mr}).

Figures 7(a) and 7(b) show for each combination how jobs are distributed with regard to their average and maximum size. In both figures, with workload W_{mr} , which has 50% rigid jobs with only 2 processors, relatively few malleable jobs retain their initial size of 2 during their execution. In addition, we observe that among the policies, EGS one tends to give more processors to the malleable jobs than FPSMA, both in average and in maximum. Indeed, on the one hand, with FPSMA, short applications (like FT in our experiments) may terminate before it is their turn to grow, i.e., before previously started jobs terminate. They are thus stuck at their minimal size. On the other hand, EGS makes all jobs grow every time it is initiated. Hence, even jobs that have been started recently grow, and only few jobs do not grow beyond their minimal size.

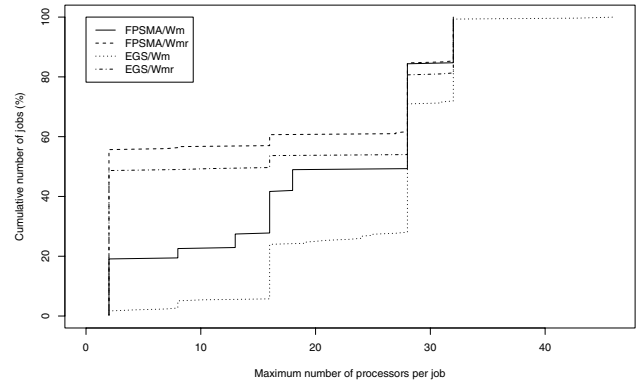
Figures 7(c) and 7(d) show the distributions of the execution time and the response time, respectively. Two groups of jobs appear clearly: those with execution times and response times less than 200 s, and those for which these times are greater than 400 s. Those two groups correspond to the two applications in the workloads (respectively FT and GADGET 2). In both cases, we observe that the W_m workload results in better performance than the W_{mr} workload, which means that malleability makes applications actually perform better.

Figure 7(e) shows the utilization of the DAS3 during a part of the experiments. With workload W_m with only malleable jobs, the EGS policy leads to a higher utilization. Indeed, as we have already observed, this policy tends to make bigger jobs. For the same reason, the utilization is better with workload W_m than with W_{mr} .

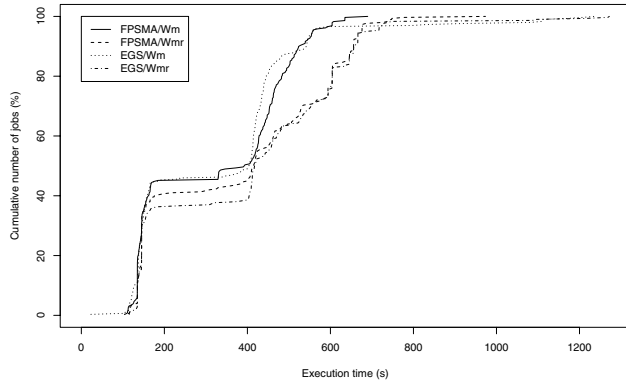
Finally, Figure 7(f) shows the activity of the malleability manager. As can be expected, the number of grow operations is much higher when all jobs are malleable (workload W_m). It is also higher with the EGS policy than with FPSMA. Indeed, each time the policy is triggered, EGS makes all of the running malleable jobs grow, while FPSMA only does so with the oldest ones.



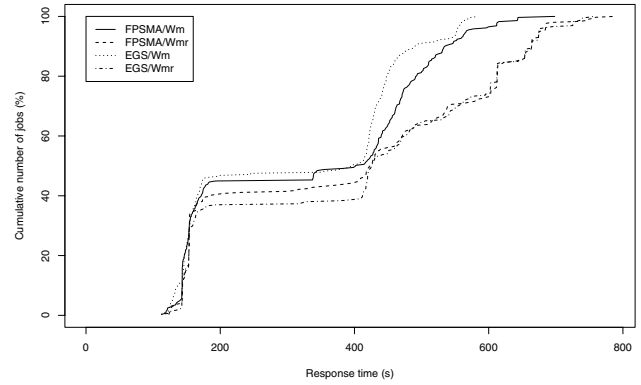
(a) The cumulative distribution of the number of processors per job averaged over the execution time of jobs.



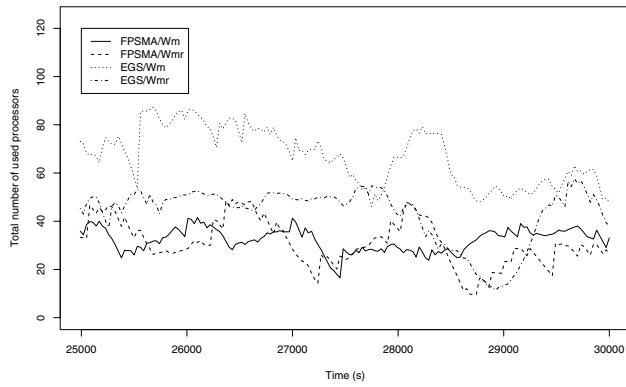
(b) The cumulative distribution of the maximal number of processors reached per job during its execution.



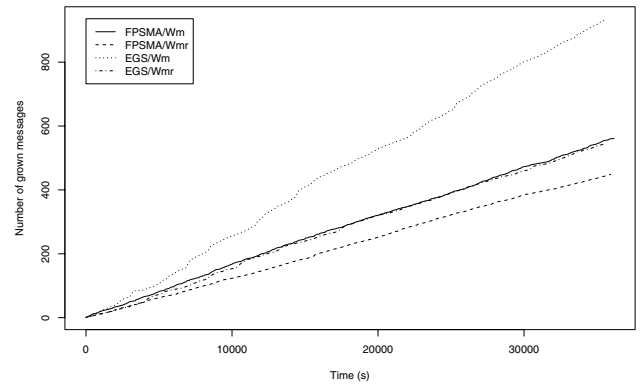
(c) The cumulative distribution of the job execution times.



(d) The cumulative distribution of the job response times.



(e) Utilization of the platform during the experiment.



(f) Activity of the malleability manager.

Fig. 7. Comparison between FPSMA and EGS with the PRA approach of job management (no shrinking).

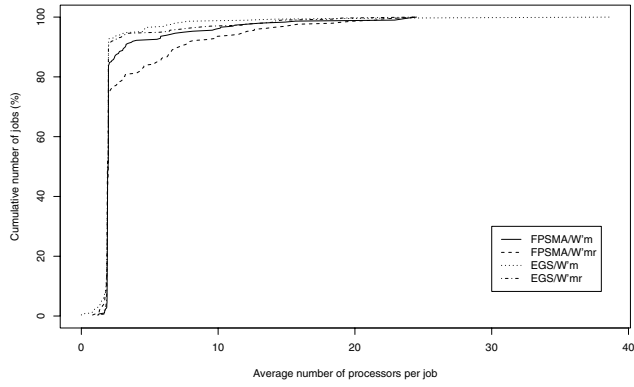
B. Analysis of the PWA approach

Figure 8 compares the FPSMA and EGS policies in the context of the PWA approach for job management, i.e., when the scheduler can also shrink jobs. With the PWA approach, the load of the system has a direct impact on the effectiveness of the malleability manager. Indeed, if on the one hand the system is overloaded, all of the jobs are stuck at their minimal size and malleability management becomes ineffective, while if on the other hand the system load is low, no job is shrunk and PWA behaves like PRA. We have therefore used workloads

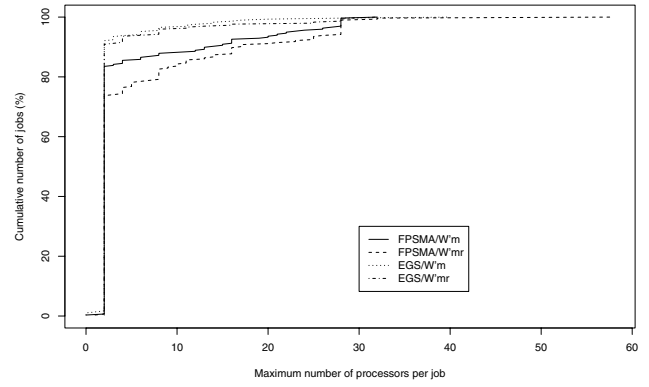
W'_m and W'_{mr} , which increase the load of the system.

Figure 8(f) shows that beyond a certain time, the malleability manager becomes unable to trigger any other change than initial placement of jobs. Similarly, Figures 8(a) and 8(b) show that many of the jobs are stuck at their minimal size, whatever the workload and the malleability management policy. This phenomenon is more pronounced with the EGS policy, which means that load balancing is achieved as expected.

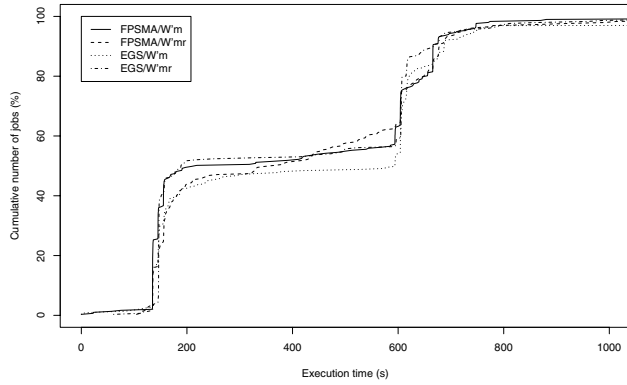
Figure 8(c) shows that the execution time is almost the same for the four runs. Most of the GADGET 2 job have an execution time of 600s, 30% higher than with PRA. This



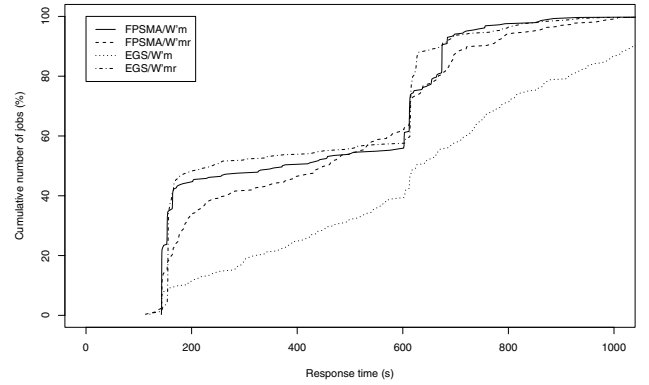
(a) The cumulative distribution of the number of processors per job averaged over the execution time of jobs.



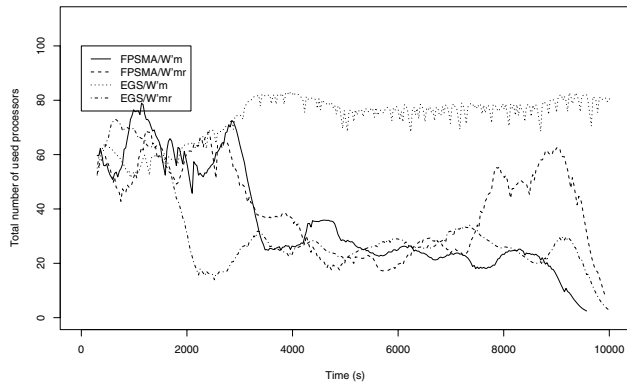
(b) The cumulative distribution of the maximal number of processors reached per job during its execution.



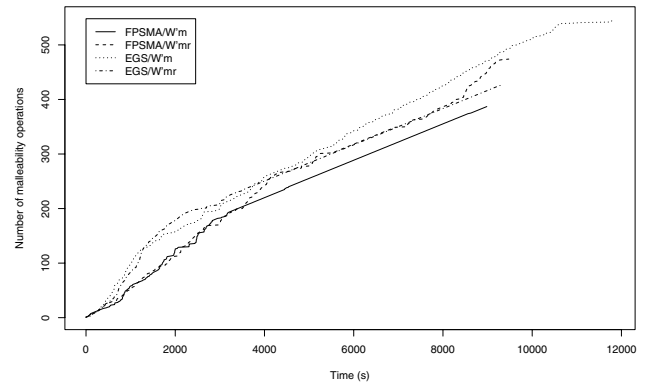
(c) The cumulative distribution of the job execution times.



(d) The cumulative distribution of the job response times.



(e) Utilization of the platform during the experiment.



(f) Activity of the malleability manager.

Fig. 8. Comparison between FPSMA and EGS with the PWA approach of job management (both growing and shrinking).

difference results from what we observe about the size of the jobs. Figure 8(d) shows that the response time is far worse for the combination of the EGS policy and W'_m workload due to higher wait time. This result confirms the system overload observed on Figure 8(e) as a high utilization. Favouring long-running jobs, FPSMA has reduced enough the execution time of GADGET 2 jobs to maintain the load sufficiently low.

VIII. CONCLUSION

In this paper we have presented the design and the analysis with experiments with the KOALA scheduler in our DAS3

testbed of the support and policies for malleability of parallel applications in multicluster systems. Our experimental results show that malleability is indeed beneficial for performance.

In case of mandatory shrinks as with our PWA policy, we have considered that it is the responsibility of the runner to enforce shrink operations. We have not experimented the behavior of the system in case the runner cannot be trusted to release the reclaimed resources. We plan in addition to study how to affect malleability management policies in order to incite applications to react to volunteer shrinks.

In our design, we have not included grow operations that

are initiated by the applications. This feature is mainly useful in case the parallelism pattern is irregular, unlike with our applications. Designing it is however not straightforward. For instance, it raises the design choice whether such grow operations can be mandatory or only voluntary, and how much effort the scheduler should do to accommodate mandatory grow operations, for instance by shrinking (either mandatorily or voluntarily) concurrent malleable jobs. Another element that we have not incorporated in our design and implementation but that we intend to add is malleability of co-allocated applications.

ACKNOWLEDGMENT

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl), which is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W), and which is part of the ICT innovation program of the Dutch Ministry of Economic Affairs (EZ). Part of this work is also carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

REFERENCES

- [1] H. Mohamed and D. Epema, "The design and implementation of the koala co-allocating grid scheduler," in *European Grid Conference*, ser. Lecture Notes in Computer Science, vol. 3470. Springer-Verlag, 2005, pp. 640–650.
- [2] J. Buisson, F. André, and J.-L. Pazat, "A framework for dynamic adaptation of parallel components," in *International Conference ParCo*, Málaga, Spain, Sept. 2005, pp. 65–72.
- [3] A. Iosup, D. Epema, C. Franke, A. Papaspyrou, L. Schley, B. Song, and R. Yahyapour, "On grid performance evaluation using synthetic workloads," in *Proc. of the 12th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, ser. Lecture Notes in Computer Science, E. Frachtenberg and U. Schwiegelshohn, Eds. Springer Verlag, June 2006.
- [4] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov, "Adaptive computing on the grid using apples," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 4, pp. 369–382, Apr. 2003.
- [5] S. Vadhiyar and J. Dongarra, "Self adaptability in grid computing," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 235–257, Feb. 2005.
- [6] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo, "Assist as a research framework for high-performance grid programming environments," Università di Pisa, Dipartimento di Informatica, Tech. Rep. TR-04-09, Feb. 2004.
- [7] L. Kalé, S. Kumar, and J. DeSouza, "A malleable-job system for time-shared parallel machines," in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, Berlin, Germany, May 2002, p. 230.
- [8] G. Utrera, J. Corbalá, and J. Labarta, "Implementing malleability on mpi jobs," in *13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Antibes, France, Sept. 2004, pp. 215–224.
- [9] "The Distributed ASCI Supercomputer," <http://www.cs.vu.nl/das3/>.
- [10] D. G. Feitelson and L. Rudolph, "Toward convergence in job schedulers for parallel supercomputers," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Springer-Verlag, 1996, pp. 1–26. [Online]. Available: citeseer.ist.psu.edu/feitelson96toward.html
- [11] G. Mounié, C. Rapine, and D. Trystram, "Efficient approximation algorithms for scheduling malleable tasks," in *11th ACM symposium on Parallel Algorithms and Architectures*, Saint-Malo, France, June 1999, pp. 23–32.
- [12] J. Blazewicz, M. Machowiak, G. Mounié, and D. Trystram, "Approximation algorithms for scheduling independent malleable tasks," in *7th International Euro-Par Conference*, Manchester, UK, Aug. 2001, pp. 191–197.
- [13] J. Blazewicz, M. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz, "Preemptable malleable task scheduling problem," vol. 55, no. 4, pp. 486–490, Apr. 2006, brief contribution.
- [14] T. Desell, K. E. Maghraoui, and C. Varela, "Malleable components for scalable high performance computing," in *HPC Grid programming Environment and COmponents*, Paris, France, June 2006.
- [15] S. Vadhiyar and J. Dongarra, "Srs: a framework for developing malleable and migratable parallel applications for distributed systems," *Parallel Processing Letters*, vol. 13, no. 2, pp. 291–312, 2003.
- [16] C. McCann and J. Zahojan, "Processor allocation policies for message-passing parallel computers," in *ACM SIGMETRICS conference on Measurement and Modeling of Computer Systems*, Nashville, Tennessee, USA, May 1994, pp. 19–32.
- [17] J. Hungershofer, A. Streit, and J.-M. Wierum, "Efficient resource management for malleable applications," Paderborn Center for Parallel Computing, Tech. Rep. TR-003-01, Dec. 2001.
- [18] J. Hungershofer, "On the combined scheduling of malleable and rigid jobs," in *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Foz do Iguacu, Brazil, Oct. 2004, pp. 206–213.
- [19] I. Foster and C. Kesselman, "Globus: a metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications*, vol. 11, no. 2, pp. 115–128, 1997.
- [20] H. Mohamed and D. Epema, "An evaluation of the close-to-files processor and data co-allocation policy in multicusters," in *2004 IEEE International Conference on Cluster Computing*. IEEE Society Press, 2004, pp. 287–298.
- [21] —, "Experiences with the Koala co-allocating scheduler in multicusters," in *Proceedings of the international symposium on Cluster Computing and the GRID*, Cardiff, UK, May 2005.
- [22] A. I. D. Bucur and D. H. J. Epema, "The maximal utilization of processor co-allocation in multicuster systems," in *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 60.1.
- [23] O. Sonmez, H. Mohamed, and D. Epema, "Communication-aware job placement policies for the koala grid scheduler," in *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2006, p. 79.
- [24] M. Aldinucci, F. André, J. Buisson, S. Campa, M. Coppola, M. Danelutto, and C. Zoccolo, "An abstract schema modelling adaptivity management," in *Integrated Research in GRID Computing*, S. Gorbach and M. Danelutto, Eds. Springer, 2007, proceedings of the CoreGRID Integration Workshop 2005.
- [25] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.
- [26] J. Buisson, F. André, and J.-L. Pazat, "Afpac: Enforcing consistency during the adaptation of a parallel component," *Scalable Computing: Practice and Experience (SCPE)*, vol. 7, no. 3, pp. 83–95, Sept. 2006.
- [27] K. van Reeuwijk, R. van Nieuwoort, and H. Bal, "Developing Java grid applications with Ibis," in *International Euro-Par Conference*, Lisbon, Portugal, Sept. 2005, pp. 411–420.
- [28] J. Buisson, F. André, and J.-L. Pazat, "Supporting adaptable applications in grid resource management systems," in *8th IEEE/ACM International Conference on Grid Computing*, Austin, USA, Sept. 2007.
- [29] R. F. van der Wijngaart, "Nas parallel benchmarks version 2.4," NASA Advanced Supercomputing division, Tech. Rep. NAS-02-007, Oct. 2002.
- [30] V. Springel, "The cosmological simulation code gadget-2," *Monthly Notices of the Royal Astronomical Society*, 2005, submitted astro-ph/0505010.
- [31] "Sun grid engine," <http://gridengine.sunsource.net>.