Scientific Programming 12 (2004) 263–273
IOS Press

Dynamic grid scheduling with job migration and rescheduling in the GridLab resource management system

K. Kurowski, B. Ludwiczak, J. Nabrzyski, A. Oleksiak and J. Pukacki
Poznan Supercomputing and Networking Center, Poland
E-mai: {krzysztof.kurowski, bogdanl, naber, ariel, pukacki}@man.poznan.pl

Abstract. Grid computing has become one of the most important research topics that appeared in the field of computing in the last years. Simultaneously, we have noticed the growing popularity of new Web-based technologies which allow us to create application-oriented Grid middleware services providing capabilities required for dynamic resource and job management, monitoring, security, etc. Consequently, end users are able to get easier access to geographically distributed resources.
In this paper we present the results of our experiments with the Grid(Lab) Resource Management System (GRMS), which acts on behalf of end users and controls their computations efficiently using distributed heterogeneous resources. We show how resource matching techniques used within GRMS can be improved by the use of a job migration based rescheduling policy. The main aim of this policy is to shorten job pending times and reduce machine overloads. The influence of this method on application performance and resource utilization is studied in detail and compared with two other simple policies.

1. Introduction

Grid environments are dynamic by nature. Heterogeneity, the high probability of failures, latencies connected with wide area networks, and the lack of dedicated access to resources, knowledge about local policies and jobs' runtimes can all cause a high degree of variance and unpredictability of application performance and resource utilization.

Consequently, the efficient scheduling of jobs before their submission often turns out to be very difficult to achieve. It appears that rescheduling methods, which take advantage of a migration mechanism, may provide a good way of improving performance [11,12,16].

Depending on the goal that is to be achieved using the rescheduling method, the decision to perform a migration can be made on the basis of a number of events. For example the rescheduling process in the GrADS project [11] consists of two modes: *migrate on request* (if application performance degradation is unacceptable) and *opportunistic migration* (if resources were freed by recently completed jobs). A performance oriented migration framework for the Grid, described in [16], attempts to improve the response times for individual applications. Another tool that uses adaptive scheduling and execution on Grids is the *GridWay* framework [12]. In the same work, the migration techniques have been classified into the application-initiated and *grid-initiated* migration. The former category contains the migration initiated by application performance degradation and the change of application requirements or preferences (self-migration). The grid-initiated migration may be triggered by the discovery of a new, better resource (opportunistic migration [14]), a resource failure (failover migration), or a decision of the administrator or the local resource management system.

Nevertheless, none of the aforementioned rescheduling algorithms focus on the migration initiated by a lack of free resources required by a new incoming application. Since this case is especially important for the high rate of incoming applications and for limited resources, we decided to explore it in detail.

In this paper, we present the rescheduling method implemented in the Grid(Lab) Resource Management

ISSN 1058-9244/04/$17.00

System (GRMS) [1] being developed in the framework of the GridLab project [2]. This method is applied when a job pending in the head of the GRMS queue cannot be submitted to any of machines for lack of the sufficient amount of required resources. The main difference, compared with the other rescheduling methods presented above, is that in our approach, in order to improve application performance, another application is migrated (to release required resources). Before the job is migrated, GRMS asks the application to checkpoint and terminate. Application-level checkpointing was used in our tests.

We compare this approach with two simple policies: first, submitting the job regardless of insufficient available resources (in this way overloading machines), and second, postponing the job submission until required resources are released.

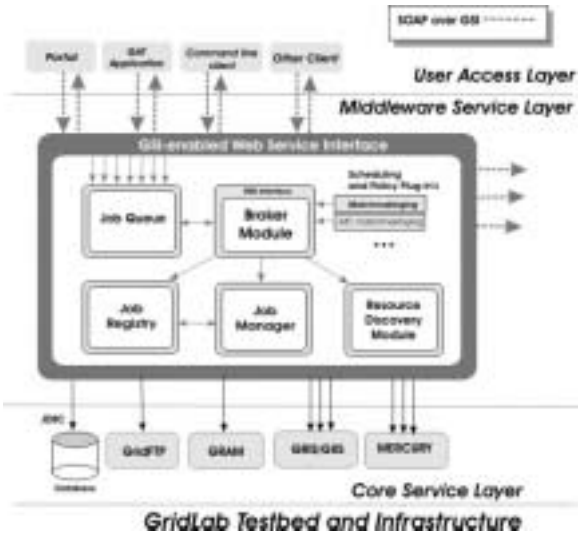

Fig. 1. GRMS v1.8.0 Architecture.

2. GRMS

GRMS is an open source meta-scheduling system for large scale distributed computing infrastructures. Based on the dynamic resource selection, mapping and advanced grid scheduling methodologies, combined with feedback control architecture, it has been tailored to deal with resource management challenges in Grid environments, e.g. load-balancing among clusters, setting up execution environments before and after job execution, remote job submission and control, file staging, and more. For our tests, we have used version 1.8.0 of GRMS, which is based on the Globus Toolkit 2.4 [3] and makes use of low-level Globus Services deployed on resources located in various academic institutions in Europe and the USA [4]. GRMS connects to the core services through a set of Java and C APIs. In particular, GRMS uses GRAM, GridFTP and GRIS/GIIS services. As a persistent service, GRMS provides a set of well-defined GSI-enabled Web Service interfaces for various clients, e.g. applications, command-line clients or portals. Moreover, GRMS is able to take advantage of middleware services, e.g. the GridLab Authorization Service or Replica Management Services, as well as to interoperate with infrastructure monitoring tools such as the GridLab's Mercury Monitoring System [5]. Therefore, GRMS is in fact one of the main components of a grid middleware layer that can be organized in many different ways depending on the particular infrastructure and applications. The architecture of GRMS together with a set of its internal modules, namely Job Queue, Job Registry, Job Manager, Resource Discovery and a central unit called Broker Module, is presented in Fig. 1. The aim of the Broker Module is to control the whole process of resource and job management. The broker has been designed in such a way that it allows us to implement various scheduling and policy plug-ins. One of the plug-ins studied in this paper, called the *Reschedule plug-in*, is responsible for job migration and rescheduling within GRMS. It is also worth mentioning the Resource Discovery Module, which monitors the status of distributed resources. It uses flexible hierarchical access to both central (GIIS) and local information services (GRIS), in particular to the Mercury Monitoring System that basically extends the functionality of GRIS/GIIS services by adding more dynamic information about jobs and the usage of resources. The remaining GRMS modules are essential for maintaining system consistency.

3. Applications

In order to be able to migrate a job, it has to be checkpointable first. In general, we can distinguish between two kinds of checkpointing: system-level and application (user)-level checkpointing [6]. In the first case, a system managing jobs usually takes advantage of the operating system mechanisms to swap an image of the application (data and stack segments of processes, CPU and memory status, etc.) to the disk, and then to recover the computation at the point where the checkpoint was generated. Application-level checkpointing requires the application developer to implement mechanisms for

storing data to a checkpoint file. In other words, checkpointing is hard coded in the application. This kind of checkpointing is obviously much more portable. Thus, due to a high heterogeneity of resources, application-level checkpointing is more applicable in Grids. Technically speaking, all applications have to implement a relatively simple Web Service interface in order to be ready for GRMS checkpoint calls. Then, during the execution, the application must register its location by providing its Web Service addresses together with a GRMS_JOB_ID. The GRMS_JOB_ID is taken from an environment variable set up by GRMS during a submission process. Once GRMS receives this information, it is able to call the application and request it to checkpoint, and if necessary to migrate it to a different location. Note that the application developer has to implement all internal mechanisms to write a checkpoint file to the local disk when the application receives a checkpoint call from GRMS.

4. Scheduling policies

As already mentioned, we have compared three different scheduling policies applied to jobs that should not be submitted due to the lack of free requested resources:

- *Overload*. Submit the job regardless of the insufficient amount of resources (overload a machine).
- *Wait*. Keep the job pending in the queue until required resources are released.
- *Reschedule*. Reschedule running jobs using migration in order to release resources needed to submit the pending job.

There are also other policies that can be used to avoid performance deterioration during the processing of jobs in the queue. One of these policies is the backfilling mechanism studied in [15] and used in many existing scheduling systems, e.g. LSF [7] or Sun N1 Grid Engine 6.0 [8]. As we decided to focus on adaptive improvements of the schedule using job migration, these techniques do not come within the scope of this paper. Nevertheless, we plan to investigate them in the future. All three considered policies are presented in detail in the following sections.

4.1. Wait policy

The wait policy keeps a pending job in the GRMS queue if it cannot be submitted to any of the machines because the amount of free resources required for this job is insufficient. GRMS periodically checks the status of resources and their availability, and if the sufficient amount is released, the job pending in the head of GRMS queue is submitted to the best available machine.

4.2. Overload policy

The *overload* policy enables submitting a job regardless of the insufficient amount of free resources. The advantage of this approach is that jobs do not have to wait in the queue. On the other hand, it obviously leads to the overloading of selected machines. If the number of jobs is reasonable and intervals between arrivals of jobs are big enough, drawbacks concerning the decrease of performance resulting from the overload of machines may be less significant than those concerning long queue pending times. Furthermore, operating systems are able to manage a larger amount of the physical memory than is actually available taking advantage of swapping techniques.

4.3. Reschedule policy

The reschedule policy checkpoints and migrates already running jobs in order to release the amount of resources required by a job pending in the queue. The rescheduling method that we used in the experiment consists of several steps. These steps, and the evaluation criteria for the selection of a job to be migrated, are presented in the following three subsections.

4.3.1. Steps of rescheduling

The rescheduling process includes the following steps:

Discover resources. First, resources that a user has an access to are selected. Note, that GRMS acts and performs all operations on resources on behalf of particular user. We assume in our experiments that rescheduling and migration operations apply to jobs submitted by a single user. However, these techniques can be easily used in a multi-user GRMS mode.

After selection of resources the user has access to, we filter the resources that meet all application's resource requirements expressed by the user within the GRMS job description. The hypothetical GRMS job request containing a requirement for 100 MB free memory available is presented below in Fig. 2. If no resources are found, then it is not possible to run the job without decreasing the performance of the job. Sometimes,

```
<grmsjob appid = "GRMS_Example_Job">
 <simplejob>
  <resource>
    <memory>100</memory>
    <ostype>linux</ostype>
    <osversion>redhat9</osversion>
  </resource>
  <executable type="single" count="1">
    <file name="exec-file" type="in">
     <url>gsiftp:// helix.bcvc.lsu.edu/
          {$HOME}/test/gatapp</url>
     </file>
    <arguments>
      <value>INPUT_1</value>
      <value>INPUT_2</value>
    </arguments>
    <checkpoint>
     <file name="checkpointFile.txt" type="in">
       <url></url></file>
     </checkpoint>
    <stdout>
<url>gsiftp://rage1.man.poznan.pl/{$HOME}/OUTPUT.txt</url></s
tdout>
<stderr><url>gsiftp://rage1.man.poznan.pl/{$HOME}/STDERR.txt
</url></stderr>
</executable>
</simplejob>
</grmsjob>
```

Fig. 2. The example of GRMS job description and the application's resource requirement for 100MB of free memory.

however, it is possible to find resources for the job by migrating other, usually smaller, jobs from busy to less busy resources. Of course, using migration in this case is often very risky, because we do not have any guarantee that no new job is started on the released resource after migration. This can be guaranteed either by advance reservation or if the Grid scheduler performing migration and job scheduling is the only entry point to the grid.

Discover resources meeting relaxed requirements. If no resources are found in the preceding step, the requirements concerning dynamic parameters are transformed into requirements concerning the corresponding static parameters. For instance, instead of free memory, the total amount of the physical memory is taken into consideration to check if particular resources meet the application's requirements. This step is just done in order to decide whether there is a potential for migration. The pending job can be potentially executed on these resources if some of the running jobs can be migrated to another machine.

Select jobs to migrate. In the next step, the system tries to determine the migration of which jobs can bring the required result, if at all. This step consists of two actions. First, GRMS searches for jobs after termination of which the pending job could be started immediately. This is nothing more than a simple preemption mechanism on the grid and we call this sub-step a *preemption check*. Actually, in the next action (or sub-step) of this step we go even further. The jobs selected at the preemption check phase are analyzed again to check which of them can be migrated to other available resources, taking into account the requirements of these jobs and the resources available at the moment. Potentially one or more such jobs exist.

Choose the best job to migrate. As a result of the preceding step, the set of jobs for migration is selected. The best of them is chosen using two sets of evaluation criteria. The first set of criteria allows GRMS to evaluate the machines a job to be migrated runs on. Obviously, this evaluation is done from the perspective of the pending job. The second set allows GRMS to evaluate the jobs that can be migrated. The former set of criteria consists of memory, load, CPU count and CPU speed, while the latter includes the number of hosts a job can

migrate to, the size of the migrating job, and the job's runtime. Further details can be found in Section 4.3.2.

Checkpoint and terminate the application. In this step, the selected job is requested to checkpoint and exit.

Move the terminated job back to the queue. The terminated job is moved back to the beginning of the GRMS queue in order to be submitted to another machine.

Submit job. In this step the job waiting for the required resources is submitted to a machine. If this step is performed after moving the job to be migrated back to the queue, the pending job is submitted to the host from which the job to be migrated was selected. Otherwise, the machine is selected using the multicriteria resource selection algorithm presented in [13].

The diagram in Fig. 3 illustrates the sequence of foregoing steps.

4.3.2. Selection of a job for migration

An accurate evaluation of the job that is to be migrated is very important, since the process of checkpointing and migration may itself be very time-consuming. There are multiple factors that influence the performance of this process. Therefore, multicriteria methods are of great importance and perfectly fit such problems [13].

In our study we adopted the model in which solutions are ranked according to values of a multicriteria evaluation function. The following function was used to evaluate job migrations:

$$f(\overline{C}) = \frac{1}{\sum_{i=1}^{n} wi} w_i^* C_i \tag{1}$$

where n is a number of criteria, c is a vector of criteria, and w is a vector of weights expressing an importance of criteria.

We evaluated jobs that were to be migrated, both by evaluating the hosts after the release of resources by migrated jobs, and by evaluating the migration costs.

In our experiment, we used the following criteria for the evaluation of hosts:

- available memory: *memAvail*,
- mean load during the last 1, 5 and 15 minutes: *CPUload*,
- CPU count: *CPUcount*,
- CPU speed: *CPUspeed*.

The following set of criteria was applied to evaluate checkpointing and migration costs:

- the number of hosts a job can migrate to (to minimize the risk of a failure): *hostNum*,
- the size of a migrating job (memory allocated by this job): *jobSize*,
- the job's current runtime (in order to migrate jobs that will not finish soon): *runTime*.

The importance of every single criterion was expressed using weights as presented in formula Eq. (1). The following values of weights were set in our experiments: memAvail $= 3$, CPUload $= 2$, CPUcount $= 1$, CPUspeed $= 1$, hostNum $= 1$, jobSize $= 3$, and runTime $= 2$.

- For more details concerning multicriteria resource selection in Grids a reader is referred to [13].

5. Experiment

5.1. Computing environment

We performed our experiments in the real testbed of the GridLab project [4]. Machines that are a part of the testbed are located in various sites across Europe and the USA. They are presented in detail in Table 1.

5.2. Jobs

The set of jobs used in the experiments consisted of 40 single processor jobs with randomly set runtimes (the number of iterations) and resource requirements. In our experiments, we chose free physical memory as the resource jobs were competing for. However, the general algorithm also allows the analysis of other resources and even multiple kinds of resources simultaneously.

Intervals between arrivals of jobs, numbers of iterations and amounts of required free memory have been drawn at random using the uniform distribution.

Unless stated otherwise, results presented in the paper were obtained for the set of jobs characterized by the following parameters:

- memory requirements (in MB): mean $= 144$, standard deviation $= 84$, max $= 275$, min $= 60$;
- duration of jobs, i.e. number of iteration processed by applications: mean $= 12250$, standard deviation $= 5549$, max $= 22000$, min $= 4000$;
- intervals between jobs' arrival times (in minutes): mean $= 4.79$, standard deviation $= 2.26$, max $= 9$, min $= 1$.

Fig. 3. Sequence of steps during job submission.

5.3. *Measurements*

We measured a number of metrics concerning resource utilization and application performance in order to evaluate three studied policies. To measure these metrics we used several sources of information. From the Mercury system [5] developed in the scope of GridLab, we obtained metrics describing resources, e.g. free memory and machine load. The Globus MDS (GIIS/GRIS services) [3] provided us with static information about resources, e.g. CPU speed, CPU count, physical memory etc. Finally, we extracted application performance indices including job execution and pending times from the GRMS database, which contains necessary information about the jobs' history.

The following parameters have been measured:

- completion time of the last job (makespan);
- mean response time of jobs;
- mean free memory on hosts;
- mean CPU load on hosts;

– standard deviation of free memory on hosts;
– standard deviation of CPU load on hosts.

6. Results

The primary objective of this paper is to compare the efficiency of the rescheduling policy with the two remaining policies that do not use migration at all. Thus, most of the analyses in this section include a comparison of particular policies. However, we have also added a subsection devoted to changes of various metrics over time and studied the influence of job requirements and duration on the performance of rescheduling.

6.1. Performance of applications

As metrics to measure the performance of methods applied we used various time measures. In particular, the response time (during which the whole request is served) and makespan (the completion time of the last job) are very important metrics for an evaluation of the algorithms' efficiency.

A comparison of these metrics for all three policies is presented in Fig. 4. We can observe that the *reschedule* policy outperforms others as regards both the response time and the makespan. The difference concerning the response time is more significant than the difference between values of makespan for particular strategies. By using the *reschedule* policy we can avoid many long queue waiting times that increase the maximal and mean response time. Using the *reschedule* policy, the mean and maximal response times were improved by 23% and 30%, respectively, compared to the *wait* strategy.

Additionally, the *wait* strategy turned out to be more profitable for the job response times and makespan than the *overload* policy. The big makespan for the *overload* strategy resulted from several jobs running on dramatically overloaded machines. Even after completion of most of the jobs it was impossible to reduce a load without the use of a job migration. Furthermore, some of the jobs that were run using the *overload* policy were not even executed.

6.2. Utilization of resources

Resource utilization is a crucial issue for administrators and resource owners. Therefore, the influence of the analyzed policies on this performance index is illustrated in Fig. 5.

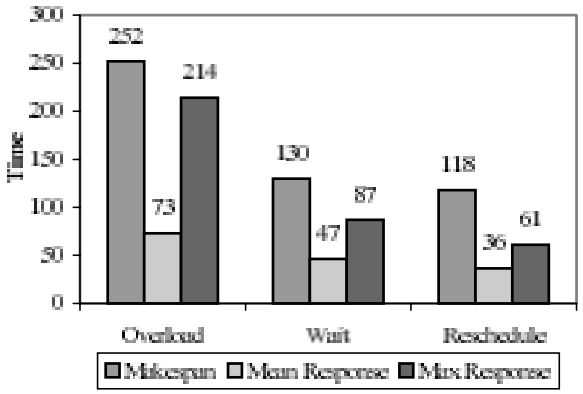

Fig. 4. Makespan, mean and max response times for the overload, wait and reschedule policies.

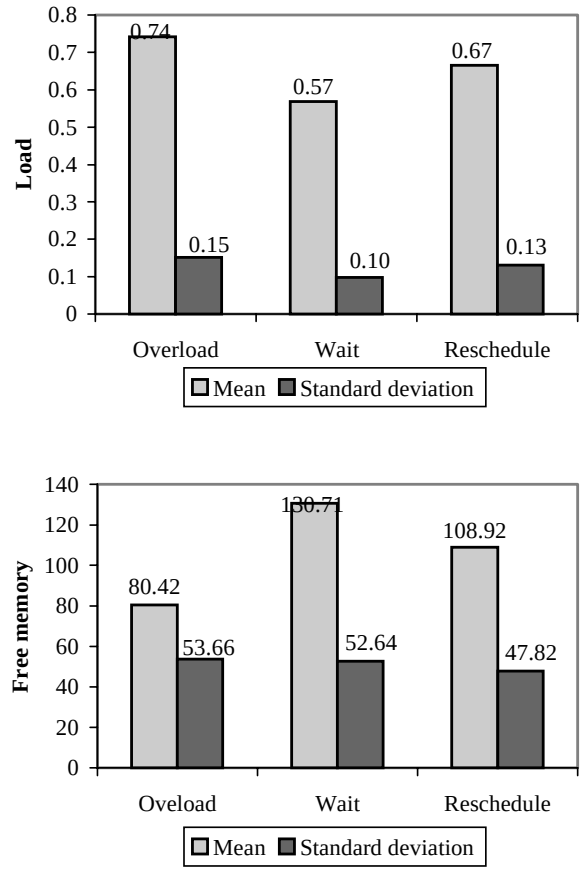

Fig. 5. Mean and standard deviation of free memory (below) and CPU load (above).

The *reschedule* policy made it possible to avoid extreme machine overloads, which lead to a considerable and unbalanced resource utilization, occurring especially in the case of the overload policy. On the

Table 1
Machines used in the experiment

Host	Country	CPU no	CPU speed	Phys. Mem
rage1.man. poznan.pl	Poland	2	Pentium-III 1.4 GHz	1.0 GB
fs0.das2.cs.vu.nl	Holland	2	Pentium-III 1.0 Ghz	2.0 GB
n0.hpcc.sztaki.hu	Hungary	2	PIII-.5G Hz	251 MB
rage2.man.poznan.pl	Poland	2	Pentium-III 1.4 GHz	1.0 GB
rage3.man.poznan.pl	Poland	2	Pentium-III 1.G Hz	1.0 GB
peyote.aei.mpg.de	Germany	4	Xeon 2.8 GHz	4.0 GB
gridentry.uni-paderborn.de	Germany	2	Pentium III 850 Mhz	512 MB
rage4.man.poznan.pl	Poland	2	Pentium-III 1. GHz	1.0 GB
helix.bcvc.lsu.edu	USA	4	Xeon 2.0 GHz	4.0 GB

other hand, the *wait* policy is characterized by lower resource utilization, since this method keeps jobs in the queue until the required resources are available. The resource utilization using the *reschedule* policy is reasonable: greater than for the *overload* and smaller than for the wait policy. Surprisingly, differences between the mean resource utilization for particular policies are rather reasonable. In the case of a load, the reason is that, we used only memory (not load) to decide about the migration to another host. Additionally, several jobs were running on the overloaded hosts long after a majority of other jobs finished, decreasing the overall load and memory usage in the case of the *overload policy*.

6.3. *Influence of resource requirements and execution times on the performance of policies*

To make the comparison of the policies more useful and detailed, we should determine classes of jobs for which particular methods perform better than for others. Therefore, we investigated the influence of job runtimes and resource requirements, i.e. "the size" of the job, on the performance of each studied policy.

Results presented in Fig. 6 confirmed our presumptions that the use of rescheduling brings the greatest benefits if it is used for relatively small and rather long jobs. Free memory and numbers of iterations given in Fig. 6 stand for the mean values for the whole set of jobs.

We observed that the especially good performance gain has been attained when the incoming job set consisting of many small long-lasting jobs was followed by the submission of jobs having large requirements (in our case concerning free memory).

6.4. *Changes of metrics over time*

It seems interesting to have a closer look at the way all policies behave. Let's see. Figures 7, 8 and 9 show how the measured performance changes over time.

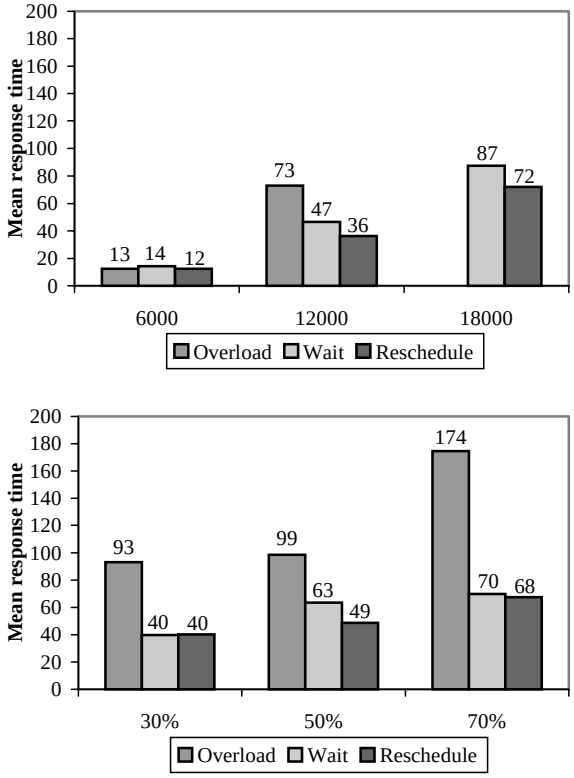

Fig. 6. Mean response time depending on application memory requirements: 30%, 50% and 70% of the mean memory available on hosts (below), and duration: 6000, 12000 and 18000 iterations (above).

The advantage of the *reschedule* policy revealed itself mainly when the test machines were saturated with incoming jobs, which means that new jobs submitted to GRMS could not be started immediately (peaks in the middle of Fig. 7). However, an excessive number of jobs decreased the performance of all policies, even the *reschedule* policy. The obvious reason of this is that jobs could not migrate to any other hosts for lack of free resources.

You can see in Fig. 8 the periods in which throughput decreased significantly for the *wait* policy. This phe-

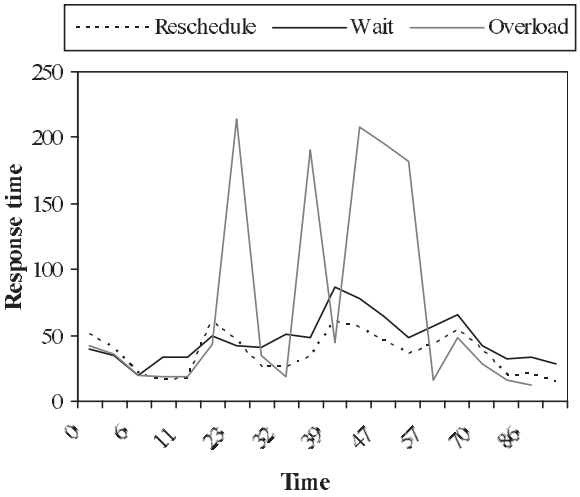

Fig. 7. Response time changing over time.

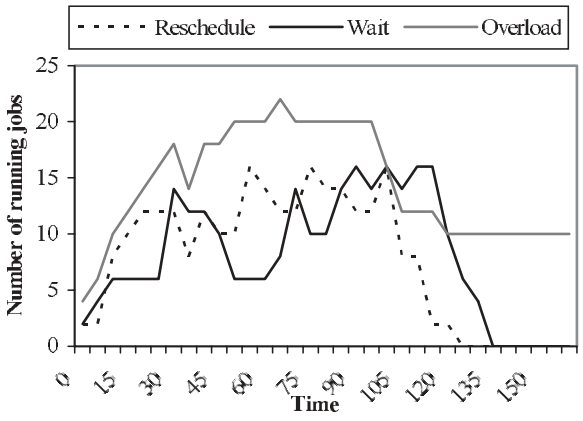

Fig. 8. Number of running jobs changing over time.

nomenon is related to the aforementioned long job waiting times. This effect did not appear to such an extent for the *reschedule* policy, since in this case jobs were rescheduled to enable waiting jobs to run (at the same time avoiding the overload of machines). Of course, throughput deteriorations also occur if jobs cannot be migrated to other hosts, but they are not as significant as for the *wait* policy (there are smaller "valleys" on the plot of the *reschedule* strategy in Fig. 8). We should also be aware that the steadily increasing number of running jobs in the case of the *overload* policy is not profitable, because it slows down job execution times. This is caused by the intense overload of machines.

Figure 9 shows that just as in the case of load and the job throughput, high resource utilization (concerning allocated memory) occurred for the *overload* policy. On the other hand, use of the *wait* policy led to the

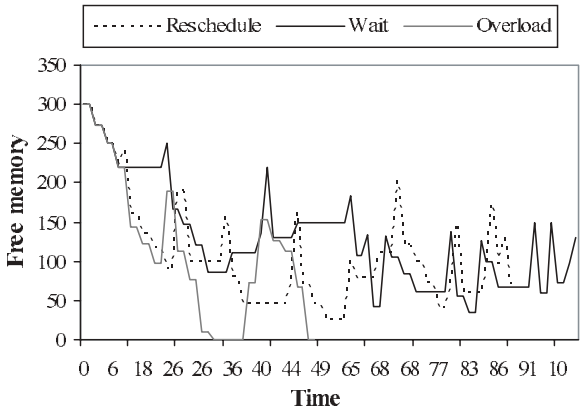

Fig. 9. Mean free memory changing over time.

lowest resource utilization (the biggest amount of free memory).

Comparing these results with the analysis of other metrics presented in this section, we can conclude that the best performance of the reschedule policy is connected with reasonable *resource* utilization: not too great (to avoid a machine overload) and not too small (to avoid excessively long waiting times for jobs in the queue).

7. Summary

7.1. Conclusions

In this paper, we presented our rescheduling method that aims to shorten queue waiting times in the Grid(Lab) Resource Management System (GRMS) and, consequently, decrease the application response times. We explored a migration that was performed due to the insufficient amount of free resources required by incoming jobs. Application-level checkpointing was used in order to provide full portability in the heterogeneous Grid environment. In our tests, the amount of free physical memory was used to determine whether there are enough available resources to submit the pending job. Nevertheless, the algorithm is generic, so we can easily incorporate other resources such as free processors, disc space, or even a network bandwidth.

Rescheduling techniques turned out to be very useful, especially for applications of a reasonable "size" and duration. Makespan, response times and resources utilization metrics were particularly improved in the case of sets of applications characterized by reasonable memory requirements and sufficiently long execution times. On the other hand, the rescheduling policy did

not work well for excessively small jobs, as they mostly fitted the available memory and did not take advantage of the migration technique. On the basis of our experiments, the optimal value of mean jobs' memory requirements appears to be about 50% of the memory available on hosts. Additionally, the memory requirements of applications should be diverse. Especially good performance gain was achieved when the set of incoming jobs consisting of many small long-lasting jobs was followed by the submission of jobs having larger requirements.

We should also emphasize that migration turned out to be useful and more efficient than the remaining two policies, although experiments were performed in the wide area network on the transatlantic testbed. Thus, benefits achieved by means of the reduction of queue waiting times and the migration of applications to the best available resources outstripped overheads connected with writing and reading the checkpoint files, application startup times, and the transfer of necessary data through the network.

The *reschedule* policy achieved the best performance due to reasonable resource utilization when compared with the remaining two strategies (greater than for the *wait* policy and lower than for the *overload* policy). The response time metrics (both the mean and maximal value) have been particularly improved using rescheduling. This is a real advantage of the rescheduling policy, since the response time is a very important metric for individual users submitting jobs to the Grid.

The rescheduling capability can be considered as a useful additional feature for performance improvement, since it is not suitable for all possible jobs' sizes, durations and sequences. The grid scheduler should try to schedule jobs as optimally as possible and then reschedule them dynamically, if this is both required and feasible. Furthermore, we have shown that the insufficient amount of free resources required by the incoming application can be a useful initiator of the migration, in addition to those investigated in other papers (and listed in Section 1), such as, for example, performance deterioration of the running application.

In general, the use of the job migration and rescheduling methods leads to the significant improvement of overall performance. Furthermore, this approach may be the only way to provide requested performance for end-users if job execution times and resource usage are not known a priori, as we assumed in our experiments. We conclude that dynamic rescheduling based on application-level checkpointing and migration is a strategy that should improve resource brokering and scheduling in heterogeneous and non-dedicated environments. Therefore these mechanisms are particularly useful for Grids.

7.2. Future research

In this paper we compared the rescheduling method with two very simple policies under the assumption that the environment does not provide any information about estimated job runtimes and queue waiting times. However, more advanced techniques can also be used, for example backfilling, which is a common method for improving job throughput [7,15]. In future research, we would like to compare these with rescheduling methods based on the job migration presented in this paper.

We also plan to make some improvements to the existing rescheduling algorithm. One of them is the attempt to prevent multiple migrations of the same job by using the history of migrations. Moreover, we are going to use the rescheduling algorithm for more than one type of resource simultaneously (supplementing the free memory with free processors and disc space, etc.). We would also like to involve more metrics (with a greater accuracy) to evaluate the selection of jobs to migrate, for example:

- estimations of the remaining job execution times,
- network metrics to estimate the costs of migration more precisely,
- exact sizes of the checkpoint files (instead of the memory allocated by applications).

In our tests we used the migration mechanism working between single hosts with public IP addresses. However, in grid environments, jobs submitted to local sites are usually run under the control of queuing systems. Internal nodes of clusters are often inaccessible from the outside due to private IP addresses or firewalls. Thus, migration between such clusters becomes more difficult. To enable this capability, we plan to apply the Mercury Monitoring System (used in the experiment to gather information about the state of resources) to checkpoint the application inside a cluster by sending an appropriate system signal. In this scenario, GRMS uses the Mercury Monitoring System's interfaces only on a front-end machine to send a checkpoint call. Another advantage of this approach is the possibility of checkpointing and migrating jobs between different queuing systems.

Other measures we are working on include the utility, stability and robustness of the new schedule obtained after rescheduling. This will allow us to provide methods for robust predictive and reactive scheduling on the grid.

Acknowledgments

We are pleased to acknowledge support from the EU GridLab project (IST-2001-32133).

References

[1] http://www.gridlab.org/WorkPackages/wp-9/.
[2] http://www.gridlab.org.
[3] http://www.globus.org.
[4] http://www.gridlab.org/WorkPackages/wp-5/testbed/gtest.html.
[5] http://www.gridlab.org/WorkPackages/wp-11/.
[6] https://forge.gridforum.org/projects/gridcpr-wg.
[7] http://www.platform.com/.
[8] http://wwws.sun.com/software/gridware/.
[9] F. Berman et al., Adaptive Computing on the Grid Using AppLeS, *IEEE Transactions on Parallel and Distributed Systems* **14**(5) (2003), 369–382.
[10] R. Buyya, D. Abramson and J. Giddy, A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker, *Future Generation Computer Systems*, Vol. 18 Elsevier Science, 2002, pp. 1061–1074.
[11] H. Dail, O. Sievert, F. Bermann et al., Scheduling in the Grid Application Development Software Project, in: *Grid Resource Management: State of the Art and Future Trends*, J. Nabrzyski, J. Schopf and J. Weglarz, eds, Kluwer Academic Publishers, 2003, pp. 71–98.
[12] E. Huedo, R. Montero and I. Llorente, *The GridWay Framework for Adaptive Scheduling and Execution on Grids*, In Proceedings of AGridM Workshop (in conjunction with the 12th PACT Conference, New Orleans (USA), Nova Science, October, 2003.
[13] K. Kurowski, J. Nabrzyski, A. Oleksiak and J, Weglarz, Multicriteria Aspects of Grid Resource Management, in: *Grid Resource Management*, J. Nabrzyski, J. Schopf and J. Weglarz, eds, Kluwer Academic Publishers, Boston/Dordrecht/London, 2003, pp. 275–296.
[14] R.S. Montero, E. Huedo and I.M. Llorente, Grid Resource Selection for Opportunistic Job Migration, *In Lecture Notes in Computer Science* **2790** (August, 2003), pp. 366–373.
[15] S. Srinivasan, R. Kettimuthu, V. Subramani and P. Sadayappan, *Characterization of Backfilling Strategies for Parallel Job Scheduling*, In Proceedings of 2002 International Workshops on Parallel Processing (held in conjunction with the 2002 International Conference on Parallel Processing, ICPP 2002), August, 2002.
[16] S. Vadhiyar and J. Dongarra, A Performance Oriented Migration Framework For The Grid, In Proceedings of CCGrid, *IEEE Computing Clusters and the Grid*, CCGrid 2003, Tokyo, Japan, May 12–15, 2003.