GRID SUPERSCALAR AND JOB MAPPING ON THE RELIABLE GRID RESOURCES

Ani Anciaux–Sedrakian, Rosa M. Badia, Raül Sirvent and Josep M. Pérez
Polytechnic University of Catalonia
Campus Nord - Modul D6 c/Jordi Girona 1-3 E08034 Barcelona, Spain
ani.anciaux@bsc.es
rosa.m.badia@bsc.es
raul.sirvent@bsc.es
josep.m.perez@bsc.es

Thilo Kielmann and Andre Merzky
Vrije Universiteit
De Boelelaan 1081A 1081HV Amsterdam, The Netherlands
kielmann@cs.vu.nl
andre@merzky.net

Abstract The dynamic nature of grid computing environment requires some predictions regarding resource reliability and application performance. In such environments, avoiding resource failures is as important as the overall application performance. The aim of this work is to identify the most available, least-loaded and fastest resources for running an application. We describe a strategy for mapping the application jobs on the grid resources using GRID superscalar [8] and GAT [1].

Keywords: Grid computing, reliability, high performance computing, GRID superscalar, Grid Application Toolkit.

1. Introduction

The number of applications that use grid computing systems are relatively limited. One of the blocking reasons is difficulty of their development. This is due to the intrinsic complexity of the programming interface in one hand and heterogeneity and dynamicity of grid environments on the other hand. The aim of this paper is to cope with complexity of grid applications development. In this context, we address two main issues: (i) the choice of computation resources for the application to obtain a robust and efficient execution, and (ii) the co-existence of distinct underlying middleware on the grid.

The first part of this work presents an approach to select compute resources by combining performance criteria (like computation and data transfer capacity) with predicted resource reliability. A grid environment offers a large numbers of similar or equivalent resources that grid users can select and use for their workflow applications. These resources may provide the same functionality, but offer different QoS properties. A workflow QoS constraint includes five dimensions: time, cost, quality, reliability and security [11]. The basic performance measurement is time (the first dimension), representing the total time required for completing the execution of a workflow. The second dimension represents the cost of workflows execution[1]. Quality refers to the measurement related to the quality of the output of workflow execution. Reliability is related to the probability of failures for execution of workflows. Finally, security refers to confidentiality of the execution of workflow jobs and the trustworthiness of resources (in [12] some studies in this field are represented).

This work is not only focused on the reliability dimension, in order to minimize failures, but also considers the time dimension. Mapping the applications jobs onto the most appropriate resources is a multi-criterial process and not always the performance is an issue but also reliability. Resources manually chosen by grid users may be the most powerful ones, but are not always the most reliable ones. The problem is how to map the jobs onto suitable resources, in order to minimize the probability of failure for execution of workflows. Workflow execution failures can occur for the following reasons: variation in the execution environment configuration, non-availability of required services or software components, overloaded resource conditions, system running out of memory, and errors in computational and network fabric components. Therefore, the natural way to maximize the reliability will be examining theses parameters which are relatively the potential reason of faults and avoiding them.

The second part of this paper shows how to provide a Grid programming environment that is both high-level and platform-independent. In general, grid

[1]It includes the cost related to the managing of workflow systems and usage charge of Grid resources for processing workflow jobs.

applications are restricted to one specific grid middleware package. Therefore, submitting unmodified existing application codes to remote grid resources may be faltering, since the desired middleware may not available in that remote resource. This runs contrary to the vary nature of grids, which imply a heterogeneous environment in which applications must run. In order to be effective, a grid application must be able to run in any environment in which it finds itself. Ideally grid applications would discover required grid services at run time, and use them as needed, independent of the particular interfaces used by the application programmer. To reach this goal, using a middleware which assume the platform-independent feature can solve this complexity of application development.

The remainder of this article is organized as follows: Section 2 describes how to select reliable and efficient resources. Section 3 first describes GRID superscalar, then reviews the implementation and integration of the proposed reliability strategy in GRID superscalar and at the end shows how to make GRID superscalar a platform independent runtime system. In section 4 some early experiments are presented. Finally section 5 concludes the paper.

2. Resource selection

Due to the variability in grid-computing environments, it is difficult to assume resources reliability: both load and availability of heterogeneous grid resources varies dynamically. As applications can have a wide variety of characteristics and requirements, there is no single best solution for mapping workflows onto known reliable resources for all workflow applications. This section describes a strategy to find the most reliable resources while also minimizing the overall application completion time, i.e. by maximizing jobs performance and/or minimizing communications time, in respect to the application characteristics.

Our approach for solving this problem starts with retrieving the information from available resources. This information may be requested before workflow execution starts, in order to help users take the right decision while choosing the appropriate resources. The proposed strategy, composed of two steps, is based on the ranking of the resources: in the first step, called trust-driven step, parameters related to the potential source of faults are collected. The resources are then ranked with respect to their ability to guarantee the maximum level of reliability. Ranking allows increasing the overall reliability of application execution. In the second step, called performance-driven step, the ranking of each resource is revisited by taking the performance criteria (computation power and data transfer capacity) into account. This compromise between reliability and performance leads to increase computation robustness and to decrease overall execution time. These two steps, which are done before execution, are described more in detail in the following part.

2.1 Trust-driven step

In the trust-driven step, some information and hardware requirements for the resources are retrieved, such as the availability of the resources, its trustfulness and the available amount of memory. Part of this information is stored in a database, in order to have a historical trace of resources state. This database can be located anywhere in the grid. The collected information is used for helping the users to find an appropriate resource where their jobs will most likely not fail. All this information may be gathered and stored either independently and/or before running the application. Obviously, the database could become more affluent if more information is retrieved about the hardware requirements.

During the trust-driven step, the availability of resources is computed first. It shows resource accessibility across an IP network. Let r_i denotes the i^{th} grid resource, na_i^j the number of times when the resource r_i was not available at the moment j and ta_i^j the total number of attempts to verify the resource availability. The availability of resources is defined as:

$$availability(r_i) = 1 - na_i^{j+1}/ta_i^{j+1} \tag{1}$$

$$na_i^{j+1} = na_i^j + dbna_i \ \ and \ \ ta_i^{j+1} = ta_i^j + dbta_i \tag{2}$$

Where $dbna_i$ and $dbta_i$ (in equation 2) is the information stored previously on the database. After the availability calculation, the database is updated; the new value of na_i^{j+1} and ta_i^{j+1} will be stored ($dbna_i = na_i^{j+1}$ and $dbta_i = ta_i^{j+1}$).

Afterwards, information concerning the resource trustfulness is collected. Trustfulness is evaluated by parameters which may generate some faults during the execution. In particular, we take into account resources dropping (i.e. job crashing during the execution due to the sudden unavailability of resource), execution environment variations, job manager failures (i.e. system cancel the job) and network failure (i.e. packets loss between resources). In this part of trust-driven step, a value (called distrust value) is assigned to each resource. The distrust value increases when the assigned resource does not meet the computation requirement and the job fails (due to the one of the mentioned parameters). For example, when a job has crashed during the execution, the distrust value of that resource will be increased. We consider that all the resources are trustable from the start (their background is blank). However in the course of the time the background of each resource will be evaluated and will confirm if its trustfulness is always maintained or not. Consequently, the resources with the lower distrust value are better matched for the components (workflow jobs) of the application. The trustfulness of resources is defined as below in
equation 3.

$$distrust(r_i) = trv_i^{j+1} = trv_i^j + dbtrv_i \tag{3}$$

Where trv_i^{j+1} represents the distrust value of i^{th} resource at the $j+1$ moment and $dbtrv_i$ is the distrust value of the resource r_i stored previously on the database.

At this point, resources are sorted with respect to the level of reliability they offer (using availability and trustfulness metrics). After all, the sorted resources will be qualified, by taking into consideration the amount of memory, which the application requires for the execution. The necessary amount of physical memory is computed using estimation on the number and the size of input, output and temporary files. Hence in order to avoid restarting the application, we first authenticated the most available and trustfulness resources, then we identified the resources with the available amount of physical memory. Once the reliability of resources is assumed known, a rank value is evaluated to each eligible resource: rrank (r_i). The resources with the smallest rrank value are the least reliable ones.

$$\text{if (rrank}(r_i) < \text{rrank}(r_j)) \rightarrow r_i \text{ is less reliable than } r_j$$

2.2 Performance-driven step

In the second step (performance-driven step), the challenge consists in selecting the eligible resources among the reliable ones, in order to obtain a high level of efficiency for the application. Therefore we need to choose the fastest and least-loaded resources where the data movement cost between resources[2] is the least.

Let $R = r_1, r_2, \ldots, r_n$ denotes the set of qualified grid resources and $T = t_1, t_2, \ldots, t_m$ designates the set of jobs running in the grid resources. The performance of j^{th} job (t_j) running in these eligible resources may be estimated by the following equation (equation number 4). It takes into consideration three crucial parameters having an impact on performance.

$$Time(t_j) = \mu \times ET(t_j, r_i) + \gamma \times FTT(t_j, r_i) + \delta(t_j, r_i) \tag{4}$$

The first parameter, $ET(t_j, r_i)$, presents the time required for completing the j^{th} job on the resource r_i. This time take into consideration the processor speed and memory access pattern.

The second parameter, $FTT(t_j, r_i)$, presents the spent time for transferring required data for running the j^{th} job in the resource r_i. The effectiveness of this parameter may be evaluated in two ways: either by using the time to transfer the required data for the job t_j; or by the considering the time of a round trip

[2]The data movement cost between master and the remote hosts contributes plainly to the overall execution time.

(source to target r_i) of packets on the network. In the first case this amount may calculate in the following way : $FTT(t_j, r_i) = L(r_j) + vol(t_j)/B(r_j)$ where $vol(t_j)$ presents the amount of the required data for t_j. L(r_j) and B(r_j) present respectively the latency and bandwidth of network estimated via NWS [10] or netperf[3]. In the second case this amount may estimate by using ping program which transfers some data in bytes.

Finally, the last parameter δ (r_i) describes queue waiting time which can increase the overall execution time of an application[4], which prompts a more detailed study. In order to retrieve the information regarding the application waiting time in a queue, we use the approach proposed in Delphoi [7] which implements an appropriate strategy to predict this waiting time. The application waiting time may in general depend on both the application size (i.e., the number of hosts required to run it) and the queue load. For this reason, the proposed strategy forecasts three categories (fully used and normally used and empty queue), where each of them uses three classes of application sizes, small (1 to 4 hosts), medium (5 to 16 hosts), and large (17 or more hosts). By taking application size and queue load into account, an average waiting time can be predicted before running the application, allowing to determine the least loaded queue at the runtime. This parameter helps the user to claim a resource with small estimated response times.

In the equation number 4, both γ and μ weighted parameters are specified to give more importance to data transfer time or to the job execution time, depending on the respective application requirement.

Hence, in order to find the resources ensuring the least data transfer time , in this step, information such as processor speed, network related characteristic (the ping program outputs, network latency and bandwidth) and charge of resources is retrieved.

Once the execution time is estimated, another rank value is assigned to each resource called prank (r_i) which expresses the power of each resource. In this case the resource r_i is more powerful when its prank value is bigger, i.e. when its estimated execution time for the job is smaller.

$$\text{if (prank}(r_i) < \text{prank}(r_j)) \rightarrow r_i \text{ is less powerful than } r_j$$

The major issue now is to find the most powerful resource among the most reliable ones, as the most reliable resources are not necessarily the most powerful ones. The proposed policy is designed to find a compromise between these two metrics. For this purpose, we give another rank value to the resources called grank (r_i). This value is a weighted linear combination of rrank (r_i) and prank (r_i) computed as follows, where α and β are the weights, which can be

[3]http://www.netperf.org/netperf/NetperfPage.html
[4]In the case of a cluster of workstations, δ (r_i) is replaced by the resource load.

customized by the users (reflecting the application requirements) to give more importance to one over the other:

$$grank(r_i) = \alpha.rrank(r_i) + \beta.prank(r_i) \tag{5}$$

As a result, the resource with the highest grank will be relatively the most powerful and the most reliable resource, in respect to the user and application requirements. Hence, the performance-derived step allows identifying the reliable resources, which will be able to finish the job in the least time.

3. GRID superscalar: a middleware independent system allowing resource prediction

The previously described strategy, which tries to find the reliable and powerful resources, is integrated in the GRID superscalar system. This section first describes briefly GRID superscalar, then explains the implementation of reliability strategy in this system using Grid Application Toolkit (GAT) [1] and argues the choice of this toolkit. The integration of this solution in GRID superscalar will help users to take the right decision while choosing the appropriate resources before application execution starts.

3.1 GRID superscalar

The GRID superscalar [8], which could be considered as a workflow system, is a framework mainly composed of a programming interface, a deployment center and a run-time system. It runs actually on top of Globus [3] Toolkit, Ninf-G [9] and ssh/scp. GRID superscalar programming environment requires the following functions in the main program: GS_On() and GS_Off() functions are provided for initialization and finalization of the run-time. GS_Open(), GS_Close(), GS_FOpen() and GS_FClose() for handling files. GS_Barrier() function has been defined to allow the programmers to wait till all grid jobs finish. The user specifies the functions (jobs), which are desired to be executed in a remote server in the grid, via an IDL file. For each of these functions, the type and nature (input, output or input/output) of the parameters must be specified. The deployment center is a Java-based Graphical User Interface, which implements the grid resource management and application configuration. It handles early failure detection, transfers the source code to the remote machines, and generates some additional source code files required for the master and the worker parts (using the gsstubgen tool). It compiles the main program on the localhost, and the worker programs on the remote hosts, and finally generates the configuration files needed at run-time. The run-time library is able to detect job dependencies, builds a job graph, which enables to discover the inherent parallelism of the sequential application, and performs concurrent job submission. Techniques such as file renaming, file locality, disk sharing, checkpointing constraints specification are applied to increase the application performance.

3.2 Resource selection on GRID superscalar

As presented in section 2, the information such as the availability and the trustfulness of resources, the resource load and the amount of physical memory, the consistency of retrieved information, etc. is required for the proposed schema.

Some of this information like the amount of physical memory or the resource load is retrieved via the Mercury Monitoring System[5][6]. Another part of this information like the network latency and bandwidth may be estimated by some tools like NWS or netperf.

A large part of resource trustfulness is computed within GRID superscalar. GRID superscalar detects if any of the worker jobs fails due to an internal error, or because it has been killed for any reason. It is important to mention that the application and the user related failures(such as forgetting to run grid-proxy-init) are not considered in the trustfulness parameter. Besides, the resource trustfulness concerning the network failure may detect via ping program to verify the packets loss between resources.

All the information regarding the remote resources can be computed using any remote job submission system[6]. Once the information is computed, the local host collects them. Thereafter, part of this information is stored in a database[7], which allows to generate the historic trace of previously computed information. When all the information is retrieved, then the resources are sorted using the equation number 5.

Both the necessary information retrieval and the reliability strategy implementation are integrated in deployment center of GRID superscalar. This later permits to realize the proposed prediction scheme for reliable resources selection, by using application performance information (computation and data transfer capacity).

[5]The Mercury Monitoring System is a general-purpose grid monitoring system developed by the GridLab project. It supports the monitoring of machines, grid services and running applications. Mercury features a flexible, modular design that makes it easy to adapt Mercury for various monitoring requirements.

[6]Like job submission system in GAT, Globus or Unicore Toolkits

[7]For this purpose, the "Advert Management" service in Grid Application Toolkit offers us the possibility of storing and retrieving information, which is kept persistent throughout multiple and independent executions if GRID superscalar: this service allows each resource to maintain its own advertisements.

3.3 Middleware independent grid programming environment

In order to have our mechanism running in a larger number of grid environments, we use Grid Application Toolkit for the implementation of our prediction strategy. GAT provides a glue layer which maps the API function calls executed by an application to the corresponding grid middleware functionality. GAT was developed by the EC-funded GridLab project. It provides a simple and stable API to various grid environments (like Globus, Unicore [4] , ssh, GridLab services [2]).

Moreover, in order to have a both high-level and platform-independent grid programming environment which allows resource prediction, we implement GRID superscalar (runs actually on top of Globus Toolkit) on top of GAT.

GRID superscalar realization requires mainly the following GAT functionalities: file management, remote job submission and job state notification. File management deals with the access management of files on remote grid storage resources (like copying, moving and deleting file instances). Remote job submission permits starting and controlling jobs running on remote grid resources. Finally job state notification examines the state (initial, scheduled, running and stopped) of remote jobs. Implementing both the prediction mechanism and GRID superscalar's runtime system using GAT, allows sustaining a both high-level and platform-independent grid programming environment.

4. Experimentation

The objective of this section is to show some results of our implementation. We first present the utilized platform, and then the result of several experiments. The presented experimentations are done on the DAS2 testbed. DAS-2 is a wide-area distributed computer situated at five Dutch Universities in the Netherlands. It consists of 200 Dual Pentium-III nodes with 1 GB Random Access Memory. The Vrije Universiteit's cluster, containing 72 nodes, is the largest cluster, the other clusters consist of 32 nodes.

The following tables show the retrieved information for 4 cluster of DAS2. The first column in both tables contains the cluster names. The second column contains the availability of resources and the third one the trustfulness of them. The fourth and fifth columns show the total amount of physical and swap memory (in KB) in the system. Please note that a part of the capacity presented for physical memory is used by the operating system, the application, etc. Finally the last column shows the file transfer time (in sec) between the master and the respective workers. In table 2, the second column presents the job size running on the workers, and the third column shows the queueing waiting time (in sec). For the sake of clarity we illustrate just the waiting time for the small jobs.

Table 1. Retrieval information regarding the reliability and performance of three workers.

worker	availability	distrust	memory	swap	file transfer time
fs1.das2.liacs.nl	1.00	0.0	1540592	2096472	11.0
fs0.das2.cs.vu.nl	0.88	5.0	1540592	2096472	11.2
fs3.das2.ewi.tudelft.nl	0.61	6.0	1026584	2096472	11.1
fs2.das2.nikhef.nl	0.79	7.0	1540592	2096472	11.3

Table 2. Queuing waiting time for three workers.

worker	job size	queue wait time
fs1.das2.liacs.nl	small	12 sec
fs0.das2.cs.vu.nl	small	10 sec
fs3.das2.ewi.tudelft.nl	small	80 sec
fs2.das2.nikhef.nl	small	135 sec

The information presented in table 1 indicates that for those jobs which need less than 32 processors, fs1.das2.liacs.nl worker is the most appropriate and reliable resource. In the case where the jobs need more than 32 processors, obligatory fs0.das2.cs.vu.nl worker (the only cluster with 72 processors) will be chosen. Regarding the application requirements user can give more importance to the reliability or the execution time. In this case, by taking into account the information presented in table two, worker fs1.das2.liacs.nl may be chosen if the reliability parameter is more important. In the opposite case, when the execution time is more important fs0.das2.cs.vu.nl worker may be chosen. To realize the experimentations, we use three different kinds of applications: matrix multiplication, cholesky factorization of matrices and fastdnaml computation to estimate the phylogenetic trees of sequences.

In order to evaluate the effectiveness of proposed reliability strategy we use fastdnaml application which uses a large sequence as input (its execution takes quiet long time). In the first case the most reliable resources in DAS2 are chosen. Therefore we use fs0.das2.cs.vu.nl cluster situated in Amsterdam as master and fs1.das2.liacs.nl cluster located in Liden as worker (we use 4 processors of this cluster). The execution of this application is completed successfully and takes 5149,55 sec. In the second case instead of fs1.das2.liacs.nl we choose fs2.das2.nikhef.nl situated in Amsterdam university as worker (we use also 4 processors of this cluster). This cluster is less reliable and our application is failed after 4038,01 sec, due to sudden unavailability of this cluster. This means that we lost 78,41% of the time for completing this execution, by choosing a less reliable resource.

Table 3. GRID superscalar on top of Globus and GAT.

application	master	worker	using GAT	using Globus
matrix multiplication	fs0.das2.cs.vu.nl	fs1.das2.liacs.nl	252,12 sec	174,25 sec
cholesky factorization	fs0.das2.cs.vu.nl	fs1.das2.liacs.nl	575,30 sec	348,02 sec
fastdnaml	fs0.das2.cs.vu.nl	fs1.das2.liacs.nl	5149,55 sec	3947,25 sec

We developed a version of GRID superscalar based on GAT. In order to evaluate the performance of GRID superscalar on top of GAT and on top of Globus, we used the three types of application that we presented. The results are presented in table 3. We notice that GAT reduces the applications performance. The performance decreases around 30% in the case of matrix multiplication, 40% in the case of Cholesky factorization and 28% in the case of fastdnaml. This is due to the non functionality of pre staging and post staging of files in the used GRAM adaptors in one hand and the lack of clustering file copying in gridftp adaptors on the other hand.

Further work should perform similar experiments in a heterogeneous environment, where the selection of reliable and efficient resources is more critical.

5. Conclusion

Distributed environments, and in particular grids, are inherently unreliable. Frequent failures of their components and applications make development difficult, especially to the scientists who are not necessarily grid experts. This paper first presents a mechanism, which allows to run the application jobs on the most reliable and most powerful resources, in respect to the application requirements. Our described system gathers all the necessary characteristics about resources. Thanks to the collected information, our system finds and chooses the most suitable and adequate resource for each of the jobs. This paper also shows how to obtain a high-level and platform-independent, grid programming environment. It proposes to combine GRID superscalar runtime system with GAT, taking advantage of their respective properties. In this way GRID superscalar may run across various Grid middleware systems such as various versions of Globus, Unicore or ssh/scp.

References

[1] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, B. Ullmer. *The Grid Appli-*

cation Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. In the proceeding of the IEEE, vol. 93(3):534-550, 2005.

[2] G. Allen, D. Angulo, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzysky, J. Pukacki, M. Russell, T. Radke, E. Seidel, J. Shalf, I. Taylor. *GridLab: Enabling Applications on the Grid.* In proceeding of Grid Computing - GRID 2002 : Third International Workshop, 39-45, 2002.

[3] I. Foster and C. Kesselman. *Globus: A Metacomputing Infrastructure Toolkit*. In International Journal of Supercomputer Applications, vol. 11(2):115-128, 1997.

[4] V. Huber *UNICORE: A Grid Computing Environment for Distributed and Parallel Computing.* In Proceedings of 6th Internatioanl Conference on Parallel Computing Technologies (PaCT-2001), Springer, LNCS 2127, 258-266, 2001.

[5] S. Hwang and C. Kesselman. *Grid Workflow: A Flexible Failure Handling Framework for the Grid*. In the proceeding of 12th IEEE International Symposium on High Performance Distributed Computing, 126–137, 2003.

[6] G. Gombás, C. Attila Marosi and Z. Balaton. *Grid Application Monitoring and Debugging Using the Mercury Monitoring System*. In Advances in Grid Computing–EGC 2005, vol. 3470:193-199, 2005.

[7] J. Maassen, R. V. Van Nieuwpoort, T. Kielmann, K. Verstoep. *Middleware Adaptation with the Delphoi Service*. In Concurrency and Computation: Practice and Experience, vol. 18(13):1659-1679 , 2006.

[8] R. Sirvent, J. M. Pérez, R. M. Badia, J. Labarta. *Automatic Grid workflow based on imperative programming languages*. In Concurrency and Computation: Practice and Experience, vol. 18(10):1169-1186, 2006.

[9] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. *Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing*. In Journal of Grid Computing, vol. 1(1):41-51, 2003.

[10] R.Wolski, N. Spring, and J. Hayes. *The NetworkWeather Service: A Distributed Resource Performance Forecasting Service for Metacomputing.* In Journal of Future Generation Computing Systems,vol 15(5-6):757-768, 1999.

[11] J. Yu and R. Buyya. *Taxonomy of Scientific Workflow Systems for Grid Computing*. In Sigmod Record, vol. 34(3):44-49, 2005.

[12] S. Zhao and V. Lo. *Result Verification and Trust-based Scheduling in Open Peer-to-Peer Cycle Sharing Systems*. In IEEE Fifth International Conference on Peer-to-Peer Systems, 2005.