Parallel Processing Letters

IMPROVED RUNTIME AND TRANSFER TIME PREDICTION MECHANISMS IN A NETWORK ENABLED SERVERS MIDDLEWARE

EMMANUEL JEANNOT
INRIA-Lorraine, LORIA
*Villers les Nancy, 54600 Villers les Nancy France**

and

KEITH SEYMOUR
Department of Computer Science, University of Tennessee
Knoxville, TN 37996-3450, USA

and

ASYM YARKHAN
Department of Computer Science, University of Tennessee
Knoxville, TN 37996-3450, USA

and

JACK J. DONGARRA
Department of Computer Science, University of Tennessee
Knoxville, TN 37996-3450, USA

Received December 2006
Revised January 2006
Communicated by Guest Editors

ABSTRACT

In this paper we address the problem of accurately estimating the runtime and communication time of a client request in a Network Enabled Server (NES) middleware such as GridSolve. We use a template based model for the runtime estimation and a client-server communication test for the trannsfer time estimation. We implement these two mechanisms in GridSolve and test them on a real testbed. Experiments show that they allow for significant improvement in terms of client execution time on various scenarios.

Keywords: Grid computing, scheduling, performance prediction, parametric modeling

* INRIA-Lorraine, Equipe Algorille, Bât B
615, rue du Jardin Botanique
54600 Villers les Nancy, France
emmanuel.jeannot@loria.fr

1. Introduction

The adoption of Grid infrastructures as a platform for supercomputing holds great promise for accelerating scientific discovery using aggregations of distributed resources. However, the use of Grid infrastructures has, for the most part, been restricted to the largest and most resource intensive projects. For Grid computing to become a true success story, it must become an infrastructure that can be easily used by the *general* community of scientist and engineers. Within this community of practitioners, the use of scientific computing environments (SCEs) such as Matlab or Mathematica is pervasive. These domain specialists are accustomed to the flexible computing environment provided by an SCE, which gives them with the tools and libraries that they need to be productive and enables them to go from conceptualization to computation to visualization in an natural fashion.

The goal of grid middleware is to provide an environment that tries to virtualize access to resources by defining a programming model that tries to be as close as possible to existing ones (C, Matlab, etc.). In order to access the available distributed resources a grid middleware defines a software architecture and relies on a set of services. These services may be in charge of transaction security, service billing, data transfer, request scheduling, service location, fault tolerance, etc.

In this paper we focus on a particular type of middleware called a Network Enabled Server (NES). In a NES environment an application is composed of a set of requests that can be executed remotely on distant servers. Several middleware systems implement this model, including GridSolve [1], Ninf [2], and DIET [3,4]. Moreover, GridRPC [5] is an emerging standard of the APIs provided by such environments and is promoted by the Open Grid Forum (OGF).[a]

In the context of scientific applications the scheduling service provided by NES middleware plays a key role in terms of performance. Indeed, choosing the right set of resources to execute an application or a request is critical to obtain an execution time as close as possible to the optimal. However, for efficiently performing this choice, the scheduler has to rely on accurate information such as the execution time of the service and the communication time of the data that have to be transferred through the network.

Here we target the improvement of determining precisely the runtime and the communication time of a request. We propose two new features that allow for an accurate estimation of this information. We use a template based model for the runtime estimation and a client-server communication test for the transfer time estimation. We have implemented them in the GridSolve middleware and tested them on a real testbed. Experiments show that they allow for significant improvement of client execution time on various scenarios.

The remainder of this paper is organized as follows. In Section , we present the GridSolve middleware. The proposed improvements are described in Section and are evaluated in Section . A conclusion is given in Section

2. GridSolve

The purpose of GridSolve is to create the middleware necessary to provide a seamless bridge between the simple, standard programming interfaces and desktop systems that dominate the work of computational scientists and the rich supply of services supported

[a] `http://www.ogf.org`

by the emerging Grid architecture. The goal is that the users of desktop systems can easily access and reap the benefits (shared processing, storage, software, data resources, etc.) of using grids. This vision of the broad community of scientists, engineers, research professionals and students, working with the powerful and flexible tool set provided by their familiar desktop computing environment, and yet able to easily draw on the vast, shared resources of the Grid for unique or exceptional resource needs, or to collaborate intensively with colleagues in other organizations and locations, is the vision that GridSolve is designed to realize.

GridSolve is a client-agent-server system which provides remote access to hardware and software resources through a variety of client interfaces.

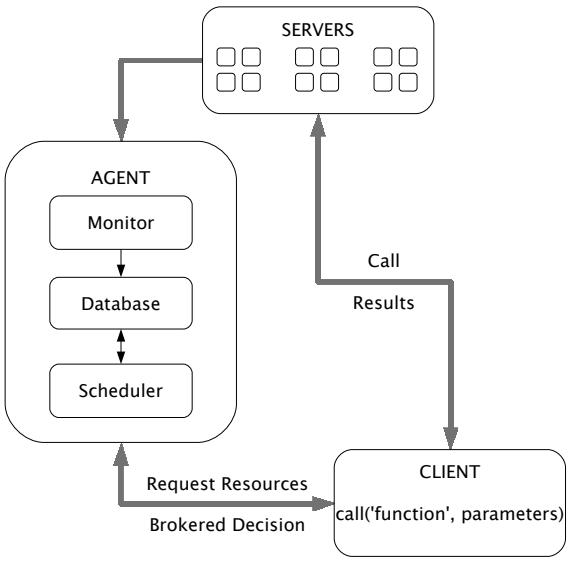

Fig. 1. Overview of GridSolve

A GridSolve system consists of three entities, as illustrated in Figure 1.

- The *Client*, which needs to execute some remote procedure call. In addition to C and Fortran programs, the GridSolve client may be an interactive problem solving environment such as Matlab.
- The *Server* executes functions on behalf of the clients. The server hardware can range in complexity from a uniprocessor to a MPP system and the functions executed by the server can be arbitrarily complex. Server administrators can straightforwardly add their own function services without affecting the rest of the GridSolve system.
- The *Agent* is the focal point of the GridSolve system. It maintains a list of all available servers and performs resource selection for client requests as well as ensuring load balancing of the servers.

In practice, from the user's perspective the mechanisms employed by GridSolve make the remote procedure call fairly transparent. However, behind the scenes, a typical call to GridSolve involves several steps, as follows:

(i) The client queries the agent for an appropriate server that can execute the desired function.
(ii) The agent returns a list of available servers, ranked in order of suitability.
(iii) The client attempts to contact a server from the list, starting with the first and moving down through the list. The client then sends the input data to the server.
(iv) Finally the server executes the function on behalf of the client and returns the results.

We have implemented a simple technique for adding arbitrary services to a running server. First, the new service should be built as a library or object file. Then the user writes a specification of the service parameters in a gsIDL (GridSolve Interface Definition Language) file. The GridSolve problem compiler processes the gsIDL and generates a wrapper which is automatically compiled and linked with the service library or object files. The services are compiled as external executables with interfaces to the server described in a standard format. The server re-examines its own configuration and installed services periodically to detect new services. In this way it becomes aware of the additional services without re-compilation or restarting of the server itself.

GridSolve is based on the GridRPC API, which represents ongoing work to standardize and implement a portable and simple remote procedure call (RPC) mechanism for Grid computing. This standardization effort is being pursued through the Open Grid Forum Research Group on Programming Models [6]. The initial work on GridRPC reported in [5] shows that client access to existing Grid computing systems such as GridSolve and Ninf [2] can be unified via a common API, a task that has proven to be problematic in the past. In its current form, the C API provided by GridRPC allows the source code of client programs to be compatible with different Grid services, provided that service implements a GridRPC API.

3. Improving the GridSolve Runtime and Transfer Time Prediction Mechanisms

To allocate a request to a server, the scheduling mechanism needs an accurate prediction of both the runtime of the service and the communication time of the data. Concerning the runtime estimation the standard GridSolve uses a simple prediction based on the complexity of the service. This estimation is very crude and has a lot of drawbacks as detailed later. Concerning the communication time estimation, in many middlewares (standard Gridsolve or DIET for instance), nothing is done. We detail our proposed solution here.

3.1. Modeling and Performance Prediction

3.1.1. The problem

To efficiently schedule an application requires being able to efficiently predict the duration of the requests that compose the application. However, predicting the duration of a request is a difficult task. Indeed, the duration might depend on the data (size and value) of the machine where the application is run and on the implementation of the service. Even when the duration of a service does not depend on the data values (as is the case with many linear algebra kernels), predicting this duration is hard. In GridSolve, the duration of the task is described in the gsIDL (GridSolve Interface Description Language) file, and is given by the constant of the higher degree of the complexity polynomial and is supposed to give an approximation of the number of operations the service has to perform when the inputs are known. The number of operations per second of the server is computed by running a sequential benchmark when the server is launched. The server periodically updates its current workload, which is used by the agent to scale down the server's speed. Then the estimated duration of the task is computed at run-time by dividing the estimated number of operations to be performed by the current speed of the server. However, computing the duration of a service based on the complexity polynomial has several drawbacks:

- First, it does not depend on the implementation. Indeed, different implementations of the same algorithm have the same complexity, but not necessarily the same speed. Assume for instance that the service is the matrix multiply routine of the BLAS (Basic Linear Algebra Subroutines). Nowadays there are a lot of different implementations of the same BLAS API, ranging from reference BLAS (a non-optimized Fortran version), to automatically tuned libraries such as ATLAS [7] and up to specific implementations optimized for a precise version of a certain CPU (*e.g.* the *goto BLAS*) [8]. The complexity of these implementations is always the same ($O(n^3)$ for multiplying matrices of order n), but the execution time might be completely different (for instance the the reference BLAS are about 6 times slower than the vendor optimized version on some CPUs). This effect is not taken into account by the standard version of GridSolve as the estimated service runtime is implementation independent.
- Moreover, obtaining the speed of the machine with a benchmark assumes that the flop-rate of each service is the same as the benchmark. In practice this is not true as compute-intensive services achieve higher flop rates than data-intensive services. In GridSolve, the flop rate is estimated by running a Linpack benchmark which is close to the peak flop rate of the processor. This is good when the service is a compute-intensive one such as for a linear algebra kernel. However, if the service is I/O bound (such as database access) or memory constrained (such as an out-of-core computation), the estimated runtime is likely to be a huge underestimation of the actual runtime.
- Finally, for a given service a slight change of a parameter may lead to a different algorithm and a different time to execute the service. For instance the `dgemm` routine

of the BLAS performs $C \leftarrow \alpha \times A \cdot B + \beta \times c$, where A, B and C are matrices. It is easy to see that the case $\alpha = 1$, $\beta = 0$ is completly different from the case $\alpha = 0$ and $\beta = 1$. However, since in GridSolve the values of α and β are not related to the volume of data, they do not appear in the complexity formula describing the `dgemm` service.

3.1.2. Proposed solution: automatic template modeling

To solve the problem described above, we propose using a template model for each service that is instantiated on each server for each different use case of the service. This template model is composed of two parts:

- A polynomial of the *parameters* of the problem. It can be any polynomial that according to the service designer fits all the behaviors of the service. For instance if we want to model the general `dgemm` BLAS routine, we have three parameters m, n and k that play a role in the model. Indeed, we are multiplying a matrix of size m by k with a matrix of size k by n to obtain a matrix of size m by n. A general polynomial in this case is : $a_1m + a_2n + a_3k + a_4mn + a_5mk + a_6nk + a_7mnk$, where the a_i are the *coefficients* of model and will be computed by GridSolve as explained later.
- A set of *categories*. For each service, different use cases can lead to different execution times. We need to differentiate these cases using the values of some parameters transmitted to the service. For the `dgemm` case we have 5 such parameters: `transa`, which tells if matrix A is transposed or not, `transb` which tells if matrix B is transposed or not, and α and β as explained above. Different values of `transa` and `transb` lead to different performance and different values of α and β lead to different cases and algorithms. More precisely we have to distinguish 2 cases for `transa` and `transb` (*e.g.* "T" and "N" and 3 cases for α and β (0, 1 and all the "other" values). The cartesian product of all these cases leads to a set of 39 different *categories* which have to be modeled differently.

The designer of a service has to define the two above parts. In order to do that, for each parameter of the service they must define the possible values that play a role in defining some categories. The value can be a string, a number, or the special string "other" for all the other possible cases. The service designer must also write down the polynomial that defines the template model. When compiling the service, GridSolve interprets this information and builds as many models as the number of categories. It also generates a code that switches to the right sub-model when given the values of the parameters.

Now let us see how the coefficients (the a_i) are computed. We have designed a parametric regression system that computes or updates the coefficients at runtime. For each category, it works as follows:

(i) At the beginning all the a_i are initialized to -1
(ii) Each time, the agent asks for a runtime estimation by sending all the parameters, the system switches to the model corresponding to the category and evaluates the model.

(iii) Each time the server runs the service it updates the coefficients of the model using this run and the previous ones. The update of the coefficients is done using a least squares regression on the coefficients.

(iv) Each time the model is updated, information about a given number of previous runs (100 by default) is stored on the local disk. Hence if the server is stopped and re-launched, the system can use this information to compute the coefficients of each category.

A few remarks need to be made at this point.

- For the very first run, when all the a_i equal -1, the model is not able to give an estimate of the duration. In this case, we use the old method (based on the complexity polynomial) to compute the runtime estimate.
- As the number of runs increases, the available information increases, and the runtime estimates become more precise. An evaluation of the accuracy of this method will be given in the experimental section.
- Since all the information is interpreted and translated into C before compiling the service, the runtime estimation is very fast (within the same order of magnitude of the old method).

Finally, it is easy to see that this improvement is a solution to our problems:

- The model is implementation-dependent. The coefficients (the a_i) are computed on-line based on the actual service runtime. Different implementations will lead to different runtime and hence to different instance of the model.
- For the same reason as above, it does not assume that the flop-rate is the same for each service.
- Furthermore, thanks to the categorization, it is able to deal with a service achieving different flop-rates and/or different use cases. For instance the `dgemm` service is able to scale a matrix (case $\beta \neq 0$ and $\alpha = 0$) or to multiply two matrices (case $\alpha \neq 0$). These two cases are modeled differently and the runtime estimation is therefore made independently.

Moreover, in the case of over-subscribed set of resources the timing prediction is still correct because we use the CPU-time to measure the runtime of a service and we divide the predicted execution time by the load of the server. The only remaining problem is to accurately compute the finish time of a service when the load is not constant. In this case scheduling technics based on historical trace manager as in [9] can be used.

3.2. Measuring Communication Cost

3.2.1. The problem

The performance prediction model described above gives us an accurate estimate of the execution time of an instance of a GridSolve service, but execution time is only part of the

overall time required to perform the remote procedure call. The other major component of the total RPC time is the communication cost of sending and receiving the data. In many cases, the communication cost actually dominates the computation cost. Let us take matrix multiplication as a simple example. On a 3.4 GHz Pentium 4, the ATLAS implementation of `dgemm` can multiply 1000x1000 matrices in 0.52 seconds. The GridSolve `dgemm` service requires sending three 1000x1000 matrices (A, B, and C) and receiving one 1000x1000 matrix (C), which in double-precision results in a total data transfer of 32MB (24MB from the client to the server and 8MB from the server back to the client). On an unloaded 100Mbit Ethernet LAN, we observed transfer rates of 11MB/sec, which means that the total data transfer time for our DGEMM example would be roughly 2.9 seconds, or more than 5 times the computation cost. Using a WAN results in an even more extreme ratio.

This exposes a weakness in a scheduling strategy based purely on computation cost. Choosing the fastest server may minimize the execution time, but if that server is on a distant network, the communication cost can easily overshadow the savings in execution time.

3.2.2. Proposed solution: client-to-server communication cost estimation

To eliminate this weakness, we need an estimate of the network performance between the client and the servers that could possibly execute the service. This can be difficult to know ahead of time given the dynamic nature of the system, so we gather the information empirically at the time the call is made. When the client gets a list of servers from the agent, it is sorted based only on the estimate of the computational cost. Normally the client would simply submit the service request to the first server on the list, but instead we first measure the bandwidth from the client to the top few servers using a simple 32KB ping-pong benchmark. Then we determine the total size of the data to be sent and received. Given the data size and the network speed, we compute an estimate of the total communication and computation RPC time for the servers and re-sort the list.

There is some cost associated with performing these measurements, but the idea is that the reduction in total RPC time will compensate for the overhead. Despite that expectation, we try to keep the measurement overhead to a minimum. The time required to do the measurement will depend on the number of servers which have the requested problem and the bandwidth and latency from the client to those servers. When the data size is relatively small, the measurements are not performed because it would take less time to just send the data than it would take to do the measurements. Also, since a given service may be available on many servers, the cost of measuring the network speed to all of them could be prohibitive. Therefore, the number of servers to be measured is limited to those with the highest computational performance. The exact number of measurements is configurable by the client. Once the measurements have been made, they can be cached for a certain amount of time so that subsequent calls on that client do not have to repeat the same measurement. The lifetime of the cached measurements is configurable by the user.

There are many other projects that monitor grid performance, see [10] or [11] for a review. For example, the Network Weather Service (NWS) [12], is a popular general system

service that can monitor the performance of network bandwidth and latency (as well as other measures) and provide a statistical forecast for future performance. However, for the GridSolve system, most of the existing systems are inappropriate because clients enter and leave GridSolve dynamically, making it difficult to measure and retain the communication costs between the clients and the full set of servers. Moreover, NWS is required to be configured on each end, which necessitate some expertise that we do not assume (at least on the client side). Hence, GridSolve has chosen to implement probes as a way of building up the communication cost matrix between a client and the servers relevant to that client.

4. Experimental Results

We have experimented with the proposed features on a real testbed. For testing what is happening on a WAN, we have used some machines of the GRID5000[b]infrastructure in France [13] and some machines at the University of Tennessee. For scheduling the requests, we used the standard Minimum Completion Time algorithm (MCT [14]). Finally, in all the following, we assume that there is no external load (servers are dedicated) and that request runtimes are deterministic.

4.1. Template Modeling Results

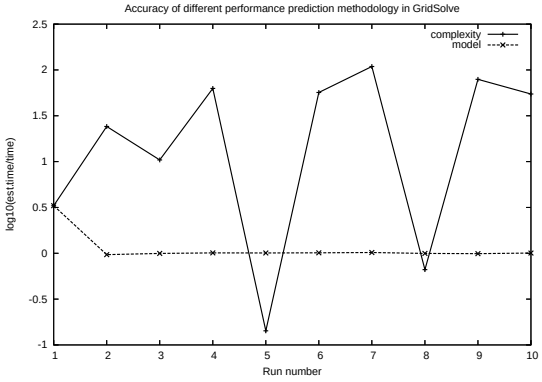

Fig. 2. Accuracy of the Automatic Modeling (1 server)

First, we have evaluated the precision of the modeling and the speed of convergence of the model. In order to do that, we have launched the `dgemm` service on one server and call sequentially this service 10 times. Results are shown Figure 2. On the x-axis we plot the run number and on the y-axis we plot the log of the ratio of estimated runtime divided

[b] `http://www.grid5000.fr`

by the measured runtime. Log shows the errors in order of magnitude. When it is zero it means the prediction is perfect. When it is positive (resp. negative) it means that the model overestimated (resp. underestimated) the actual runtime. We plot two curves: one without the template modeling feature (curve *complexity* as the prediction is made with the basic complexity system) and one with the template modeling feature (curve *model*). We see that for the first run (when the model is not instantiated) the predictions are the same. This is due to the fact that for the first run the system switches back to the basic complexity evaluation leading to the same estimation. From the second run we see that the template modeling feature is very accurate (within 5%) while the basic system is still sometimes 2 orders of magnitude wrong.

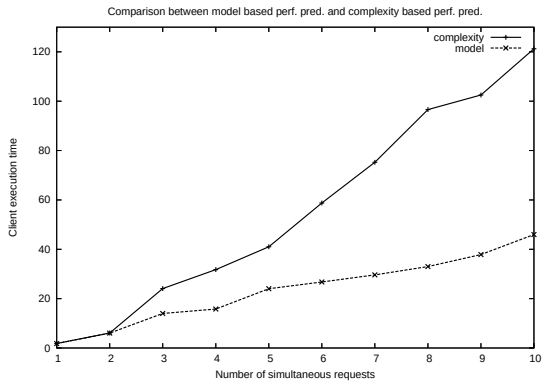

Fig. 3. Improved Client Request Execution Time. One server with a slow implementation of BLAS and one with a fast implementation

Second, we have evaluated how the template modeling feature can help to improve the client execution time. In order to do that we have built a client that submits a given number of simultaneous (non-blocking) `dgemm` requests. The size of the matrix is chosen randomly. We have set up an environment with two servers. One server is executing an optimized implementation of the BLAS. The other server is executing the reference BLAS, which is a slow implementation of the same library. In figure 3, we plot the runtime of the client when we increase the number of simultaneous requests. As before, we have two cases: one with the template modeling feature enabled and one with only the basic complexity modeling. We see that with the template modeling mechanism, the client runtime is up to three times faster than with only the basic modeling. This is explained as follows. On these servers, the reference BLAS are 6 times slower than the optimized BLAS. Moreover the basic modeling does not distinguish between the two implementations while the template modeling makes a clear distinction in terms of performance of the two implementations.

Therefore, in the case of the basic modeling, the scheduler allocates requests in a round robin fashion, while when the template modeling is active the scheduler favors the faster implementation. In the worst case, half of the requests last 6 times longer in the first case than for the second case, leading to a factor of 3 runtime increase.

4.2. Communication Measurement Results

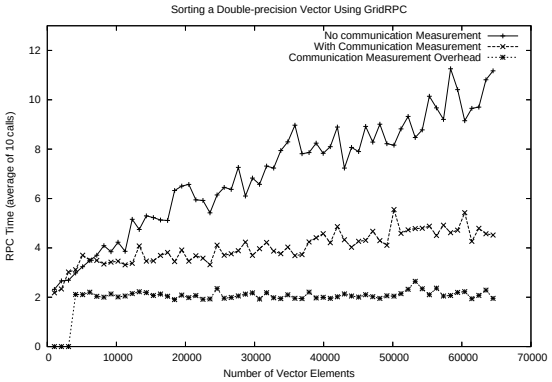

Fig. 4. Effect of Communication Measurement on RPC Time

To measure the effectiveness of rescheduling using the network measurement information, we use a client at the University of Tennessee and a grid of 11 servers. One of the servers is a relatively slow Pentium 3 machine located on the UT campus, while the other ten servers are much faster AMD machines located in France as part of GRID5000. Using array sizes from 1k to 64k elements, we make ten calls to a quicksort service and measure the average time for the calls to complete.

One experiment performs network measurements to re-sort the server list and the other experiment uses the original server list as provided by the agent. The original GridSolve scheduler chooses the servers located in France since their computational power is much greater, but in this case the actual RPC time is shorter when the local server is chosen because of the faster network connection. Figure 4 shows that using the network measurement information leads to a reduction in the average RPC time compared to the original scheduling.

The measurement overhead is also shown in Figure 4. For array sizes under 4k elements, there was no overhead because the measurement threshold had not been reached, therefore no measurements were performed. Above 4k elements, the measurement overhead is roughly 2 seconds, but since the server selection is better, the overall time does

not exhibit a corresponding increase. Moreover, on a LAN, the measured overhead is very small and almost unnoticeable.

5. Conclusion

In this paper we have presented GridSolve a grid middleware that allows for remote execution of services. In order to efficiently execute client requests the GridSolve scheduler has to rely on important information: the computation execution time and the communication time. Here, we have proposed two features that improve the estimation of this information.

The contribution of this paper is twofold. First we have introduced a template based modeling mechanism that is able to accurately predict the runtime of the service based on previous execution. It improves the standard runtime estimation of GridSolve as it is more accurate and takes into account the specificity of the service and the machine it runs on. Second, we have developed an estimator of the communication cost between the client and the server. Since communication cost is often very large such an estimator enables to discard fast remote server if the gain in terms of computation time is overshadowed by the communication time.

These two features have been implemented in GridSolve and tested on a real testbed. Experiments show that they allow for significant improvement in terms of client execution time on various scenarii. Hence, they are now implemented in the distributed version of GridSolve available at `http://icl.cs.utk.edu/gridsolve`. However, these ideas are general and not tied to GridSolve. They could easily be implemented in any grid middleware that needs that kind of information, such as other GridRPC implementations (DIET, Ninf, etc.).

Future work is directed towards the design of enhanced scheduling strategies that would efficiently take into account this accurate information and try to optimize criteria other than the response time, such as the fairness, the servers throughput, or the quality of service.

References

[1] A. YarKhan, K. Seymour, K. Sagi, Z. Shi, and J. Dongarra. Recent Developments in GridSolve. *International Journal of High Performance Computing Applications*, 20(1), spring 2006.

[2] Hidemoto Nakada, Mitsuhisa Sato, and Satoshi Sekiguchi. Design and Implementations of Ninf: Towards a Global Computing Infrastructure. In *Future Generation Computing Systems, Metacomputing Issue*, volume 15, pages 649–658, 1999.

[3] DIET (Distributed Interactive Engineering Toolbox). `http://graal.ens-lyon.fr/DIET`.

[4] Philippe Combes, Frédéric Lombard, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In A. Jean-Marie, editor, *Advances in Computing Science - ASIAN 2002. Internet Computing and Modeling, Grid Computing, Peer-to-Peer Computing, and Cluster Computing. Seventh Asian Computing Science Conference*, volume 2550 of *Lecture Notes in Computer Science*, pages 110–124, Hano?, Vietnam, December 2002. Springer-Verlag.

[5] K. Seymour, N. Hakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In M. Parashar,

editor, ***GRID 2002***, pages 274–278, 2002.

[6] Global Grid Forum Research Group on Programming Models. `http://www.gridforum.org/7_APM/APS.htm`.

[7] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In ACM, editor, ***SC'98: High Performance Networking and Computing: Proceedings of the 1998 ACM/IEEE SC98 Conference: Orange County Convention Center, Orlando, Florida, USA, November 7–13, 1998***, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1998. ACM Press and IEEE Computer Society Press. Best Paper Award for Systems.

[8] Kazushige Goto and Robert van de Geijn. High-performance implementation of the level-3 BLAS. Technical Report CS-TR-06-23, The University of Texas at Austin, Department of Computer Sciences, May 5 2006. Thu, 4 Jan 107 17:59:12 GMT.

[9] Y. Caniou and E. Jeannot. Multi-Criteria Scheduling Heuristics for GridRPC Systems. ***International Journal of High Performance Computing Applications***, 20(1):61–76, spring 2006.

[10] Dong Lu, Yi Qiao, Peter A. Dinda, and Fabián E. Bustamante. Characterizing and Predicting TCP Throughput on the Wide Area Network. In ***25th International Conference on Distributed Computing Systems (ICDCS 2005), 6-10 June 2005, Columbus, OH, USA***, pages 414–424, 2005.

[11] Serafeim Zanikolas and Rizos Sakellariou. A taxonomy of grid monitoring systems. ***Future Generation Computer Systems***, 21(1):163–188, January 2005.

[12] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. ***Future Generation Computer Systems***, 15(5–6):757–768, 1999.

[13] Raphël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. ***International Journal of High Performance Computing Applications***, 20(4):481–494, November 2006.

[14] M. Maheswaran, S. Ali, H.J. Siegel, D. Hengsen, and R.F. Freund. Dynamic Matching and Scheduling of a class of Independent Tasks onto Heterogeneous Computing System. In ***Proceedings of the 8th Heterogeneous Computing Workshop (HCW '99)***, April 1999.