INTERACTIVE GRID-ACCESS USING GRIDSOLVE AND GIGGLE

M. Hardt,[*2] K. Seymour,[*1] J.Dongarra,[1] M. Zapf,[3] N.V. Ruiter[3]

[1]*Innovative Computing Laboratory, University of Tennessee, Knoxville, USA*
[2]*Steinbuch Centre for Computing, Forschungszentrum Karlsruhe, Germany*
[3]*Institute for Data Processing and Electronics, Forschungszentrum Karlsruhe, Germany*
e-mail: hardt@iwr.fzk.de, seymour@cs.utk.edu

Abstract.

General purpose Problem Solving Environments (PSEs) like Matlab are widely used in the fields of science for development of new algorithms. If a lot of computing power is required to run these algorithms, todays PSEs lack support for accessing the distributed infrastructures of the organisation (i.e. grids), which limits the size of the problems that can be solved. This contribution shows a new approach to utilize the grid from within PSEs without major adjustments by the user. The primary tools are GridSolve and and the grid-middleware gLite. The applicability is illustrated by an exemplary algorithm (Mandelbrot calculations).

Keywords: GridSolve, gLite, computing, gridRPC, grid computing, problem solving environment, Matldab, giggle

1 INTRODUCTION

The Interactive European Grid Project (int.eu.grid) provides a distributed computing infrastructure based on gLite. Presently gLite does not provide support for interactive applications nor for graphical (GUI) applications. Within int.eu.grid, we are investigating different methods for adding this functionality.

This article gives an overview about gLite and the additional middleware GridSolve. GridSolve has not been used in combination with gLite. We describe in detail how this can be done and also show performance measurements of the solution that we implemented.

1.1 gLite

We use the gLite installation with its interactive extensions as well as the Message Passing Interface (MPI) as provided by the Interactive European Grid [5]. This infrastructure is described in detail in "A Grid Infrastructure for Parallel and Interactive Applications", also published in this journal.

The gLite [2] middleware provides an interface to allocate heterogeneous compute

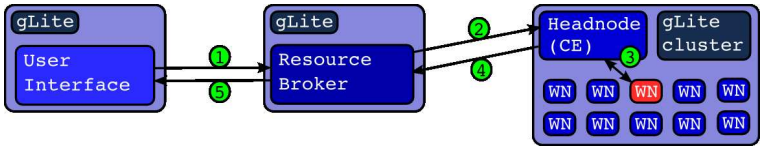

Fig. 1. The diagram shows the path of a batch job using the gLite infrastructure. For execution on the WorkerNode (WN) data and application have to be transferred from the user interface via the ResourceBroker (RB) (1) to the CE (2) and back (3) (4) (5).

resources on a "per job" basis. Within this job paradigm, one job consists of a description of job requirements:

- dependent input data that has to be already on the grid
- operating system (OS) requirements
- memory or CPU requirements

and a self-contained piece of software that will be executed inside a remote batch queue. For job submission gLite provides a specialised computer enclosing the User-Interface (UI). This computer contains client software for submitting jobs as well as the software that is going to be submitted to the grid. When submitting a job, the UI forwards it to the ResourceBroker (RB) ((1) in Fig. 1). The RB has access to several sources of monitoring to find (2) the resource with the best match to the user's job-description. This resource is typically the headnode of a cluster or Compute Element (CE) in gLite terminology. The CE in turn forwards (3) the job to one of the cluster or Worker Nodes (WN) where the job will be executed. If output was created on the WN it can either be stored and registered in the grid for later use or it has to be transported back along the path (3) (4) (5).

1.2 Limitations of gLite

The infrastructure described above is known to scale from large to very large numbers of resources (see Fig. 2). Being designed to meet these scaling requirements, the useability was not the primary focus. This becomes apparent when trying to use the infrastructure with applications for which gLite was not intentionally designed. One example for such applications could be a scientific problem solving environment (PSE) e.g. Matlab or a spreadsheet calculation program (e.g. Microsoft Excel). The differences between this style of application and the one that gLite was designed for are manyfold:

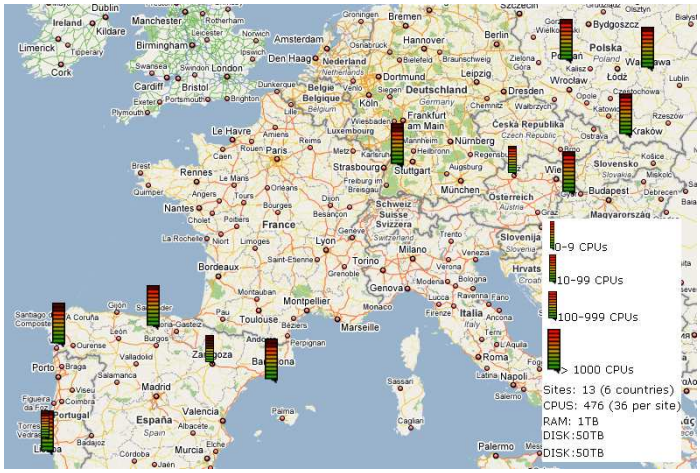

Fig. 2. European part of the LCG grid. The figures below date from October 2007 and comprise the worldwide installation

A typical software consists of several parts: A graphical user interface (GUI) and a backend that processes computations based on the user's commands. These computations can be more or less complex. The idea of grid-computing is that the software can utilize remote resources in the grid that are available in large quantities. The application programmer ensures that the application is capable of using the resources in parallel.

The job paradigm, however, requires a certain structure of an application. This is because firstly the startup of a job in gLite has an overhead in the range of $10\,s$. Secondly a job is an application, i.e. a self contained part of a larger application has to be sent to the grid and bootstraped on the assigned WorkerNode in order to process and return information. While the design of gLite allows the job to access data remotely on behalf of the user further communication between the user and the job is not supported. This is very different from the kind of parallel processing usually used in multithreading or cluster computing environments. Quintessentially, this lack of support for direct communication makes interactive useage of the grid very difficult and API-like calls of complex tasks virtually impossible.

Another issue is related to the job paradigm and the fact that we utilize WNs in many different organisational domains and countries. This leads to the problem that software dependencies might not be fulfilled at all assigned WN.

Finally software licenses might not match the gLite model. While some applications are licensed on a per-user base, that one user can run on as many resources as he likes, other applications require one running license per entity. Many scientific problem solving environments (e.g. Matlab) require one license per computer.

This contribution shall describe how it is possible to overcome these limitations. In the following sections we describe the additional middleware (GridSolve) on top of gLite

(see Section 3) as well as the integration of GridSolve and the underlying gLite (4). Finally we prove that the described solution works and give a performance as evaluated wi th our prototype in Section 5.

2 STATE OF THE ART

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 [4]. The initial work on GridRPC reported in [8] shows that client access to existing Grid computing systems such as GridSolve and Ninf [7] 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.

As of September 2007, the GridRPC API is an OGF standard and has been implemented by several Grid Middleware systems. Ninf [7] is a project from the National Institute of Advanced Industrial Science and Technology in Japan. The current version, called Ninf-G, is a GridRPC-compliant programming middleware system using the Globus Toolkit as the underlying mechanism. Another ongoing project with a compliant implementation of the GridRPC API is the Distributed Interactive Engineering Toolbox (DIET) [1] from ENS Lyon, *et al*. DIET is similar, but is based on a hierarchical view of the system and uses CORBA as its underlying mechanism.

To help to verify the compliance of these implementations, the GridRPC working group has developed software that can test various aspects of their behavior. The results of this interoperability testing were published in an OGF Informational Document [9] in early 2007. This report confirmed that Ninf-G, GridSolve, and DIET all conformed to the GridRPC specification.

Scientific computations will increasingly rely on very large data sets, so future work on GridRPC has to emphasize data management. Grid middleware should be prepared to deal with data sets that are too large to reside on the client's machine or as in the case of workflow systems, impractical to transfer back and forth between client and server. There is currently a proposal in the GridRPC working group for a Data Handle API that will help to address this issue.

3 GRIDSOLVE

3.1 Introduction

The purpose of GridSolve is to create the middleware necessary to provide a seamless bridge between computational scientists using desktop systems and the rich supply of services supported by the emerging Grid architecture. The goal is that the users of desktop systems can easily access and reap the benefits (in terms of shared processing, storage,

software, data resources, etc.) of using grids. GridSolve is designed to enable a broad community of scientists, engineers, research professionals and students to easily draw on the vast, shared resources of the Grid. Working with the powerful and flexible tool set provided by their familiar desktop computing environment, they can tap into the power of the Grid for unique or exceptional resource needs. In addition to harnessing computational power, the Grid infrastructure can enable new forms of collaboration with colleagues in other organizations and locations.

3.2 How GridSolve Works

GridSolve is a client-agent-server (or *brokered RPC*) system which provides remote access to hardware and software resources through a variety of client interfaces.

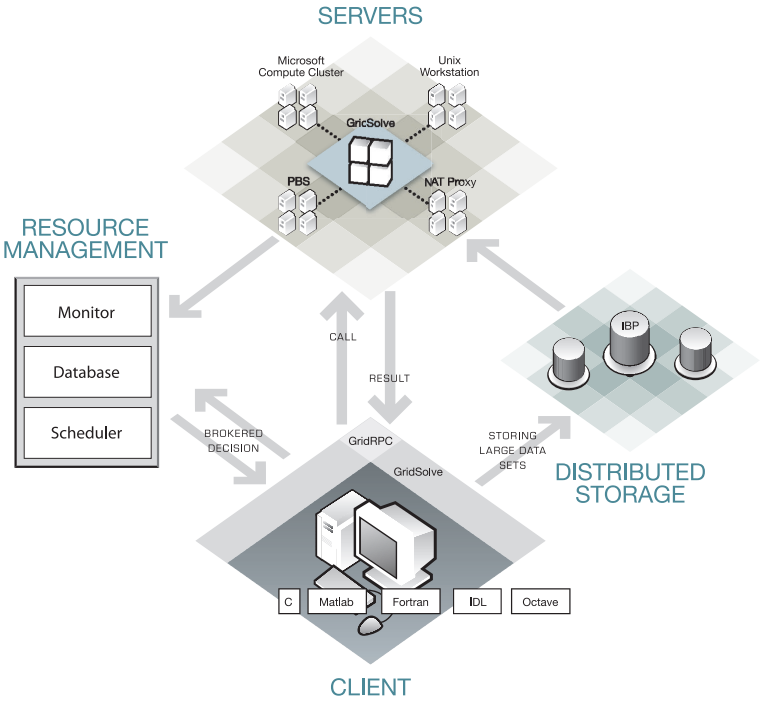

Fig. 3. Overview of GridSolve

The system consists of three entities, as illustrated in Figure 3.

- 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, Octave, or IDL (Interactive Data Language).
- 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 add straightforwardly 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:

1. The client asks the agent for an appropriate server that can execute the desired function.
2. The agent returns a list of available servers, ranked in order of suitability.
3. 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.
4. Finally the server executes the function on behalf of the client and returns the results.

In addition to providing the middleware necessary to perform the brokered remote procedure call, GridSolve aims to provide mechanisms to interface with other existing Grid services. This can be done by having a client that knows how to communicate with various Grid services or by having servers that act as proxies to those Grid services. GridSolve provides some support for the proxy server approach, while the client-side approach would be supported by the emerging GridRPC standard API [8].

3.3 Integrating User Services

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. Thus 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.

Normally the GridSolve server executes the actual service request itself, but in some cases it can act as a proxy to other services such as Condor. The primary benefit is that the client-server communication protocol is identical so that the client does not need to be aware of every possible back-end service. A server proxy also allows aggregation and scheduling of resources on one GridSolve server such as the machines in a cluster.

3.4 Scheduling

The selection of the best server for a particular job is carried out at several layers. When a new service is added, the author should provide a rough characterization of the perfor-

mance in terms of the arguments to the function. For example, sorting an N element array may be characterized with `COMPLEXITY="N * log(N)"` in the service configuration file. As the service is invoked, the server keeps track of the typical execution time for various problem sizes and uses a least squares regression to compute coefficients for an expression that more closely characterizes the expected performance. This is useful in cases where different implementations of a service have the same theoretical execution time, but very different real-world performance (e.g. vendor-tuned BLAS (Basic Linear Algebra Subprograms)compared with the reference BLAS). Both the theoretical and observed information are sent to the GridSolve agent, which uses them to determine the ranking of the servers. After the ranked list is returned to the client, it may choose to refine the list based on communication performance. For instance, a very fast server may not be the best choice if it is only reachable through a slow connection. Thus, the client can run a quick series of communication tests to estimate the time that it would take to send and receive the data from each of the servers. The server list is then re-sorted based on this information.

3.5 Network Address Translators

As the rapid growth of the Internet began depleting the supply of IP addresses, it became evident that some immediate action would be required to avoid complete IP address depletion. The IP Network Address Translator [3] is a short-term solution to this problem. Network Address Translation presents the same external IP address for all machines within a private subnet, allowing reuse of the same IP addresses on different subnets, thus reducing the overall need for unique IP addresses.

3.5.1 Complications in the Presence of NATs

As beneficial as NATs may be in alleviating the demand for IP addresses, they pose many significant problems to developers of distributed applications such as GridSolve [6]. Some of the problems as they pertain to GridSolve are:

- *IP addresses are not unique* – In the presence of a NAT, a given IP address may not be globally unique. Typically the addresses used behind the NAT are from one of several blocks of IP addresses reserved for use in private networks, though this is not strictly required. Consequently any system that assumes that an IP address can serve as the unique identifier for a component will encounter problems when used in conjunction with a NAT.
- *IP address-to-host bindings may not be stable* – This has similar consequences to the first issue in that GridSolve can no longer assume that a given IP address corresponds uniquely to a certain component. This is because, among other reasons, the NAT may change the mappings.
- *Hosts behind the NAT may not be contactable from outside* – This currently prevents all GridSolve components from existing behind a NAT because they must all be capable of accepting incoming connections.

- *NATs may increase connection failures* – Connections through NATs may be dropped depending on the particular implementation (especially after a period of inactivity). This implies that GridSolve needs more sophisticated fault tolerance mechanisms to cope with the increased frequency of failures in a NAT environment.

To address these issues we have developed a new communication framework for GridSolve. To avoid problems related to potential duplication of IP addresses, the GridSolve components will be identified by a globally unique identifier specified by the user or generated randomly. The mapping between the component identifier and a real host will not be maintained by the GridSolve components themselves. There will be a discovery protocol to locate the actual machine running the GridSolve component with the given identifier. In this sense, the identifiers of the component is a network address that is layered on top of the real network address. Thus, a component identifier is sufficient to uniquely identify and locate any GridSolve component, even if the real network addresses are not unique. This is somewhat similar to a machine having an IP address layered on top of its MAC address. Since NATs may introduce more frequent connection failures, we have implemented a mechanism that allows a client to submit a problem, break the connection, and reconnect later at a more convenient time to retrieve the results. We may also want to enhance the protocol to allow restarting partial transfers.

An important detail for this work on a new communication model is the *proxy*, which is a component that allows servers to exist behind a NAT. Since a server cannot accept unsolicited connections from outside the private network, it must first register with a proxy. The proxy acts on behalf of the component behind the NAT by establishing connections with other components or by accepting incoming connections. The component behind the NAT keeps the connection with the proxy open as long as possible since it can only be contacted by other components while it has a control connection established with the proxy. To maintain good performance, the proxy only examines the header of the connections that it forwards. It uses a simple table-based lookup to determine the destination address of each connection. Furthermore, to prevent the proxy from being abused, authentication may be required.

The programming interface that applications use to communicate through the proxy is based on the BSD sockets API. To simplify for developers the modification of their code to be NAT-tolerant, our API mirrors the socket API as closely as possible.

4 INTEGRATION OF GRIDSOLVE AND GLITE

On the one hand we are motivated to use gLite because of the large amount of resources provided, but on the other hand we want to use GridSolve because it provides easy access to remote resources. The integration of both platforms promises an API-like access to resources that are made available by gLite.

One challenge is that the developments of the grid-infrastructure middleware gLite and the GridRPC middleware GridSolve have progressed simultaneously without mutual consideration. Therefore neither of the systems were designed to work well with the other one. In order to integrate both software packages, certain integrative tasks have to

be performed.

To keep these tasks as reproducible and useful as possible, we have developed a set of tools and created a toolbox that we named *giggle* (**G**enuine **I**ntegration of **G**ridSolve and **gL**it**e**).

4.1 Giggle design

We want to use the gLite resources within the GridSolve framework, hence we have to start GridSolve servers on a number of gLite WorkerNodes. This is done by submitting gLite jobs that are in fact so called "pilot jobs", i.e. the gLite job is used only to start up a daemon which provides the allocated resource for the clients that connect to it. In our case this daemon is the GridSolve *server*. Pilot jobs provide two advantages:

- Fault tolerance: Only those pilot jobs that start the GridSolve daemon successfully will be available for clients to connect. Thus we only have resources that are proven to work. The downside is that it is difficult to control the precise number of active resources.
- The provided resources can be used independently from the gLite middleware. This gives us the intended interactive control of the resources.

Giggle simplifies the pilot job mechanism for the user by providing default jobs that are sent to the gLite infrastructure whenever the user requests more resources. Using the interactive grid extensions, it is possible to allocate several computers that are either scattered across the grid or confined to one cluster or a combination of both.

Every single pilot will be started on a WorkerNode (WN) by gLite. It is impossible to know which software is installed at that particular WN. This is why giggle downloads and installs a pre-built binary package of GridSolve together with the most commonly used libraries. Currently these packages are downloaded from a webserver.

For enhanced speed of startup time and network throughput, caching is supported. Furthermore, shared filesystem clusters benefit from gained installation speed, because in that case only one shared installation per cluster is carried out. After the installation, the GridSolve *server* is started. It connects to the *agent* and is thereafter available for the user. In order to accomplish this, we have defined infrastructure servers, that carry out specific tasks:

- The user has access to the *developers workstation*. This is typically his desktop or laptop computer. Giggle and the GridSolve client (*gs_client*) have to be installed.
- A *webserver* is used to store all software components which have to be installed on the WorkerNodes by giggle. We cannot ship these components via gLite mechanisms because then all packages are transferred via two additional computers, as can be seen in Fig. 1.
- The *servicehost* runs the GridSolve agent (`gs_agent`) and optionally, the proxy (`gs_proxy`). A separate host is currently required, because of specific firewall requirements which can not be easily integrated with the security requirements of a

webserver. Technically the servicehost can run on any publically accessible computer, e.g. the above mentioned webserver.

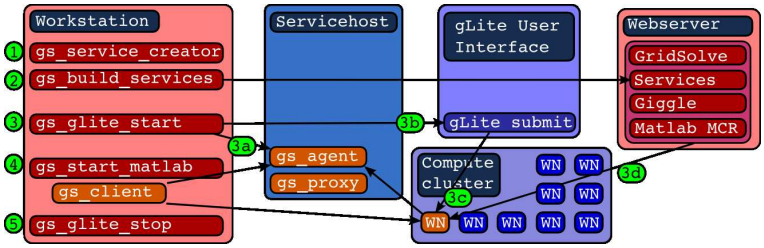

Fig. 4. Architectural building blocks of giggle and their interaction.

Fig. 4 shows these three additional computers and their interaction with the gLite hosts.

4.2 Giggle tools for creating services

Creating services (i.e. making functions available remotely) via the GridSolve mechanism requires in principle only a few steps: definition of the interface and a recompilation using the GridSolve provided tool *problem_compile* (see Section 3).

However, while developing our own services for GridSolve on gLite, we found that it is better to include our services into the GridSolve build process and recompile the service together with the whole GridSolve package. This is more stable when installing the service and GridSolve in a different location on several computers. We also found that this approach holds when modifications to the underlying GridSolve sources (e.g. updates, fixes) are done. Thus we have facilitated this process (with the tools gs-service-creator and gs-build-services) within giggle.

- `gs-service-creator` ((1) in Fig. 4) generates a directory structure that contains the required files for creation of a new service. The gs-service-creator is template-based. Currently two templates – plain C RPC and Matlab compiler runtime RPC – are supported. Furthermore a mechanism is included that supports the transport of dynamically linked shared libraries to the server.
- `gs-build-services` (2) is the tool that organises the compilation of GridSolve and the service. It hides the complexity of integrating our service into the GridSolve build system. The tool compiles the sources in a temporary location, where the GridSolve build specifications are modified to integrate the new service. The compilation process also recompiles GridSolve. This makes the build procedure longer, but we found that this ensures a higher level of reproducibility, flexibility and robustness against changes of the service or the underlying version of GridSolve. The output of the build process is a package (.tar.gz) file that contains the service. Optionally a GridSolve distribution tarball can be created.

After a successful build, the generated packages have to be deployed on the webserver to be available for download by grid jobs.

4.3 Giggle tools for resource allocation

The gLite resource allocation is done via the gLite User Interface (UI). We provide three tools that avoid logging into the UI and manage the involved gLite jobs. Furthermore the authentication to the grid is taken care of.

- `gs-glite-start` (3) launches the resource allocation. This tool starts a chain of several steps: First it starts up the GridSolve components (agent and optionally the proxy) on the servicehost (3a). Then it instructs (3b) the gLite User Interface (UI) to submit a given number of gLite-jobs (3c) to the resources of the interactive grid project. The jobs download (3d) and install required dependencies and start the GridSolve server. The server connects to the agent. Then the WorkerNode is available to the user via the GridSolve client.
- `gs-glite-info` can be used to display a short summary of the gLite jobs.
- `gs-glite-stop` (5) When the user is done using the grid, gs-stop-grid frees the allocated resources and termindates the GridSolve daemons. Otherwise resources would remain allocated but unused.

Currently these tools rely on passwordless ssh in order to connect to the gLite User Interface machine. There the user commands are translated to gLite commands.

4.4 Giggle tools for the end-user

`gs-start-matlab` (4) configures a users Matlab session so that it can access GridSolve resources. This involves configuring Matlab to find the local GridSolve client installation as well as directing the client to the previously started agent.

Please note that Matlab is taken as an example representative of many other applications. Being available for Java, C, Fortran, Octave and more the GridSolve client can be used from various programming languages in different applications.

5 MEASUREMENTS

We made a performance evaluation based on the CPU intensive loop benchmark. This benchmark allows to specify the number of loops or iterations and the amount of CPUs to be used. Advantages of the loop benchmark are

- no communication: this is an important factor for scaling
- linear scaling: two iterations take twice as long as one
- even distribution: every CPU computes the same amount of iterations.

With this benchmark we measured the performance characteristics of our solution. This will resemble both: the overhead introduced by GridSolve as well as an effect that originates from the different speeds of the CPUs that are assigned to the problem. This effect may be called unbalanced allocation.

Within this series of measurements we can modify three parameters:

- Number of gLite WNs. This is not the number of CPUs because we have no information about how many CPUs are installed in the allocated machines. GridSolve however used all that are found.
- Number of processes that we use to solve our problem.
- Amount of iteration that we want to be computed.

We chose iterations between 10 and 1000 which correspond to 6 s and 10 min runtime on a single Pentium IV-2400 CPU. The iterations were distributed to the resources provided by the production infrastructure of int.eu.grid. Resources were allocated at SAVBA-Bratislava, LIP-Lisbon, IFCA-Santander, Cyfronet-Krakow and FZK-Karlsruhe. Typically we have allocated 5 to 20 WorkerNodes (WNs), most of which contain 2 CPUs. The precise number of CPUs is unknown.

We measured the time to compute the iterations over the number of CPUs used. Measurements were repeated 10 times for averaging purposes. The graphs show the inverse of the computing time as a function of the number of processes into which we divided the benchmark. Out of the 10 repititions of each measurement we show the average of all measurements as well as the maximum and minimum curves. The difference originates from the dynamic nature of the grid. If one server exceeds a time limit, it will be terminated. In this situation GridSolve chooses a different resource where this part is computed again. This leads to a prolonged computation of the whole benchmark, hence a difference between the “min” and the “max” curve. The min curve resembles a better result while the max curve stands for the longest runtime of the benchmark. We refer only to the theoretical and the min curve further in this discussion. The graphs also show a theoretical curve, which is based on the assumption of ideal scaling (i.e. if doubling the number of processes, the result will be computed in half the time

The difference between the maximum and the minimum curve Another reason for this difference is the unbalanced allocation of CPU speeds that we have mentioned above. Since these effects are of a random nature it is not possible to compare them between the different graphs.

Results are shown in Fig. 5 to 7.

The comparison between a measurement with a low number of iterations (Fig. 5) and one with more iterations (Fig. 6) (6 s and 5 min respectively) reveals the overhead that is caused by GridSolve. We observed that this overhead depends on the amount of CPUs that we utilise and occurs mostly during submission and collection of results. This is also indicated by the increased computing time for more than 4 CPUs in Fig. 5.

In all measurements we can find one point at which the curve stops following the theoretical prediction and continues horizontally. This indicates that the compute time does not decrease even if further increasing the number of parallel processes. This is because

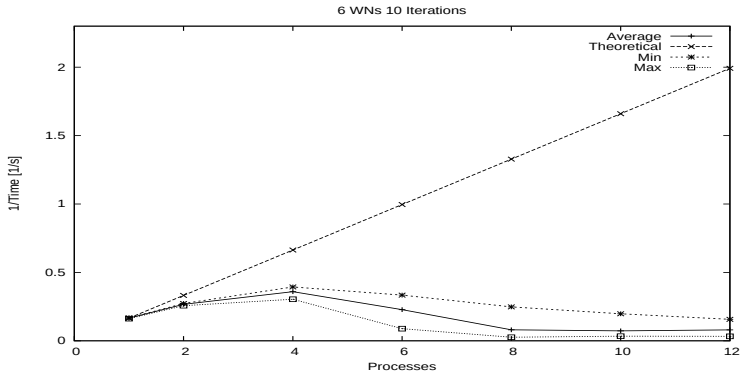

Fig. 5. 6 gLite WNs allocated. 10 iterations require 6 s on a Pentium-IV CPU.

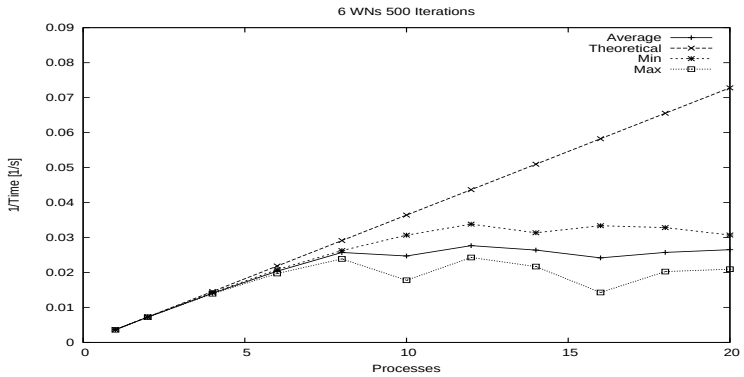

Fig. 6. 500 iterations run 5 min on a Pentium-IV CPU, 30 s on the grid.

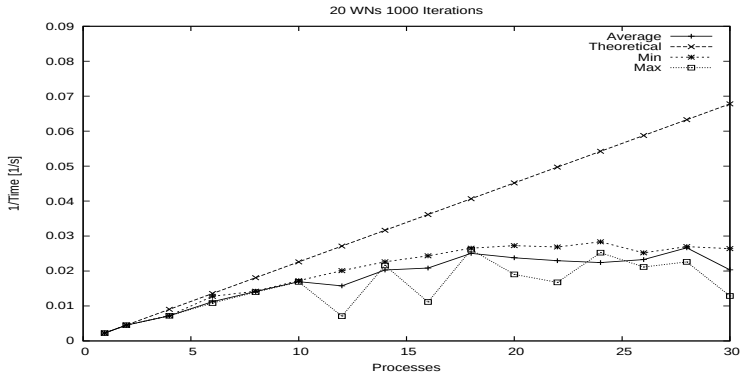

Fig. 7. With 20 WNs allocated and 1000 iterations we can the system scale to 18 processes.

at this point the number of available CPUs is smaller than the number of processes. The theoretical curve does not take this effect into account.

In 6 and 7 we can see that even the min-curve grows slower than the optimum (theoretical) curve. This is due to the heterogenous nature of the grid. In our pool of resources, we have CPUs with different speeds. GridSolve allocates the fastest CPU first and only uses slower CPUs when all fast CPUs are already busy. As the theoretical curve is extrapolated from the performance on only one CPU, it resembles the ideal scaling behaviour. In Fig. 6 this can be observed at 6 and more processes, in Fig. 6 we can see this effect already starting with 4 processes.

6 CONCLUSION

We are able to show that both GridSolve and its integration with gLite via giggle work well. Scaling of resource useage works in principle, but certain obstacles are still to overcome. The showstopper regarding proper scaling is however, the fact that the underlying resources disappear. Nevertheless Gridsolve provides a result in a fault tolerant manner.

We have reached the goal of using the gLite infrastructure in an interactive, API-like fashion from problem solving environments like Matlab.

7 *

References

1. E. Caron, F. Desprez, F. Lombard, J.-M. Nicod, L. Philippe, M. Quinson, and F. Suter. A scalable approach to network enabled servers (research note). *Lecture Notes in Computer Science*, 2400, 2002.
2. EGEE Project. gLite website. glite.web.cern.ch/glite.
3. K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631, May 1994.
4. Global Grid Forum Research Group on Programming Models. `http://www.gridforum.org/7_APM/APS.htm`.
5. Interactive European Grid Project. Website. www.interactive-grid.eu.
6. K. Moore. Recommendations for the Design and Implementation of NAT-Tolerant Applications. Internet-draft, February 2002. Work in Progress.
7. H. Nakada, M. Sato, and S. Sekiguchi. Design and Implementations of Ninf: Towards a Global Computing Infrastructure. In *Future Generation Computing Systems, Metacomputing Issue*, volume 15, pages 649–658, 1999.
8. 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.
9. Y. Tanimura, K. Seymour, E. Caron, A. Amar, H. Nakada, Y. Tanaka, and F. Desprez. Interoperability Testing for The GridRPC API Specification. Open Grid Forum Informational Document, February 2007.