Cluster Computing Facilities in a Service-Oriented Architecture

Dana Petcu[1,2], Marcin Paprzycki[3,5], and Maria Ganzha[4,5]

[1] Computer Science Department, Western University of Timişoara,
[2] Institute e-Austria Timişoara, Romania
petcu@info.uvt.ro
[3] Institute of Computer Science, SWPS
[4] Elbląg University of Humanity and Economics
[5] Systems Research Institute Polish Academy of Sciences, Warsaw, Poland
{maria.ganzha, marcin.paprzycki}@ibspan.waw.pl

Abstract. When considering the development of the grid as the universal computing infrastructure, one of the important issues that has to be faced is "porting" legacy codes to the grid. In this paper an approach to wrapping cluster codes as grid services using the latest generation of grid middleware is analyzed. This approach was derived from several experiments of wrapping legacy codes. The application-depend parts of the wrapper are discussed and communication templates are presented.

1 Introduction

Grid computing is sometimes confused with cluster computing. The key difference is that a cluster is a single set of computer nodes, residing in a single location, while the grid is composed of a variety of resources that can be (and most often are) geographically distributed. Among these resources there could be single-processor computers, data repositories **and** cluster computers. While there are differences between grids and clusters, it is already clear that there always going to be a significant relationship between them: within grids there always will be a place for computer clusters. There are at least two reasons for this. First, certain classes of problems will always require a relatively tight coupling of processors. Second, cluster computing is one of the most price-efficient ways of building powerful computers with large amounts of available memory (computational power and the total amount of available memory are the two main reasons for parallel computing; see for instance [12]). However, it can be already observed that as capabilities and in particular bandwidth of existing computer networks advance, problems that were previously the exclusive domain of cluster computing become solvable also utilizing grid computing. At the same time, advances in grid computing will result in constant increase in the total number of problems that utilize cluster nodes made available as services within grids.

In this context we would like to specify that an application is grid-enabled when it can run in a grid. According to [3] there are six path/strategies for grid application enablement (and, in particular, for grid-enabling cluster computing codes): (1) batch anywhere; (2) independent concurrent batch; (3) parallel batch; (4) service; (5) parallel services; (6) tightly coupled parallel programs. Those strategies start with the simplest case of running a given code somewhere within a grid, and end with the most complicated option of fully exploiting the grid as a computational infrastructure treated as a parallel computer. Approaching it from a slightly different angle, the first two strategies focus on the ability of an application to run within a grid, the next three significantly adapt the function and value of the application by enabling it to use a grid without requiring significant changes that are specific to the grid middleware, while the latest one explicitly exploits advanced features of the grid infrastructure for its operation.

In this paper we are interested in the fourth and fifth strategies allowing the transition of cluster computing codes from a batch to a service-oriented architecture. In the fourth path, the client uses the grid middleware to invoke the service that further calls the service that is structured to be callable (typically as a class). In this case the client and the server are loosely coupled. Furthermore, the same service can be shared among multiple independent clients and can maintain its state between calls. However, it is the fifth strategy that provides multiple service instances and permits these instances to be callable in parallel on client's behalf. It is this path that is of particular interest of ours as it is the way in which most successful (from the point of view of resulting computational efficiency) transition of legacy cluster codes into the grid can be expected.

Let us now focus our attention on service oriented computing. The OASIS organization [11] defines the service oriented architecture (SOA) as follows: a paradigm for organizing and utilizing distributed capabilities that may be under control of different ownership domains; SOA provides uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.

The attraction of the SOA is that it builds on concepts of reusable software components, while emphasizing the service abstraction. This means that ideally, SOA services are interoperable, reusable, independent, stateless and autonomous. Furthermore, to enable interoperability, services should be composable, loosely coupled, and standards compliant. In the SOA environment, resources available across a network are made available as independent services that can be accessed without knowledge of their underlying platform implementation. The primary focus of the SOA latest developments is on dynamic reconfiguration of services, and on developing business services based on Web services and, more recently, on grid services. It is the latter point that is of our interest in this paper. Let us therefore briefly underline some of the benefits of using grid services (rather than Web services):

– grid services are statefull—output may depend on the history of its calls (this fact can be exploited as it is demonstrated in the third example presented in the next section);

– service is named by a service handle that does not provide information like the location, implementation, or status of the service;
– dynamic discovery and monitoring of the services can be achieved;
– grid services are instantiated by factories that can implement a load balancing strategy;
– grid services provide a natural solution to the problem of resource reclamation (associated with services in the event of failures or lack of interest by any relevant clients); specifically, time can be set when a specific service will self-destruct unless kept alive by subsequent increases in its termination time; the service instance does not need to be explicitly destroyed, since it will be terminated automatically.

In this context, our aim is to introduce our approach to wrapping cluster codes as grid services using the latest generation of grid middleware. In the next section we present an overview of our proposal. We follow with an analysis of parts of the wrapper that are application-dependent and illustrate them using three different grid-implemented examples. Finally the standardization of the client code is taken under consideration. Here, we introduce the recently proposed *ACliGs* toolbox and illustrate its usage when wrapping [17], a software toolbox for the numerical solution of partial PDEs.

2 Wrapping a cluster-based code as grid service

To be able to expose legacy (cluster) codes as grid services they have to be appropriately wrapped up. Recently, some tools have been developed to automate the deployment of legacy codes into the grid (e.g. GEMLCA [2]). Unfortunately few of these tools support the last generation of grid middleware.

Using the latest version of Globus Toolkit the task of wrapping (parallel) legacy codes is simplified. Performing such a wrapping task for several codes, a template for time-steps and communication can be gradually revealed. In what follows we present such a template. Figure 1 describes a simple approach. The parallel legacy codes are represented in this figure by the service components.

Let us now describe the process depicted in Figure 1. The client knows the address of a register of services and queries it about a specific (cluster) service. The service register sends back the address of the service factory that matches the query. The client contacts the service factory and requests a service instance. The service factory creates an instance of a service interface. Then the client sends to the interface starting parameters (e.g. names of input files) needed by the remote code—in our case a cluster code. The service interface will launch the code of interest, and:

– if the code will run on the server, then the code will be launched by a thread of the service interface;
– if a cluster scheduler is installed on the server in conjunction with the grid middleware, then the code will be launched on the cluster by the scheduler that is called by a thread of the service interface;

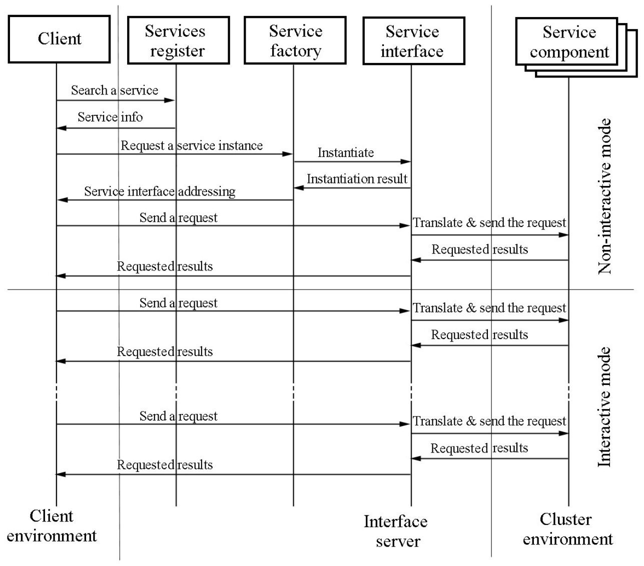

Fig. 1. Time steps and communications necessary in order to perform cluster operations

– if no scheduler is installed and the server is not part of the cluster, but lies in the same security domain as the cluster, the remote code can be invoked through classical rsh/ssh commands by a thread of the service interface.

At this stage the service interface will establish the necessary communication channels with the (cluster) code.

There are two types of possible interactions with the remote code:

non-interactive mode: the code receives the inputs sent by the service interface, executes, sends back results, and stops (top part of Figure 1);

interactive mode: the code is activated by the service interface and waits the inputs on the communication channels from the service interface; it stops only when the input coming from the service interface requests it (bottom part of Figure 1).

Note that the service interface and the client should include facilities for secure file transfers (e.g. input files for simulation codes).

When using the latest version of the Globus Toolkit, the client, the factory service, and the service interface are almost generic for all of the above described cases. Therefore, we concentrate our attention and in the next sections discuss those parts that are application-dependent.

3 Examples of service interface description

As mentioned above, the service interface consists of two parts. One is application-independent while the other important part is application-dependent. We will illustrate differences between these two parts in three case studies where:

1. the service interface is playing the role of the work distributor, and the service components are codes that are running independently of each other (embarrassing parallelism);
2. the service interface activates the (parallel) code in a non-interactive mode;
3. the service interface activates the (parallel) code in an interactive mode.

In these three examples different styles of communication between the service interface and the service components (application-dependent) were tested: communication through exchange of files, through a notification service, and through sockets.

3.1 Parallel image processing

Let us consider the case of applying a decision tree-based classification algorithm to a large image file. If the test image is a (high resolution) satellite image, the classification algorithm running on a typical current desktop computer will take several minutes to complete. The same computation can be done much faster on a cluster computer. Furthermore, an almost ideal speedup can be attained since the image can be cut into smaller pieces, and then the algorithm can be applied to each image fragment independently in parallel. Output of each part is a new sub-image, and the final classification image is obtained by merging these small output images.

The service interface in this case will play the role of (a) splitting the image to be processed into smaller sub-images, (b) launching several service components (on separate cluster nodes) that are applying the classification algorithm, and (c) merging the processed image-pieces into the final result.

We have tested the architectural approach using GIMP [6] as the service component. GNU Image Manipulation Program, shortly GIMP, is a freely distributed software designed for image processing tasks, like image composition. GIMP can be run in a non-interactive mode (using appropriate scripts), which is useful when remote processing is needed.

Factory and interface services and the client were constructed as generic as possible. In this way also other image processing algorithms can be applied in the proposed solution (the GIMP library can be simply replaced by another image processing library). Client inputs (to be send to the service interface) consist of the following:

1. an image reference;
2. number of image parts;
3. a GIMP script-file to be applied on each image part.

In our work, we have considered the simplest case when the cluster has NFS directories and the image to be processed is located somewhere within those directories. Therefore the image reference is pointing to the specific file location. The script-file is located on the user side and it is transferred into the cluster using the *gridftp* component of the Globus Toolkit (as parts of the client and service interface codes). Optionally, if necessary, the initial image and the final image can be transferred to and from the cluster's NFS directories (to the client desktop or to any remote machine). The only difference will be the total processing time as an extra staging and post-processing data transfer times will be required.

After the script is received from the client, the service interface (running on a web server) splits the specified image into a predefined number of parts, launches the requested number of GIMP servers in the non-interactive mode—each on a single remote node of a local cluster (using ssh), specifying the name of the (small) image file and also the script file to be applied to it. Note that here we assume that each node of the cluster is a single-processor node. In the case when cluster nodes were multi-processor and/or multi-core then an appropriately larger number of GIMP services per node would be started. The service interface starts the composition of the output image as soon as processed sub-image parts are available (here, communication is achieved through file transfer).

Services used in this application, the web-based client, and the test results were described in more details in [15]. Here we will only mention that an efficiency of 62% on 9 processors was achieved.

3.2 Parallel simulation

Let us now consider the second example, where a computational fluid dynamics (CFD) simulation code is wrapped as a grid service. CFD problems require a lot of computing time and a huge memory to execute; typically they generate a very large amount of resulting data; however, the most important limiting factor is the computing power. CFD is a "greedy" consumer of computing resources, requiring special parallel computing algorithms. Since solution of large-scale CFD problems on parallel computers have been studied for a long time now, a large number of well-developed CFD codes and libraries are currently available (see for example an overview of these codes in [9]). Numerical simulations performed by these codes are applied to many industrial and scientific problems. In this context, the main challenge is to find comfortable ways to re-use these codes in the grid. Note also that very often CFD codes rely on a number of external tools (converters, mesh generators, visualization tools, and so on). The installation procedure and the management of a large number of necessary utilities and their versions is a time-consuming task and should be delegated. In this context one of the problems is to find comfortable ways to access remotely installed codes. Industry requirements are strict in what concerns intellectual property of all research results. Therefore, when talking about reusing codes and remote access, the security of the environment is essential. One current practical solution for the three above described requirements (re-use, remote access and security) is to wrap legacy codes as grid services and to deploy them in a grid environment.

Usually a CFD code takes as inputs several files describing, in a specific language, the problem to be solved. Typically, its output files are large and contain simulation results that are then interpreted / rendered by image processing / visualization tools.

For our tests we have selected Gerris [5], an open source software library for the solution of the partial differential equations describing fluid flow, allowing spatial discretization with automatic and dynamic local refinement, and utilizing a multigrid Poisson solver. Gerris uses an MPI library for parallel computations and the GTS [8] library (a GNU Triangulated Surface Library, also an open source software library that provides a set of functions to deal with three-dimensional surfaces meshed with interconnected triangles). In the application, additional external tools are used for visualization of results and for format conversions.

Factory and interface services and the client were, again, constructed as generic as possible. Therefore, in the resulting application, the Gerris library can be easily replaced with a different CFD code. The client inputs (to be sent to the service interface) consist of:

1. a simulation file;
2. number of processes used by Gerris;
3. the name(s) of the resulting file(s) to be transferred to the client side.

The service interface is much simpler than in the previous example. After it receives the above mentioned parameters from the client and after the simulation file, located on the client side, is transferred at the site where the Gerris is available (staging process), the service interface launches a thread that activates the Gerris code. Since the Gerris is a parallel code itself, there is no need to initialize its instances on each cluster node separately (this is done by the Gerris itself). The communication between the service interface and the components is done by a notification component that is informing the service interface when the simulation is completed. At this moment, the resulting file(s) is(are) transferred back to the client side using gridftp facilities.

More details about the grid-enabled Gerris service can be found in [16].

3.3 Parallel interactive computations

Let us now consider the third example, where a software tool working usually in an interactive mode is wrapped into a grid service. In this case the client will be required to “constantly” send new inputs to the remotely activated code.

To facilitate this scenario the service interface has to launch at least two threads: one that starts the remote software component and one that allows constant communication with it. The means of communication will depend on facilities provided by the remote software.

As an example of such a scenario we discuss Maple2g, a recently proposed grid-enabler for the Maple computer algebra system. Maple2g has a specific component that allows the start of several Maple processes on a cluster in a master-worker style. These processes are able to cooperate by exchanging messages to solve a complex problem if the user gradually provides the master process with a correct description of the work and communication to be completed. A detailed description of the component under discussion was provided in [13].

Factory and interface services as well as the client were, again, constructed as generic as possible. Therefore, Maple can be easily replaced with another computer algebra system (e.g. Mathematica or Mathcad). The client inputs (to be sent to the service interface) consist of:

1. initially, the request for launching the remote Maple master process;
2. gradually, command lines to be interpreted by the Maple master process.

The service interface for the Maple2g-cluster component launches the Maple master process that reads two special files: one with the Maple2g definitions and another that tells Maple to open a socket connection (as a server) with the service interface and wait for commands. Then the service interface establishes the connection (as client) with the master process—as soon as a new command has arrived from the client. Such commands can include the directive to launch other Maple processes (done in Maple2g using MPICH or MPICH-G2). Details about the service interface as well as some usage examples can be found in [14].

4 Towards automated generation of the client code

The concept of a single specialized client for each remote grid service is unfortunately not scalable. Every time a service is created, corresponding clients must be written. But in a grid environment, the user should be able to discover these services and dynamically interact with the service interface, without having to install a new client. Several recent tools are available to encapsulate and execute complex applications as Grid services.

1. The ASSIST [1] provides the application developer with a proxy library whose entries are stub methods for the remote Web Service. However, user intervention is required to generate the final code.
2. A .NET tool, Webservice Studio, can be used to invoke Web service methods interactively. This tool is meant for Web service implementers to test their services without having to write the client code or to access other Web services. Unfortunately Webservice Studio does not work well with complicated methods.
3. A complex service called Xydra-OntoBrew [4] offers an automatic solution to client creation problem for simple services and portlet clients. Unfortunately, Xydra is a rather complex response to the client code creation problem, while quite often simpler solution could be used; especially when a workflow execution of combined services is desired.

Table 1. The methods of the ACliGs class

Method	*Aim*
getListServ(url)	Get the contact list of the registered Grid services.
queryList(url,name)	Get the contact list of service's names that are (partially) matching the given name.
getWSDL(servContact)	Get the (full) WSDL of the specified Grid service and write it as a local file labeled with the service name obtained from the (short) WSDL file provided by directly questioning the service.
getServName(servContact)	Get the service name by inspecting the (short) WSDL file.
getListOp(servName)	Get the list of the exposed operations of the service
getTypeOpIn (servName, opName)	Get the types of the input arguments of the specified operation.
getTypeOpOut (servName, opName)	Get the type of the output of the specified operation.
buildStubs(servName)	Build the stubs classes needed to access the service.
buildClient (servName, opName)	Build a client to invoke the specified operation of the named service.
transfer(direction, list_of_fileNames)	Transfer the named files from the client to the server side (direction=0) or from client to the server (direction=1)
buildNotifClient (servName)	Build a simple notification consumer

In this context, package for *A*utomatic-creation of *Cli*ents for *G*rid *S*ervices, shortly *ACliGs* was proposed in [16]. *ACliGs* was written in Java, for client-server interaction within grids based on the Globus Toolkit 4 [7] (GT4), de facto standard of the current Grid middleware. While the complete description of *ACliGs* can be found in [16], let us present here, in Table 1, methods of the *ACliGs* class.

Interestingly, this relatively small number of methods allows us to support all functions necessary to build grid service clients (for a complete discussion, including details of wrapping of the Gerris package as a grid service—described briefly in Section 3.2, see [16]). Let us now illustrate how the *ACliGs* can be used to build grid service clients. For this purpose we will utilize the UG [17] software toolbox for the numerical solution of partial differential equations on unstructured meshes. UG's main features are robust multigrid solvers for unstructured, locally refined meshes, which enables it to be applied to diverse application fields like Navier-Stokes equations, elasto-plasticity, density driven flow and multiphase flow in porous media. UG contains a large library of parallel solution methods for linear, nonlinear and time dependent problems and a parallel grid manager that is capable of local refinement and coarsening of unstructured meshes in two

Table 2. Operations exposed by the grid service wrapping UG' codes

Operation	*Aim*
setModule(String module)	Select the module to be used: diff2da, fe2d, fe3d, or ns2d, ns3d, sc2d; by default fe2d
setNoProcs(int number)	Indicate the number of processors participating in the simulation; by default 1
exec(String script)	Launch the simulation described in the script file
getExample()	Get an example of a script file for the active module
getHelp()	Get a user manual for script language
getStatus()	Get the status (Active, Done or Error) of the simulation; by default the status is Done

and three space dimensions. Specifically, UG uses domain decomposition with message passing paradigm and takes the adaptive dynamic load balancing into account. UG was recently used to solve equations on two-dimensional unstructured grids with a fully implicit finite volume discretization scheme and parallel adaptive multigrid [10] and with good results on a parallel computer with thirty processors and running MPI. We have tested the UG wrapper by running its example scripts in a smaller cluster environment using PVM.

A simulation in UG is started by a command line that specifies: (a) the executable name, (b) *ugrun*, (c) a module name, e.g. *ns* for incompressible Navier-Stokes equations, (d) number of processors to be used in the simulation, and (e) name of the script file with simulation description written in a specific language (UG commands that are interpreted by the UG shell). While modules describe problem classes and solving procedures, the script specifies parameters that define a member of the problem class as well as the combination of solving procedures. We have wrapped UG' codes as a grid service and specific service operations are described in Table 2. The status of the simulation and the execution type are stored as resource properties.

Finally, a simple example of using UG' scripts in the case of the UG service is given in Figure 2. In this case some file transfers take place: an example of a script file for simulation and the results of the simulation (images).

5 Concluding remarks

In this paper we have focused our attention in porting legacy cluster-computing codes to the grid. We have discussed three practical examples that illustrate different scenarios involved in gridifying cluster-based codes. Observations made in these three examples were used as a backdrop for introducing the *ACliGs* toolbox for building grid service clients. Usage of *ACliGs* was illustrated in the case of the UG package for solution of PDE's on unstructured meshes. The *ACliGs* package is currently being further tested as in the near future it will be used to build a grid environment and a portal for the NanoSim project aimed

```
>     queryList http://194.102.62.15:8080/ UG
http://194.102.62.15:8888/wsrf/services/cfd/UGService
1 entries
>     getWSDL http://194.102.62.15:8888/wsrf/services/cfd/UGService
UG.wsdl available in the local directory
>     buildStubs UG
Building stubs... Successful
Log file: UG.log
>     buildClient UG setModule; buildClient UG exec; buildClient UG getExample;
 Building client... Successful
 Log file: ClientUG_setModule.log
 [... omitted here]
>     ClientUG_setModule diff2da; ClientUG_getExample
Transfer Example.scr ... successful
>     vim Example.scr
 [... omitted here]
 # solver: multigrid; smoother: Gauss-Seidel
 solver = "mgc"
 [...]
 smoother = "gs"
 [...]
 # problem
 new square $b model problem $f full scalar $h 4000000;
 [...]
 # output
 openwindow 10 100 320 320 $n Grid-sq;
 [...]
>     #modify the output into a PostScript file
 [...]
 openwindow 10 100 320 320 $d ps $n Grid-sq.ps;
>     transfer 0 Example.scr
Transferring files ... Successful
>     ClientUG_exec Example.scr
>     transfer 1 Grid-sq.ps Er.ps
Transferring files ... Successful
>     gimp Grid-sq.ps Er.ps
```

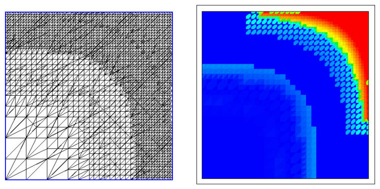

Fig. 2. An example of using the ACliGs' scripts to build and launch clients of UG grid service

at deployment and interconnection of multiple CFD codes. It will be also used to deploy symbolic solvers as Grid services.

Acknowledgement

Research of Dana Petcu was partially supported by the projects NanoSim CEEX-I-PC-D03-PT04-439 and MedioGrid 19CEEX-I03/2005 funded by the Romanian Ministry of Research.

References

1. Aldinucci, M., Danelutto, M., Paternesi, A., Ravazzolo R., and Vanneschi, M.: Building Interoperable Grid-aware ASSIST Applications via Web-Services, Proc. of Intl. PARCO 2005: Parallel Computing, 2005, and http://compass2.di.unipi.it/TR/Files/TR-05-24.pdf.gz.
2. Delaitre, T., Kiss, T., Goyeneche, A., Terstyanszky, G., Winter S., and Kacsuk, P.: GEMLCA: Running Legacy Code Applications as Grid Services. Journal of Grid Computing **3**, no. 1-2, June 2005, pp. 75–90.
3. Kra, D.: Six Stategries for Grid Application Enablement, Part 1: Overview. On-line at http://www.ibm.com/developerworks/grid/library/gr-enable/.
4. Gannon, D., Alameda, J., Chipara, O., Christie, M., Dukle, V., Fang, L., Farrellee, M., Kandaswamy, G., Kodeboyina, D., Krishnan, S., Moad, C., Pierce, M., Plale, B., Rossi, A., Simmhan, Y., Sarangi, A., Slominski, A., Shirasuna, S., and Thomas, T.: Building Grid Portal Applications from a Web Service Component Architecture, Procs. of the IEEE Publication **93**, Issue 3, 2005, pp. 551– 563.
5. Gerris, http://gfs.sourceforge.net/.
6. GNU Image Manipulation Program, http:// www.gimp.org/
7. Globus Toolkit, http://www.globus.org/toolkit/.
8. GNU Triangulated Surface Library, http://gts.sourceforge.net/.
9. Magnus, A., Tveito, B.A. (eds.): Numerical Solution of Partial Differential Equations on Parallel Computers, Springer, 2005.
10. Mo, Z.: Parallel adaptive solution for two dimensional 3-T energy equation on UG, Journal Computing and Visualization in Science **9** (3), 2006, pp. 165–174
11. Organization for the Advancement of Structured Information Standards, http://www.oasis-open.org/
12. Paprzycki, M., Stpiczynski, P.: A Brief Introduction to Parallel Computing. In: E. J. Kontoghiorghes (ed.), Handbook of Parallel Computing and Statistics, Taylor and Francis, Boca Raton, FL, 2006, pp. 3-41
13. Petcu, D., Dubu, D., Paprzycki, M.: Grid-based Parallel Maple. Procs. PVMMPI 2004, Budapest, Hungary, September 19-22, 2004, eds. D. Kranzmüller, P. Kacsuk, J. Dongarra, LNCS **3241**, 2004, pp. 215-223.
14. Petcu, D.: Improving Computer Algebra Systems by Using Grid Services, Procs. 1st Austrian Grid Symposium, J. Volkert, T. Fahringer, D. Kranzlmüller, W. Schreiner (eds.), Austrian Computer Society, Band 210, ISBN 3-85403-210-2, 2006, 102-110
15. Petcu D.: Grid Services for Satellite Image Processing. WSEAS Transactions on Computers, Issue 2, Vol. 6, ISSN 1109-2750, 347-354, 2007.
16. Petcu, D.: Automatic Generated Clients of Grid Services for Computational Fluid Dynamics. Procs. MATH'06, 10th Internat. Conference on Applied Mathematics, Dallas, Texas, USA, November 1-3, 2006, ISBN 960-8457-55-6, pp. 96-101
17. UG, http://sit.iwr.uni-heidelberg.de/˜ ug/.