



IST-2001-32133

GridLab - A Grid Application Toolkit and Testbed

D 3.7 Triana Remote Steering

Author(s):	Matthew Shields and Ian Wang
Document Filename:	GridLab-3-D3_7-0001-TRS
Work package:	WP3 Work-Flow Application Toolkit (TGAT)
Partner(s):	Cardiff University
Lead Partner:	Cardiff University
Config ID:	GridLab-3-D3_7-0001-1.0-Draft_A
Document classification:	PUBLIC

Abstract: This document outlines a mechanism for performing remote steering of applications from within a workflow running in the Triana environment. The remote application will be executed and steered on the GridLab Testbed using the GAT integrated with Triana and *GridMonSteer*, a generic computational steering architecture for grid environments, developed in Cardiff



Last amendment date: 2005/03/15 & time: 17:50:03

Contents

1	Introduction	2
2	GridMonSteer	2
2.1	Architecture	2
2.2	Monitoring Arguments	3
2.3	Steering Arguments	4
2.4	Examples	4
2.4.1	Example 1	5
2.4.2	Example 2	5
2.4.3	Example 3	5
3	Remote Steering Within Triana	5
3.1	An Example Experiment	5
3.2	Triana Workflow Implementation	6
4	Conclusion	7

1 Introduction

Remote steering of applications is an area that is receiving increasing attention. Typically a scientist with a long running computational experiment would like to receive feedback on the progress of the experiment and perhaps interact with it. Unfortunately *steering* an application is only really possible if the ability to do so has been engineered into the application code before hand. Often it is not possible to re-engineer an existing application to incorporate steering and writing a new application from scratch is too costly. In this scenario we will look at an application of computational steering that is non-invasive and can be used with any suitable application with no modification. The application is run within *GridMonSteer* an application wrapper that is responsible for executing the application, providing the application with its input parameters and returning output parameters to the steering environment, in this case Triana.

A typical research scenario for computational scientists is to perform “parameter runs” on new or existing applications to examine the effect of parameter ranges on the result. An application is run multiple times in succession with perturbed parameters in order to find a particular solution to a problem. The changes to the input parameters could be performed in a brute force manner, searching for all combinations, or more subtly where the changes to the parameters are *steered* toward an eventual solution. It is this form of remote steering that we will consider in this document.

2 GridMonSteer

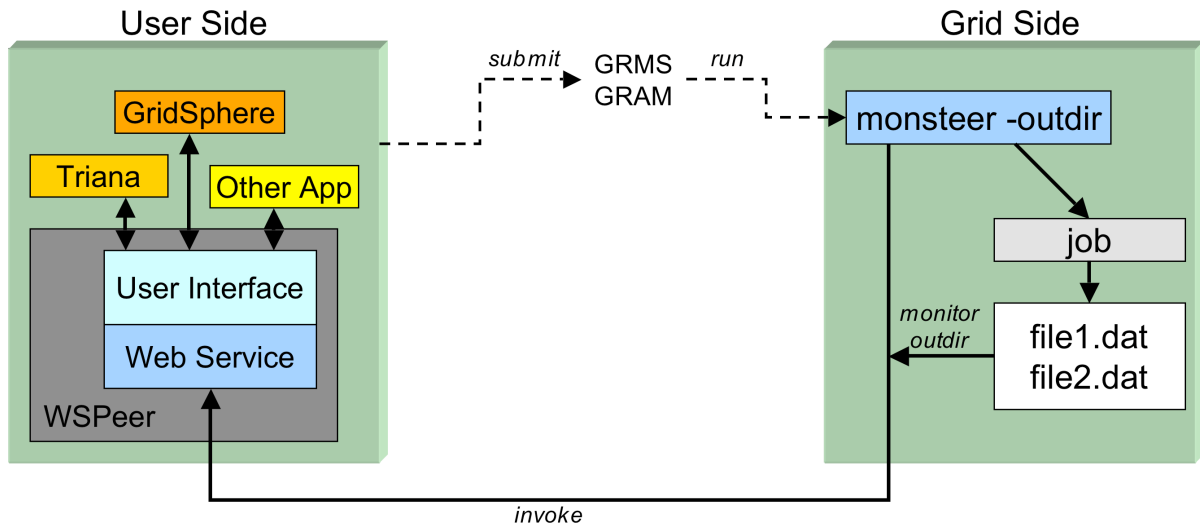
GridMonSteer is a generic computational steering architecture for grid environments. It is grid infrastructure independent and can be used in environments running any grid resource manager, including GridLab GRMS [1], Condor/G [2] and Globus GRAM [3].

2.1 Architecture

The *GridMonSteer* architecture can be seen in figure 1. The architecture is split into two distinct components, the *User Side* where the client application such as Triana, GridSphere or any other grid aware application sits, and the *Grid Side* where the GridMonSteer wrapper and the application it is controlling sit.

Communication between client side and grid side is also split. The initial job submission to start GridMonSteer and its job application is via a grid resource manager, such as GridLab GRMS or Globus GRAM [4]. The communication from GridMonSteer back to the client application is via web service calls. Software developed at Cardiff University called WSPeer allows the client application (e.g. Triana) to dynamically deploy itself as a web service and to receive web service invocation calls. The benefit of this model is that all the communication after the job has executed originates from the grid side, bypassing problems with firewall.

The calls from the GridMonSteer application include requests for input parameters and input files before the job is executed, and notification of output files while and after the job has finished. The actual calls that are issued depend on the arguments that are passed to GridMonSteer when it is executed by the grid resource manager. In the next sections we outline the monitoring and steering arguments that can be passed to GridMonSteer.


 Figure 1: *GridMonSteer* Architecture

2.2 Monitoring Arguments

The set of gridMonSteer arguments that we refer to as monitoring arguments are used to specify the information that is sent from gridMonSteer (grid side) to the controller (user side). The following arguments are used to specify which files output by the legacy job are notified to the controller, and the notification policy used:

- out - Sends output files to the controller after job execution has finished.
- monitor - Same as -out except the output files are sent immediately after their creation.
- update - Same as -out except incremental updates to the output files are sent periodically during job execution.

Each of the arguments above take a file list which details the names of files and/or directories to be monitored, and may include wild-cards (e.g. *.jpg). The files detailed in the file list are all given relative to the execution directory of the job. In addition to standard file/directory names, the keywords `STDOUT` and `STDERR` can be used in the file list to specify monitoring the standard output and standard error streams respectively. For example, the argument `-update STDOUT` indicates periodic updates from the standard output stream should be sent to the controller.

Although the basic file notification arguments detailed above cover most monitoring situations, in some scenarios applications generate a large number of output files of which the scientist is only interested in a subset, and this subset can only be determined based other output files. For these scenarios gridMonSteer uses a register/select system, whereby output files are registered with the controller when they become available and gridMonSteer requests the controller/user to select the files that are of interest. The request to select files is made repeatedly to the controller so that the set of files selected for notifications/updates can be dynamically tuned. The `-xout`, `-xmonitor` and `-xupdate` arguments provide register/select compliments for the basic file notification arguments.

One issue that needs to be considered is how it is determined whether a file has been fully created when using the `-monitor` argument. Due to the legacy nature of the jobs running

within gridMonSteer, there is no way for gridMonSteer to know if the job has finished creating a file or is merely pausing before making further updates. The approach used within gridMonSteer to this intractable problem is that files are only considered fully created when they have not been updated for a set period of time (set using the `-urate` argument). If it is found for a particular application that semi-complete files are notified to the controller then an increased `urate` can be used, the disadvantage being less timely output information.

In addition to output file notification, the `-state` argument can be used to indicate that job state information should be notified to the controller by gridMonSteer. This state notification occurs when the job is started or stopped, and includes information such as the host name, running directory and run time. Such information is particularly useful in a grid environment where the job may reside in a job queue for some time and execute on an unfamiliar, remote resource. This information is also useful when the `-run` argument is used (see Section 2.3).

2.3 Steering Arguments

The gridMonSteer steering arguments are used to specify the information that is requested from the controller (user side) by gridMonSteer (grid side). The `-run` argument specifies that gridMonSteer asks the controller whether the job should be rerun after it has terminated. This can be used for example to rerun a small job numerous times without having to resubmit it to a grid scheduler. Obviously the same effect could be achieved executing a batch script, but when combined with the `-state` argument (see Section 2.2) this gives much finer view of the overall job progression.

Further to the benefits mentioned above, the `-run` argument also allows the controller to alter the command-line arguments on the rerun job, or even change the job completely. Effectively this can function as a steerable, dynamic batch script, offering considerable benefits over the alternative, which would be to submit a succession of independent jobs. These benefits include not incurring multiple scheduling delays and not having to deal with the complexity of the independent jobs being run in different environments.

As well as the job and arguments executed, gridMonSteer can request input files from the controller to be staged in the execution directory before the job is run/rerun. This is specified using the following arguments:

- `-in` - Requests input files from the controller prior to job execution.
- `-append` - Same as `-in` except incremental updates to the input files are repeatedly requested during job execution.

As with the equivalent arguments output files (see Section 2.2), the above commands take a file list detailing the files to be requested. This file list can include the keyword `STDIN` to indicate requesting the standard input. With the `-append` repeated requests are made to the controller, with the information returned appended to the existing file.

2.4 Examples

The following are examples of gridMonSteer usages:

2.4.1 Example 1

```
gms in param.txt xmonitor outdir cactus
```

- Pre-stages the param.txt file before Cactus is executed.
- While Cactus is executing, monitors outdir and uses register/select to let the user select from the available output files.

This is an example of the Cactus monitoring scenario used at SC04.

2.4.2 Example 2

```
gms append STDIN update STDOUT xmonitor outdir myprog
```

- Incrementally request appends to the STDIN while the job is executing.
- Incrementally send updates to the STDOUT while the job is executing.

This combination allows the user to interactively control a legacy application that is steered via the command-line.

2.4.3 Example 3

```
gms run state inspiralsearch P 0 1000
```

- Repeatedly asks whether the job is to be rerun and for the job command + arguments to be executed.
- Notifies when the job is started/stopped and the execution time.

This example demonstrates how gridMonSteer could be used to load balance distributed parallel searches. In this scenario, multiple copies of `inspiralsearch` are run in parallel. After each cycle, gridMonSteer notifies the controller how long the searches took, and the work distribution can be tuned in the next iteration through changing the job arguments.

3 Remote Steering Within Triana

In this section we will outline an example computational experiment combining GridMonSteer with GRMS job submission on to the GridLab Testbed from within a Triana workflow that performs steering.

3.1 An Example Experiment

For illustration purposes this example will use a relatively simple legacy application. We have a two dimensional boundary element solver written in Fortran. The code is used to simulate electro-magnetic wave scattering and is typical of the sorts of codes used by computational engineers and scientists on a daily basis. The main inputs to the program are two files, the first contains a two dimensional closed contour, and the second a control file defining the characteristics of the incident wave. The contour file consists of a series of x, y coordinate pairs and is

generated by a separate program. The control file is a series of values for the wave frequency, direction, amplitude and phase. The output of the code takes the form of two files one representing a radar cross section and the other a current. The input and output files are local to the program and must be in set locations in order for the program to run.

A common task performed with this type of program is a parameter space based optimisation. For instance if this simulation were used to evaluate the radar reflectivity of an aircraft flying over a radar system the scientist might run the simulation many times with different values for the incident angle of the wave, examining the output to see the difference the input values make.

We will demonstrate this example by running the solver within GridMonSteer and a Triana workflow on the GridLab testbed. The workflow will repeatedly execute the solver, changing one of the input parameter values slightly each time based on an evaluation of the output from the solver. This *cost function* evaluation will either output the modified input parameter causing the workflow to loop, executing the solver once more, or output a message if a halting condition has been met. Thus the application is steered within the workflow toward a solution for the parameter space search.

3.2 Triana Workflow Implementation

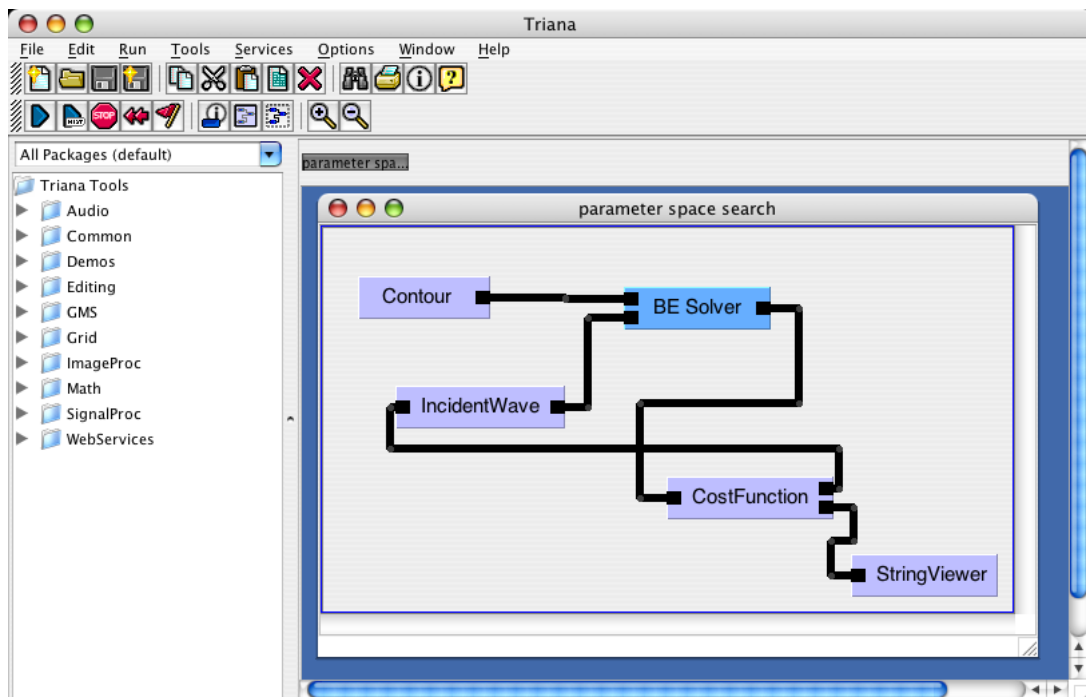


Figure 2: Parameter Space Search Workflow

The workflow illustrated in the screen shot figure 2 is the workflow that performs the experiment. The boundary element solver is run by instantiating a *Job* tool with the GridMonSteer extensions and adding the details for the Fortran executable. We specify the executable name and the input and output parameter files in the interface for the *Job* component.

In addition to the standard *Job* component we have built three specific tools to make up this workflow:

Contour is a tool that loads from file or creates the two dimensional contour file that is the first input to the solver. The output from the tool is the contour file and that is passed to the *Job* component which in turn will use GridMonSteer to save that file to our specified location on the remote resource. For this example we will use the same contour file for each iteration but we could modify that as a different optimisation.

IncidentWave is the tool that generates the file containing the four wave parameters which can be seen in its simple user interface in figure 3. Like *Contour* its output is a file that gets passed to the solver job via GridMonSteer. It also has a single input parameter which is the feedback loop from the *CostFunction* tool which contains the modified input parameters to feed back into the execution on the subsequent iteration.

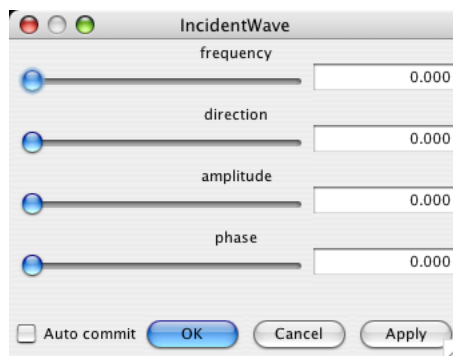


Figure 3: *IncidentWave* Tool User Interface

CostFunction is a tool that evaluates the solution from the solver and is responsible for making a decision on whether to halt the iteration of the workflow by sending output to its halting output, in this case a message displayed in a string viewer component, or whether to modify the input parameters and pass them back into the workflow for another iteration. The cost function in this component is not relevant to this simple example, here it is just a statistical average over the output data set. However, in a real world experiment this cost function would be carefully designed, it could be anything from a simple arithmetic calculation of the fitness of the output to something as complicated as another piece of legacy code run under GridMonSteer.

Upon running this workflow it will continue to execute until the halting condition for the *CostFunction* component has been met. Once finished the final output is displayed in the string viewer component.

4 Conclusion

We have outlined in this document a specific type of *remote steering* for applications where there are no steering capabilities engineered in to the application itself. This is the common case for legacy applications such as the boundary element example discussed in section 3.1. Although codes such as this may be old fashioned when compared to current computational codes written specifically for the grid, they are still often extremely useful and well used. Running codes

such as this on the grid is relatively straight forward as long as the correct input files are in place prior to execution and the output files are collected post execution. We have shown through the use of the GridMonSteer application wrapper that it is possible to integrate these legacy applications easily within a workflow tool such as Triana. In addition through the use of some simple additional tools in Triana we have implemented a remote steering scenario for performing parameter space searches or optimisations. This scenario is a commonly used one in computational science and engineering and could be easily extended to include more complex applications and supporting tools.

References

- [1] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor, “Enabling Applications on the Grid: A GridLab Overview,” *International Journal of High Performance Computing Applications: Special Issue on Grid Computing: Infrastructure and Applications*, vol. 17, pp. 449–466, November 2003.
- [2] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, “Condor-G: A Computation Management Agent for Multi-Institutional Grids,” in *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPCD-’01)*, 2001.
- [3] I. Foster and C. Kesselman, “Globus: A Metacomputing Infrastructure Toolkit,” *Int. Journal of Supercomputing Applications*, vol. 11, no. 2, pp. 115–128, 1997.
- [4] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, “A Resource Management Architecture for Metacomputing Systems,” in *Proc. IPPS/SPDP ’98 Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 62–82, 1998.