Scalable Computing: Practice and Experience
Volume 7, Number 3, pp. 75–86. http://www.scpe.org
ISSN 1895-1767
© 2006 SWPS

WEBCOM-G AND MPICH-G2 JOBS*

PADRAIG J. O'DOWD, ADARSH PATIL AND JOHN P. MORRISON†

Abstract. This paper discusses using WebCom-G to handle the management & scheduling of MPICH-G2 (MPI) jobs. Users can submit their MPI applications to a WebCom-G portal via a web interface. WebCom-G will then select the machines to execute the application on, depending on the machines available to it and the number of machines requested by the user. WebCom-G automatically & dynamically constructs a RSL script with the selected machines and schedules the job for execution on these machines. Once the MPI application has finished executing, results are stored on the portal server, where the user can collect them. A main advantage of this system is fault survival, if any of the machines fail during the execution of a job, WebCom-G can automatically handle such failures. Following a machine failure, WebCom-G can create a new RSL script with the failed machines removed, incorporate new machines (if they are available) to replace the failed ones and re-launch the job without any intervention from the user. The probability of failures in a Grid environment is high, so fault survival becomes an important issue.

Key words. WebCom-G, Globus, MPICH-G2, MPI, Grid Portals, Scheduling and Fault Survival

1. Introduction. The Globus Toolkit [1] has become the de-facto software for building grid computing environments and MPICH-G2 [2] uses Globus to provide a grid-enabled implementation of the MPI v1.1 standard. Being able to run unmodified legacy MPI code in this manner, strengthens the usability of grid computing for end users. However, there are some disadvantages/limitations to running MPICH-G2 jobs with RSL scripts (e.g., little or no support for job recovery after failure) and these will be discussed in detail in Section 3. This work investigates using WebCom-G to address some of these issues, such as automating the deployment, execution & fault survival of MPICH-G2 jobs [3, 4]. Other work [5] has investigated the use of WebCom-G for handling issues like fault survival with MPICH and running MPI jobs in a Beowulf cluster environment.

The remainder of this paper is organised as follows: Globus and its Execution Platform is discussed in Section 2. MPICH-G2 and how it uses Globus is described in Section 3. In Section 4, the WebCom-G Grid Operating System is presented, including an in-depth look at the WebCom-G components used in this work. In Section 5, the control of MPICH-G2 by WebCom-G is discussed and how WebCom-G can ensure the fault survival of MPICH-G2 jobs. In Section 6, some sample executions and results are presented to verify that the system operates correctly. Finally, Section 7 presents conclusions of this work.

2. Globus and its Execution Platform. Globus [1] provides the basic software infrastructure to build and maintain Grids. As dynamic networked resources are widely spread across the world, information services play a vital role in providing grid software infrastructures, discovering and monitoring resources for planning, developing and adopting applications. The onus is on the Information services to support resource & service discovery and subsequently use these resources and invoke services. Thus the Information services is a crucial part of any Grid.

An organisation running Globus hosts their resources in the Grid Information Service (GIS), running on a Gatekeeper machine. The information provided by the GIS may vary over time in an organisation. The information provider for a computational resource might provide static information (such as the number of nodes, amount of memory, operating system version number) and dynamic information such as the resources uncovered by the GIS, machine loads, storage and network information. Machines running Globus may use a simple scheduler or more advanced schedulers provided by Condor, LSF, PBS and Sun Grid Engine.

Typically users submit jobs to Globus (2.4) by means of a Resource Specification Language (RSL) script executing on the Gatekeeper, provided they have been successfully authenticated by the Grid Security Infrastructure(GSI). The RSL script specifies the application to run and the physical node(s) that the application should be executed on and any other required information. The Gatekeeper contacts a job manager service which in turn decides where the application is to be executed. For distributed services, the job manager negotiates with the Dynamically Updated Request Online Co-allocator (DUROC) to find the location of the requested service. DUROC makes the decision of where the application is to be executed by communicating with each machines' lower level Grid Resource Allocation Manager (GRAM). This information is communicated back to the job manager and it schedules the execution of the application according to its own policies. If no job manager

*The support of Science Foundation Ireland and Cosmogrid is gratefully acknowledged.
†Computer Science Dept., University College Cork, Ireland. ({p.odowd, adarsh, j.morrison}@cs.ucc.ie).

is specified, then the default service is used. This is usually the "fork" command, which returns immediately. A typical grid configuration is shown in Fig. 2.1.

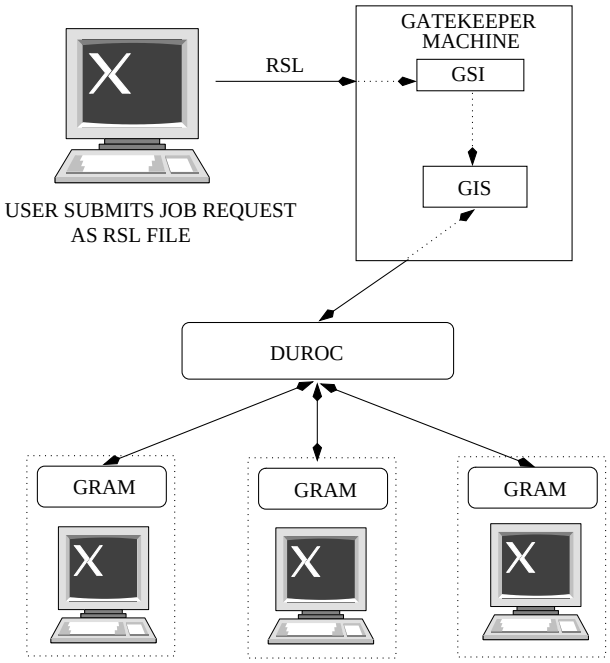

FIG. 2.1. *Globus 2.4 Execution model. A user generates an RSL script and submits it to the Gatekeeper. The Gatekeeper, in conjunction with DUROC and GRAM facilitate the distributed execution of requested services on the underlying nodes*

There are some disadvantages to using RSL scripts. Most notably, in a distributed execution if any node fails, the whole job fails and will have to be re-submitted by the user at a later time. There is no diagnostic information available to determine the cause of failure. Only resources known at execution time may be employed. There is no mechanism to facilitate job-resource dependencies. The resource must be available before the job is run, otherwise it fails. There is no in-built check-pointing support, although some can be programmatically included. This may not be feasible due to the particular grid configuration used. Also, RSL is only suited to tightly coupled nodes, with permanent availability. If any of the required nodes are off-line, the job will fail.

3. MPICH-G2. MPICH-G2 [2] is a grid-enabled implementation of the MPI v1.1 standard. It uses services from the Globus Toolkit to handle authentication, authorisation, executable staging, process creation, process monitoring, process control, communication, redirecting of standard input (& output) and remote file access. As a result a user can run MPI programs across multiple computers at different sites using the same commands that would be used on a parallel computer or cluster. MPICH-G2 allows users to couple multiple machines, potentially of different architectures, to run MPI applications. It automatically converts data in messages sent between machines of different architectures and supports multi-protocol communication by automatically selecting TCP for inter-machine messaging and (where available) vendor-supplied MPI for intra-machine messaging. According to [2], performance studies have shown that overheads relative to native implementations of basic communication functions are negligible.

As shown in Fig. 3.1, MPICH-G2 uses a range of Globus Toolkit services to address the various complex situations that arise in heterogeneous Grid environments. MPICH-G2 was created by creating a 'globus2' device for MPICH [6]. MPICH supports portability through its layered architecture. At the top is the MPI layer as defined by the MPI standards. Directly underneath this layer is the MPICH layer, which implements the MPI interface. Most of the code in an MPI implementation is independent of the underlying network communication system. This code, which includes error checking and various manipulations of the opaque objects, is implemented at the MPICH layer. All other functionality is passed to lower layers by means of the Abstract Device Interface (ADI). The ADI is a simpler interface than MPI proper and focuses on moving data between the MPI layer and the network subsystem. Those wishing to port MPI to a particular platform need

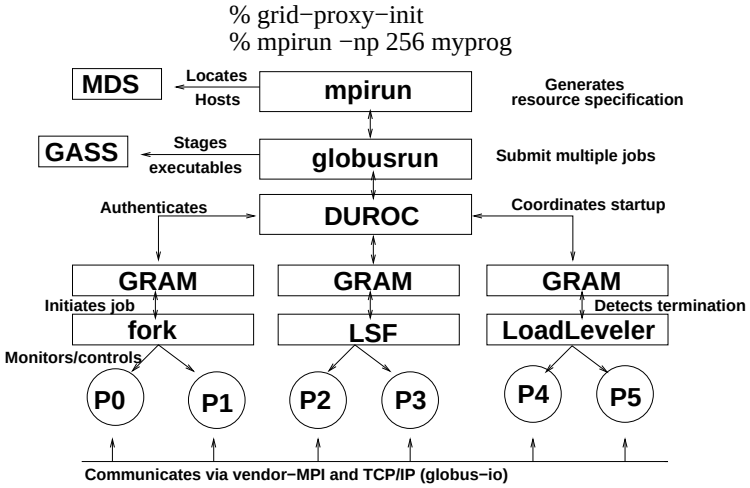

FIG. 3.1. *Overview of the startup of MPICH-G2 and the use of various Globus 2.4 Toolkit components to hide and manage heterogeneity. (This diagram was taken from [2])*

only define the routines in the ADI in order to obtain a full implementation. This is how MPICH-G2 works, by creating a special ADI device that uses the Globus Toolkit.

4. WebCom-G. WebCom-G [7, 8] is a Grid middleware that seeks to act as a "Grid Operating System". Unlike other Grid middlewares, WebCom-G (through it's use of the CG Model [9]) plans to hide the underlying complexity of a Grid infrastructure from application developers and end users. With WebCom-G, applications can be easily built as simple CG workflows and submitted to to a WebCom-G Grid for execution. WebCom-G also provides interoperability with existing middlewares like Globus, DCOM, Corba and EJB. CG workflows can be made to target services created with different middlewares. This section presents an overview of WebCom-G and the advantages it provides to application developers and end users.

The WebCom-G Grid Operating System is based on a pluggable module architecture and constructed around a WebCom kernel [10, 11, 12, 13]. It offers many features suited to Grid Computing. Before the WebCom-G project began, much research had already been carried out in the development of the WebCom meta-computer and the WebCom-G project was able to leverage this work. WebCom-G can be seen as a "Grid-Enabled" implementation of WebCom, where new features have been added to WebCom to allow it operate in Grid Environments.

Architecture of WebCom-G—A Modular Base Design. WebCom-G is an abstract machine architecture [7, 8, 10] consisting of a number of modules and a central component called the Backplane (see Fig. 4.1).

The Backplane component forms the basis of a WebCom-G machine and is used to load required modules. A basic WebCom-G system consists of the backplane and a set of five core modules:

1. Processing Module
2. Communications Module
3. Fault Tolerance Module
4. Load Balancing Module
5. Security Manager Module

Though users are free to implement there own Processing Modules, in this paper, the Processing Module refers to a Condensed Graph Engine [9]. A detailed description of all these modules is beyond the scope of this paper, a more thorough discussion of them found in the referenced papers.

To allow easy adaption to specific applications, the Backplane component allows the dynamic loading of different modules. The pluggable nature of each WebCom-G module provides great flexibility in the development of new modules and adapting WebCom-G to new application areas. The Backplane acts as a boot-strapper that co-ordinates the activities and communication between modules via a well defined interface. Inter-module communication is carried out by a WebCom-G messaging system through the Backplane. The Backplane inspects messages and determines whether they should be routed to local modules or to other WebCom-G machines, via the current communication manager. This mechanism provides great flexibility especially as "the

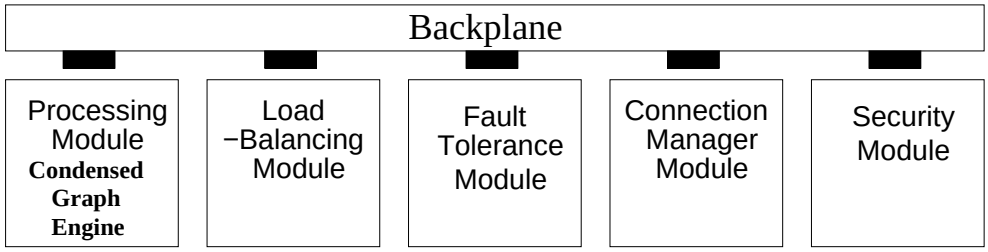

FIG. 4.1. *A minimal WebCom-G installation consists of a Backplane component, a Communication Manager Module and a number of module stubs for Processing, Fault Tolerance, Load Balancing and Security*

mechanism can be applied transparently across the network: transforming the metacomputer into a collection of networked modules". This allows an arbitrary module to request information from local modules or modules installed on different WebCom-G machines. For example, when making load balancing decisions, a server's Load Balancing Module can request information from the Load Balancing Module of each of its clients.

To facilitate the development of new applications and the integration of existing applications into WebCom-G, a number of programming and development tools were developed to allow the easy creation of Condensed Graphs. These tools include

1. APIs for languages like Java and C++.
2. An XML Schema for expressing Condensed Graphs.
3. Visual Tools that allow users to create Condensed Graph applications graphically e.g., WebCom-G IDE
4. High Level Language compilers that compile existing applications (written in languages like Java) directly into Condensed Graphs. These compilers can extract parallelism from sequentially written applications and have the advantage of not requiring application developers to know anything about the Condensed Graph Model of Computing.

4.1. WebCom-G IDE. The WebCom-G Integrated Development Environment provides a graphical method for creating Condensed Graphs. From within the IDE a user can create, load, save and execute Condensed Graph applications. A palette of nodes is supplied; these can be dragged onto the canvas, and linked together to form the graph. Fig. 4.2 shows an example graph to calculate the factorial of a number, which was developed with the WebCom-G IDE.

The IDE has the advantage of allowing the rapid development of Condensed Graphs compared to using the CG construction APIs or writing a graph manually in XML. When a Condensed Graph is created in the IDE and is being output, it is converted to XML which can be stored on disk (& subsequently reloaded by the IDE) or submitted directly to a running WebCom-G machine for execution from the IDE.

The palette on the IDE by default contains a set of core nodes for the construction of Condensed Graphs, but users can also create their own nodes which can be stored in a *Node Database* and loaded by the IDE and displayed on the palette. Also another very important feature of the IDE is that it can load services from the Interrogator Database and display these services on the palette as well. The *Interrogator Database* will be discussed in Section 4.2; it's basically a database of all the available services running on all the machines making up the current WebCom-G Grid. At specified intervals, WebCom-G machines can run their Interrogators, which register all available services on these machines with the Interrogator Database.

With all these services displayed on the palette, users can easily create distributed applications from the IDE that incorporate many different technologies, which then can be submitted to a WebCom-G Grid for execution, where the WebCom-G properties like load-balancing, fault tolerance and security can be taken advantage of.

4.2. Default WebCom-G LB Module & Interrogators. The Load Balancing module of a WebCom-G machine decides where instructions (tasks) are to be executed. (In WebCom-G, tasks are referred to as instructions—but note the term instructions can mean instructions of various grain sizes. So in WebCom-G, a large sequential program is referred to as an instruction.) Different Load Balancing modules use different strategies when scheduling instructions for execution. They can take into account issues like security restrictions (by consulting with a Security Manager module), specialized resource locations and access reliability. In addition to these type of considerations, different Load Balancing modules can be employed to implement specific load balancing algorithms.

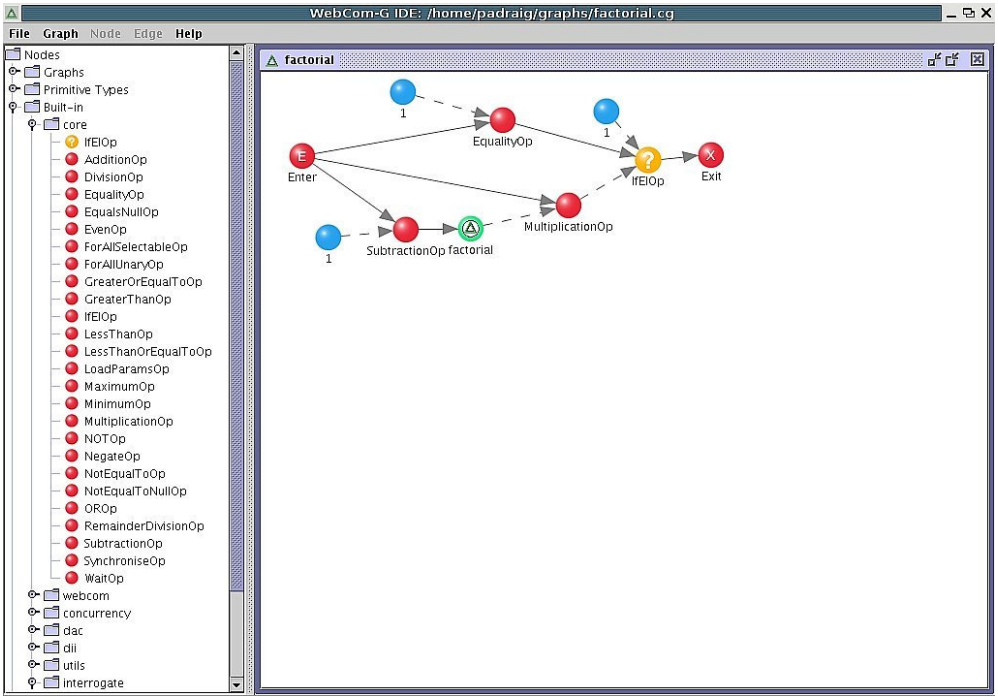

FIG. 4.2. *An example graph to calculate the factorial of a number, which was constructed in the WebCom-G IDE*

Though WebCom-G Load Balancers can take on many forms, and users are free to implement their own load balancing strategies, one very important Load Balancer module is the Default WebCom-G LB module. The default WebCom-G Load Balancer (which employs its own targeting mechanism), allows service invocations to be targeted at specific machines that provide those particular services. Some of the main components of this Load balancer are the Interrogators.

1. **Services:** When the term *"service"* is used in this paper it can cover a broad range of software types, some of these are listed below:
   (a) Machine dependent users applications and licensed software i. e., Matlab or just standard user applications installed on a machine.
   (b) CORBA, EJB and DCOM services.
   (c) Web Services.
   (d) Globus Web Services and other Grid services.
   The above list only mentions some of the possibilities and users can easily create their own execution plug-ins, that allow new service types to be handled.
2. **Interrogators:** An *Interrogator* is a component that can interrogate a WebCom-G machine to see if and what services the machine is running. Different Interrogators have to be created for different service types .i.e an CORBA Interrogator for CORBA, a EJB Interrogator for EJB etc. Once an Interrogator gathers a list of services present on a particular machine, it can register these services with an Interrogator Database. This Interrogator Database will contain a list of all the services running on all the machines making up the current WebCom-G Grid. This Interrogator Database can then be consulted by WebCom-G Load Balancer modules to see where certain services can be executed. Also the Interrogator Database can be used by the WebCom-G IDE, see Section 4.1.

When WebCom-G is running on a machine, it can run it's own interrogators to see what services the machine is running and then register these services with the Interrogator Database. Any WebCom-G machine in general can use this Interrogator Database for scheduling issues e.g., to see what machines are running a certain service. A top-level WebCom-G machine can request all it's client WebCom-G machines (and their clients, if they have any) to run their interrogators by simply executing a particular Condensed Graph application. This Condensed Graph is recursive, in that when it executes on a machine it invokes the interrogators on that machine and passes an instance of its self onto all the clients of the machine for execution. So every

client WebCom-G machine connected to the tree, registers all they're available services with the Interrogator Database.

When the default WebCom-G Load Balancer is asked to schedule a service (or standard instruction) that is not in the Interrogator Database, it can do two possible things depending on the rules that were set by the user—it can either schedule the service/instruction for execution on a remote client (using a round-robin scheduling policy) or execute the instruction locally.

4.3. WebCom-G Portal Server. As well as using the IDE and web services to submit jobs, a WebCom-G (Web) Portal Server can also be used to access a WebCom-G Grid (e.g., `http://portal.webcom-g.org`). In order to avoid a single point of failure, a number of Portals per Grid can be maintained. The WebCom-G Portal Server allows users to access a WebCom-G Grid, once a Web Browser is installed on their machine. When a user is granted access to a WebCom-G Portal, they can perform the following activates:

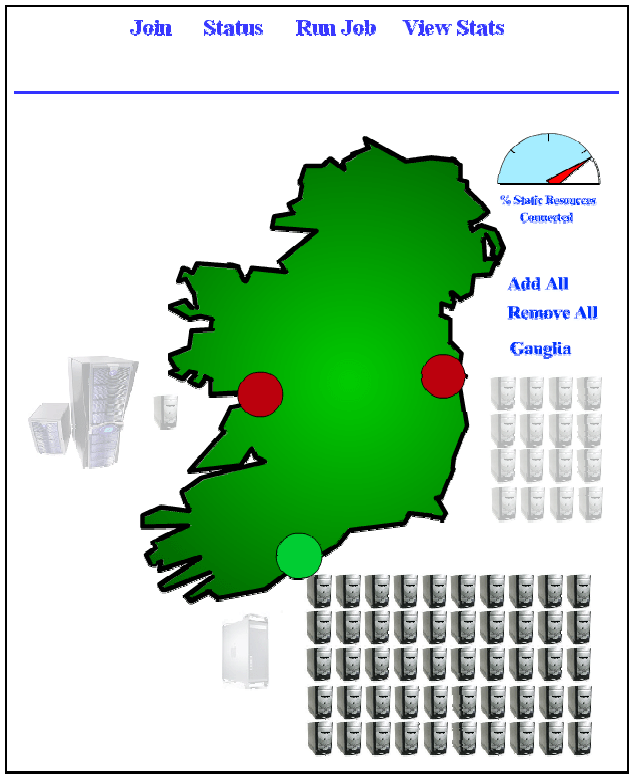

FIG. 4.3. *Graphical view of the resources connected to the WebCom-G Portal*

1. Donate the idle compute cycles of their PC to the WebCom-G Grid
2. Submit jobs to the WebCom-G Grid for execution—jobs can range from simple sequential programs to complicated parallel/distributed Condensed Graph applications. Results from executed jobs can be stored on the server for collection by a user at a later time.
3. View the statistics of the resources in the current WebCom-G Grid (If security access allows). Statistics that can be viewed range from CPU load, memory usage, software packages installed on machines, current status of executing jobs etc. Various external Information Gathering Systems can be used by WebCom-G to make scheduling decisions (e.g., the use of Ganglia [14]) and a user can view this information on the Portal Server—see Fig. 4.4.

Adding proper security mechanisms to WebCom-G (& WebCom-G Grids in general) to guarantee the privacy of users' data etc, is still the focus of on going work and despite a lot done, a lot of work remains. Normally the WebCom-G daemon running on the WebCom-G Portal forms the head of the Grid (the root of the tree), though this isn't necessary always the case, parts of a WebCom-G Grid can form a peer-to-peer topology e.g., its possible to have a number of WebCom-G Portals connected to each other as peers and some client resources only visible to certain Portals, but if the need arises portals can donate some of their resources

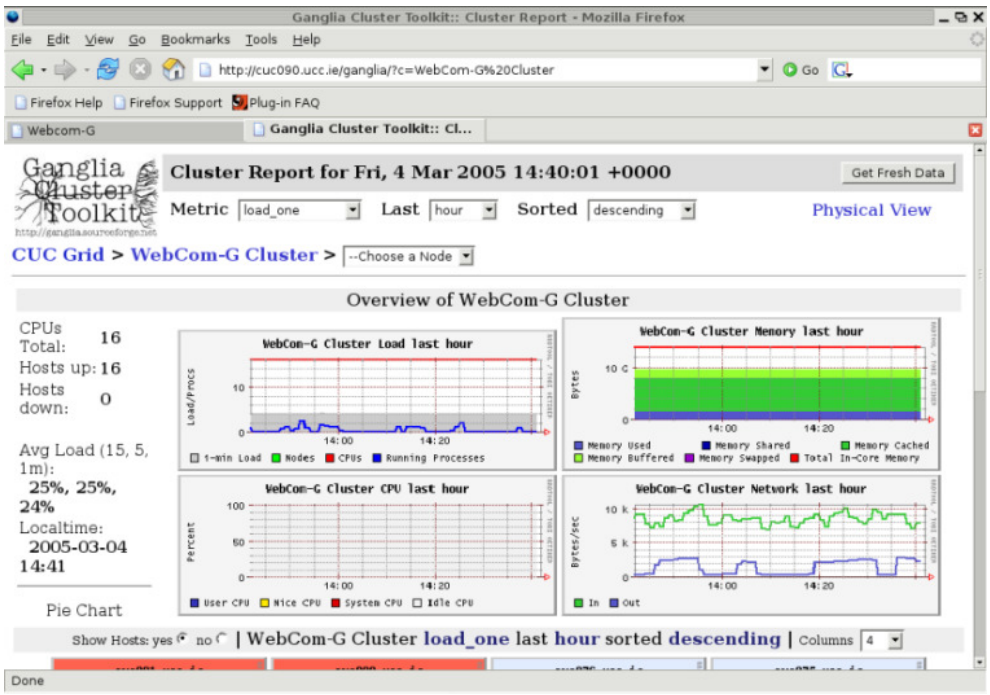

FIG. 4.4. *Information gathered by Ganglia on the resources attached to a WebCom-G Grid*

to other portals (if security permissions allow).

Fig. 4.4 shows the information gathered by Ganglia [14] from a number of resources connected to a WebCom-G Grid. This information can be used by WebCom-G to make scheduling decisions and can also be displayed to a user, who wants to see the current state of Grid resources.

5. Automating the Deployment, Execution & Fault Survival of MPICH-G2 jobs with WebCom-G. Authorised users can visit a WebCom-G Portal (see Section 4.3) and submit their applications, which are written in the form of Condensed Graphs. WebCom-G then schedules the execution of these Condensed Graphs across the available resources. As well as the end results, various statistics about each job and the current state of the underlying resources (see Fig. 5.1) are also maintained by the Portal.

The above notion was extended for allowing portal users to execute MPI (MPICH-G2) jobs on the WebCom-G Portal (and the WebCom-G Grid). Users can visit the WebCom-G Portal, upload their MPI code and specify the number of machines they require. WebCom-G firsts decides the machines on which the MPI application will execute. The MPI code is then compiled and executed on these machines. If any errors occur during the compilation, they are returned to the user and the process aborted. Finally the application is scheduled for execution on the underlying resources and the following tasks are performed:

- A Resource Specification Language (RSL) script is built on the fly, depending on machine availability, path & package availability.
- The application is run using the RSL file.
- In the case of failure, the application is rescheduled with a new RSL file, dynamically created by WebCom-G from a new interrogation of the underlying resources. WebCom-G then re-launches the job with the new RSL file. No user intervention is required at this point.
- Finally results are sent to the Portal, for collection by the user.

5.1. Overview of the MPICH-G2 CG Application. This section presents an overview of the MPICH-G2 Condensed Graph application that was developed, in order to use WebCom-G to manage MPICH-G2 jobs (i.e. handling issues like fault survival etc).

Fig. 5.2 shows the MPICH-G2 Condensed Graph application that was created with the IDE. The following presumptions were made about the graph based on the Globus test-bed that was used in its development. All machines taking part used a shared file-system, and before the graph is executed the source will have been

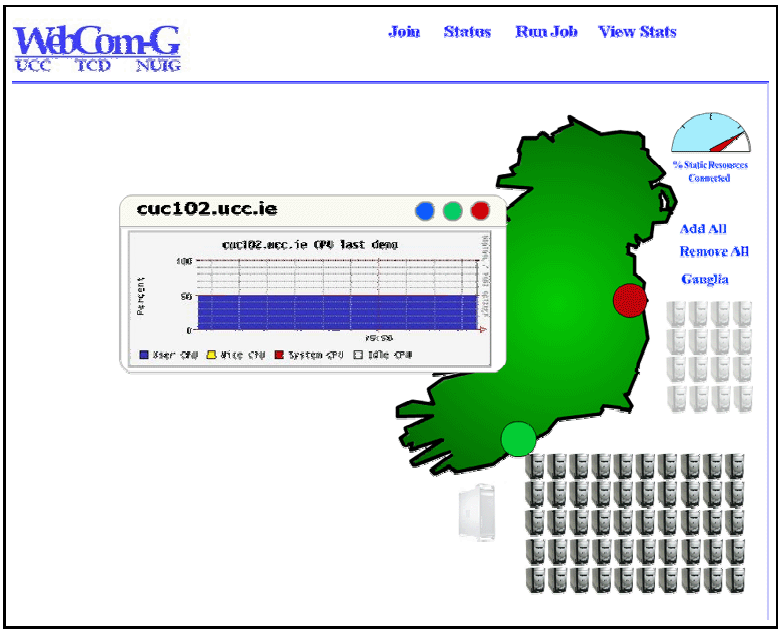

FIG. 5.1. *Zooming in on the current state of a particular machine/resource in a WebCom-G Grid*

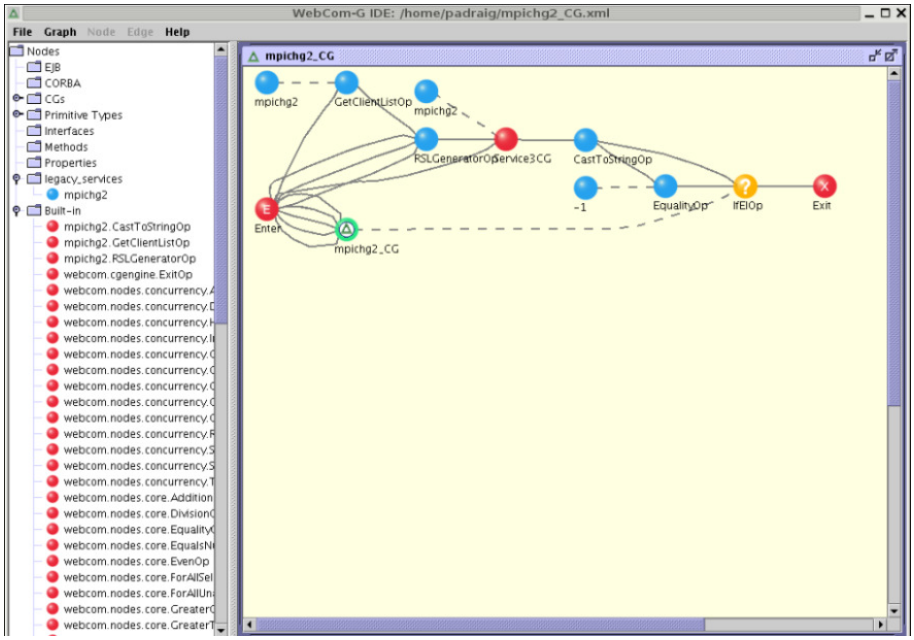

FIG. 5.2. *The MPICH-G2 Condensed Graph Application, developed in the WebCom-G IDE*

compiled (and will have to have compiled without errors, otherwise the process is aborted and the user is notified.) Though the graph can easily be changed to account for machines that don't use a shared file system. (Whether there is a shared file-system present or not, this has no affect on testing whether the fault survival mechanism of WebCom-G works.)

The graph takes five inputs:

- Number of machines required.
- MPI source code.
- Directory to run the MPICH-G2 job from.
- Arguments to pass to the MPICH-G2 job.

- Timeout: Users (if they want) are allowed to specify a timeout for their MPICH-G2 job, if the execution time of a job exceeds the timeout, the job is thought to have failed and is rescheduled.

Overview of the main nodes of the graph (see Fig. 5.2) and how it executes:

- *GetClientListOp*—This node takes two parameters, the name of a service, which in the case of this graph is mpichg2 and the number of machines required. The node when executed, outputs a list of machine names (no greater then the number of required machines). This node uses the current list of available client machines connected to the (WebCom-G) Portal and the Interrogator Database to see which of these clients can execute MPICH-G2 jobs and then creates a list of machines that can be used in the execution of the MPICH-G2 job.
- *RSLGeneratorOp*—This node generates an RSL file which can be used to execute the MPICH-G2 job. It takes the list of machine names generated by the GetClientListOp node, the name of the executable. the directory to run the MPICH-G2 job from and the arguments to pass to the job. From these, this node generates the RSL Script.
- *Service3CG*—This node executes the MPICH-G2 job with the generated RSL Script and the timeout if specified by the user. This node then monitors the execution of the job and in the event of a job failure/crashing, it detects the job failure and reports it. This node can detect failures of MPICH-G2 jobs in the following ways:
  - *mpirun returns and data has been written to the "stderr"*: There are a few bugs in the current release of MPICH-G2 (for details, readers are referred to the MPICH-G2 web site at `www.hpclab.niu.edu/mpi/`) that affect the detection of failures from the mpirun command. One of these bugs is that "the exit code passed to MPI_Abort does not get propagated back to mpirun". Another is that sometimes "when calling MPI_Abort, "stdout/stderr" are not always flushed unless the user explicitly flushes (fflush) both prior to calling MPI_Abort, and even then, the data is sent to stdout/stderr of the other processes". These bugs are due to be fixed in future releases of MPICH-G2. So as the Service3CG can not get the exit code passed back by MPI_Abort to mpirun, it just monitors the "stderr", if any data is written to "stderr" the node considers the job to have failed. When users are submitting MPI jobs to the WebCom-G Portal, they are expected to be aware of these issues. When future releases of MPICH-G2 are released, these problems can be removed.
  - *A WebCom-G client (that is in the RSL script) fails*: Users (if they want) can specify that a job is considered to have failed if a WebCom-G machine which was included in the RSL script fails during the execution of the MPICH-G2 job. So when a WebCom-G client fails, the Service3CG forces the job to terminate and reports the job as failed. One potential problem with this type of fault detection is that, it could be possible that the WebCom-G client that failed might have finished actively taking part in the job and so even despite its failure, there is no reason to reschedule the job.
  - *Timeout*: Users (if they want) are allowed to specify a timeout for their MPICH-G2 job, if the execution time of a job exceeds the timeout, the job is thought to have failed and is rescheduled. This timeout is only supplied as an option and is not ideally suited to a Grid environment, as resources in the grid are heterogeneous in nature. So executing an MPICH-G2 job on different machines (of different performance levels), obviously can cause the execution time of the job to vary greatly between runs. It's presumed, if a user specifies a timeout that they are thinking of a worst case scenario.
- *IfElOp*—When this node fires, if the MPICH-G2 job failed it will evaluate to true, in which case the *MPICHG2_CG* node fires, this node is really just a recursive call to the graph itself and causes it to execute again. (When the graph is re-executed any machines that have failed with not be in the list generated by the *GetClientListOp* node. Users can set in a properties files to have a re-integration of the underlying resources again in case any new resources have been added since the previous execution, though any new resources that join the Portal, usually perform interrogation when they join.) At the moment, the job will only be re-executed five times, after that a message is returned to the user informing them that the job failed five times in a row. It is then up to the user, whether they want to re-submit the job straight away or wait a certain amount of time. Otherwise the job will have executed correctly and the graph exits and returns the results to the Portal for collection by the user.

6. Sample Executions and Results. A simple application was developed to demonstrate the system in action (using WebCom-G to manage MPICH-G2 jobs)—the application simply counts all the prime numbers up to a certain number (1 million) and returns the result. The number partition (1 to 1 million) is divided

evenly among the available machines—obviously with this type of application, its execution time decreases as more machines are added. No complex algorithms were used just a simple brute force check, as the goal was to highlight WebCom-G's management of MPICH-G2 jobs not the efficiency of the MPI program.

Our test-bed consisted of six Debian machines that had Globus, MPICH-G2 and WebCom-G installed. So when the Portal was scheduling MPICH-G2 jobs it could use these six machines.

The following executions were performed and their results recorded:

- Simulation 1: Execute the application on five machines (no failures).
- Simulation 2: Execute the application on four machines (no failures).
- Simulation 3: Execute the application on five machines & during it's execution one controlled machine failure occurred, causing WebCom-G to reschedule the job
- Simulation 4: Execute the application on five machines & during it's execution two controlled machines failures occurred, causing WebCom-G to reschedule the job

Note: As there are only six machines in the test-bed, if a user wants five machines but two fail then obviously the application can only be re-run with four. This is because our test-bed had just six machines, it's presumed that in a true Grid environment, more machines would be available then required by the user and if one of their allocated machines fails, a new one can be obtained in its place.

The *'Total Execution Time'* of the application includes the time from when 'mpirun' was called to when it returns. The *'Individual Execution Time'* includes the time each machine spent executing its own prime count on the number partition that was assigned to it. In the case of a controlled machine failure this time indicates the time from when it started until it failed. (Time is shown in seconds.) As the number space (1 to 1 million) is divided into number segments, the segments with smaller numbers with execute much faster then the segments with larger numbers. For example in the following experiments Machine 1 gets smaller numbers then Machine 5, so Machine 1 checks its assigned number space alot faster then Machine 5. Also its worth noting that in Simulation 3 and 4, Machine 1 finished executing before the failures, but when the application was re-launched Machine 1 re-did the work it had already completed - this is because the current system guarantees only fault survival. If the job fails before fully finishing, it is completely re-launched.

Simulation 1 : Execute the application on five machines (no failures).

- Total execution time: 232
- Individual execution time for each machine:
  - Machine 1: 28.3
  - Machine 2: 78.2
  - Machine 3: 123.5
  - Machine 4: 171.2
  - Machine 5: 207.9

Simulation 2 : Execute the application on four machines (no failures).

- Total execution time: 281
- Individual execution time for each machine:
  - Machine 1: 43.4
  - Machine 2: 120.5
  - Machine 3: 188.6
  - Machine 4: 262.5

Simulation 3 : Execute the application on five machines & during it's execution one controlled machine failure occurred, causing WebCom-G to reschedule the job.

- Total execution time: 355
- Individual execution time for each machine -run 1:
  - Machine 1: 28.4
  - Machine 2: 78.6
  - Machine 3: 100
  - Machine 4: 100
  - Machine 5: (failed at) 100
- Individual execution time for each machine -run 2:
  - Machine 1: 28.2
  - Machine 2: 78.5

  - Machine 3: 123.3
  - Machine 4: 171.2
  - Machine 5: (new machine) 207.3

Simulation 4 : Execute the application on five machines & during it's execution two controlled machines failures occurred, causing WebCom-G to reschedule the job.

- Total execution time: 411
- Individual execution time for each machine -run 1:
  - Machine 1: 28.3
  - Machine 2: 78.1
  - Machine 3: 100
  - Machine 4: (failed at) 100
  - Machine 5: (failed at) 100
- Individual execution time for each machine -run 2:
  - Machine 1: 43.4
  - Machine 2: 118.3
  - Machine 3: 188.3
  - Machine 4: (new machine) 256.4

(With the two machines failures, only four machines left)

Summary. Fig. 6.1 shows the total execution times of the different simulations. Due to the simplicity of the application these results are, as would be expected:

- Simulation 1 takes 232 seconds to execute with five machines.
- Simulation 2 takes 281 seconds to execute with four machines, obviously taking longer to execute then Simulation 1.

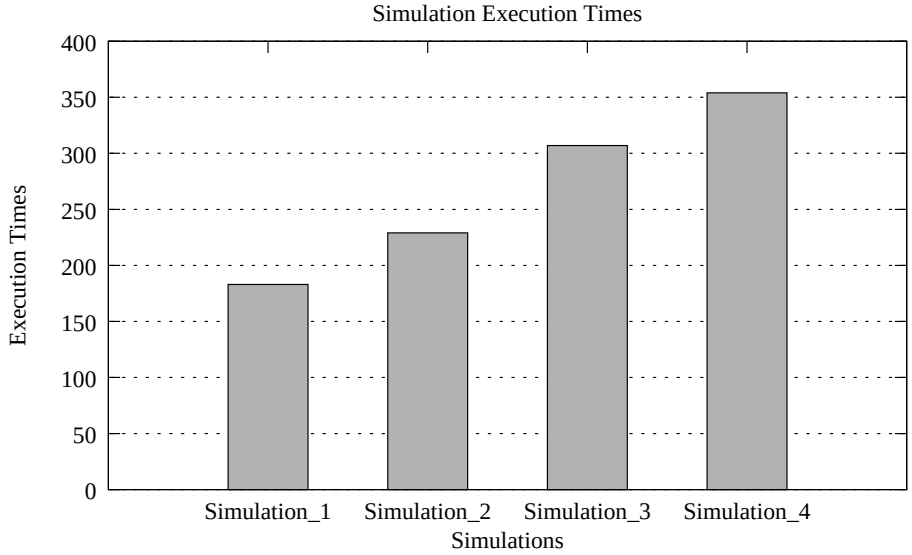

FIG. 6.1. *Total execution times of the different simulations*

- Simulation 3 takes 355 seconds to execute, because during its first execution at time 100, machine five fails, causing the whole job to fail and be rescheduled. During its second execution the application completes fully. So the total execution time for this simulation is roughly around: 100 (time in the first execution when the job failed) + 232 (the total execution time from Simulation 1) + 23 (the overhead of running the graph in WebCom-G and the start-up of an MPICH-G2 job—the time 100 refers only to the time the program spent executing the prime count code, it does not include the startup time for the MPICH-G2 job).
- Simulation 4 takes 411 seconds to execute, because during its first execution at time 100, two machines (four and five) fail, causing the whole job to fail and be rescheduled. During its second execution

the application completes fully (but with only four machines, as there were only six machines in our test-bed and two had failed). So the total execution time for this simulation is roughly around: 100 (time in the first execution will the job failed) + 281 (the total execution time from Simulation 2) + 25 (the overhead of running the graph in WebCom-G and the start-up of an MPICH-G2 job—the time 100 refers only to the time the program spent executing the prime count code, it does not include the startup time for the MPICH-G2 job).

7. Conclusions. In this paper, it was discussed how WebCom-G (& the CG Model) can be used to automate the deployment, execution & fault survival of MPICH-G2 jobs. The main purpose of this research has been to focus on fault survival issues related to MPICH-G2 jobs and allow users to execute their MPI jobs in grid environments, through the use of simple easy to use web interfaces. If a job submitted to the grid fails, the whole job will have to be re-submitted. However, the benefits of using WebCom-G for scheduling these jobs are immediately perceptible. If a job fails WebCom-G's fault survival will ensure that the job is automatically rescheduled. This may be to the same grid, or a different grid with the required resources. This dynamic scheduling of grid operations is possible by delaying the creation of the RSL script until just before it is required. Once the RSL script has been created, the physical configuration is determined.

REFERENCES

[1] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[2] Nicholas T. Karohis. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. citeseer.ist.psu.edu/554400.html.

[3] Padraig J. O'Dowd, Adarsh Patil, and John P. Morrison. Managing MPICH-G2 Jobs with WebCom-G. In *Proceedings of the 4th International Symposium on Parallel and Distributed Computing (ISPDC)*, Lille, France, July 2005.

[4] Adarsh Patil, Padraig J. O'Dowd, and John P. Morrison. Automating the Deployment, Execution & Fault Survival of MPICH-G2 jobs with WebCom-G. In *Proceedings of the 2005 Cluster Computing and Grid (CCGrid) Symposium*, Cardiff, UK, May 2005.

[5] Brian Clayton, Therese Enright, and John P. Morrison. Running MPI Jobs with WebCom. In *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, June 2005.

[6] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[7] John P. Morrison, Brian Clayton, David A. Power, and Adarsh Patil. WebCom-G: Grid Enabled Metacomputing. *The Journal of Neural, Parallel and Scientific Computation. Special Issue on Grid Computing.*, 2004(12)(2):419–438, April 2004.

[8] David A. Power, Adarsh Patil, Sunil John, and John P. Morrison. WebCom-G. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2003)*, Las Vegas, Nevada, June 2003.

[9] John P. Morrison. *Condensed Graphs: Unifying Availability-Driven, Coercion-Driven and Control-Driven Computing*. PhD thesis, Technische Universiteit Eindhoven, 1996.

[10] David A. Power. *A Framework for: Heterogeneous Metacomputing, Load Balancing and Programming in WebCom*. PhD thesis, University College Cork, 2004.

[11] James J. Kennedy. *Design and Implementation of an N-Tier Metacomputer with Decentralised Fault Tolerance*. PhD thesis, University College Cork, 2004.

[12] David A. Power. WebCom: A Metacomputer Incorporating Dynamic Client Promotion and Redirection. Master Thesis, University College Cork, 2000.

[13] John P. Morrison, James J. Kennedy, and David A. Power. WebCom: A Volunteer-Based Metacomputer. In *The Journal of Supercomputing, Vol. 18(1)*, pages 47–61, 2001.

[14] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. In *Parallel Computing, Vol. 30, Issue 7*, July 2004.

Edited by: M. Tudruj, R. Olejnik.
Received: February 24, 2006.
Accepted: July 30, 2006.