Alchemi: A .NET-based Enterprise Grid Computing System

Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal

Grid Computing and **D**istributed **S**ystems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
Email:{akshayl, raj, rranjan, srikumar}@cs.mu.oz.au

Abstract: *Computational grids that couple geographically distributed resources are becoming the de-facto computing platform for solving large-scale problems in science, engineering, and commerce. Software to enable grid computing has been primarily written for Unix-class operating systems, thus severely limiting the ability to effectively utilize the computing resources of the vast majority of Windows-based desktop computers. Addressing Windows-based grid computing is particularly important from the software industry's viewpoint where interest in grids is emerging rapidly. Microsoft's .NET Framework has become near-ubiquitous for implementing commercial distributed systems for Windows-based platforms, positioning it as the ideal platform for grid computing in this context. In this paper we present Alchemi[1], a .NET-based framework that provides the runtime machinery and programming environment required to construct enterprise/desktop grids and develop grid applications. It allows flexible application composition by supporting an object-oriented application programming model in addition to a file-based job model. Cross-platform support is provided via a web services interface and a flexible execution model supports dedicated and non-dedicated (voluntary) execution by grid nodes.*

1 Introduction

The idea of metacomputing [2] is very promising as it enables the use of a network of many independent computers as if they were one large parallel machine, or virtual supercomputer for solving large-scale problems in science, engineering, and commerce. With the exponential growth of global computer ownership, local networks and Internet connectivity, this concept has been taken to a global level – popularly called as grid computing [1][8]. This, coupled with the fact that desktop PCs (personal computers) in corporate and home environments are heavily underutilized – typically only one-tenth of processing power is used [32]– has given rise to interest in harnessing these underutilized resources (e.g., CPU cycles) of desktop PCs connected over the Internet. This new paradigm has been dubbed as Internet computing, which is also called by several different names including enterprise/desktop grid computing [16], peer-to-peer (P2P) computing [17], and public distributed computing.

There is rapidly emerging interest in grid computing from commercial enterprises. A Microsoft Windows-based grid computing infrastructure will play a critical role in the industry-wide adoption of grids [9][14][16][21] due to the large-scale deployment of Windows within enterprises. This enables the harnessing of the unused computational power of desktop PCs and workstations to create a virtual supercomputing resource at a fraction of the cost of traditional supercomputers. However, there is a distinct lack of service-oriented architecture-based grid computing software in this space. To overcome this limitation, we have developed a Windows-based grid computing framework called Alchemi implemented on the Microsoft .NET Platform.

While the notion of grid computing is simple enough, the practical realization of grids poses a number of challenges. Key issues that need to be dealt with are heterogeneity, reliability, application composition, scheduling, resource management and security [13]. The Microsoft .NET Framework [3] provides a powerful toolset that can be leveraged for all of these, in particular support for remote execution (via .NET Remoting [4] and web services [22]), multithreading, security, asynchronous programming, disconnected data access, managed execution and cross-language development, making it an ideal platform for grid computing middleware.

Alchemi was conceived with the aim of making grid construction and development of grid software as easy as possible without sacrificing flexibility, scalability, reliability and extensibility. The key features supported by Alchemi are:

[1] Alchemi Project is supported by an ARC Discovery Project and Melbourne University internal grants.

- Internet-based clustering [19][20] of heterogeneous desktop computers;
- dedicated or non-dedicated (voluntary) execution by individual nodes;
- object-oriented grid application programming model (fine-grained abstraction);
- file-based grid job model (coarse-grained abstraction) for grid-enabling legacy applications and
- web services interface supporting the job model for interoperability with custom grid middleware e.g. for creating a global, cross-platform grid environment via a custom resource broker component.

Alchemi has already been used in creating and deploying several science and commercial applications on enterprise Grids. They include: (a) **BLAST** (Basic Local Alignment Search Tool) used in identifying similarities between biological sequences, (b) Gridbus broker [33] that supports integration/utilisation of Alchemi-based enterprise grids as nodes within global grids, (c) CSIRO Australia's hydrology application for catchments modeling and simulation, (d) Microsoft Excel spreadsheet processing [35], (e) Satyam India's Microarray data analysis application that aids in early detection of breast cancer, and (f) high performance cryptography for encryption/decryption [34].

The rest of the paper is organized as follows. Section 2 presents the Alchemi architecture and discusses configurations for creating different grid environments. Section 3 discusses the system implementation and presents the lifecycle of an Alchemi-enabled grid application demonstrating its execution model. Section 4 presents the object-oriented grid thread programming model supported by the Alchemi API. Section 5 presents the results of an evaluation of Alchemi as a platform for execution of applications written using the Alchemi API. It also evaluates the use of Alchemi nodes as part of a global grid alongside Unix-class grid nodes running Globus software. Section 5 presents related works along with their comparison to Alchemi. Finally, we conclude the paper with work planned for the future.

2 Architecture

Alchemi's layered architecture for an enterprise grid computing environment is shown in Figure 1. Alchemi follows the master-worker parallel programming paradigm [29] in which a central component dispatches independent units of parallel execution to workers and manages them. In Alchemi, this unit of parallel execution is termed 'grid thread' and contains the instructions to be executed on a grid node, while the central component is termed 'Manager'.

A 'grid application' consists of a number of related grid threads. Grid applications and grid threads are exposed to the application developer as .NET classes / objects via the Alchemi .NET API. When an application written using this API is executed, grid thread objects are submitted to the Alchemi Manager for execution by the grid. Alternatively, file-based jobs (with related jobs comprising a task) can be created using an XML representation to grid-enable legacy applications for which precompiled executables exist. Jobs can be submitted via Alchemi Console Interface or Cross-Platform Manager web service interface, which in turn convert them into the grid threads before submitting then to the Manager for execution by the grid.

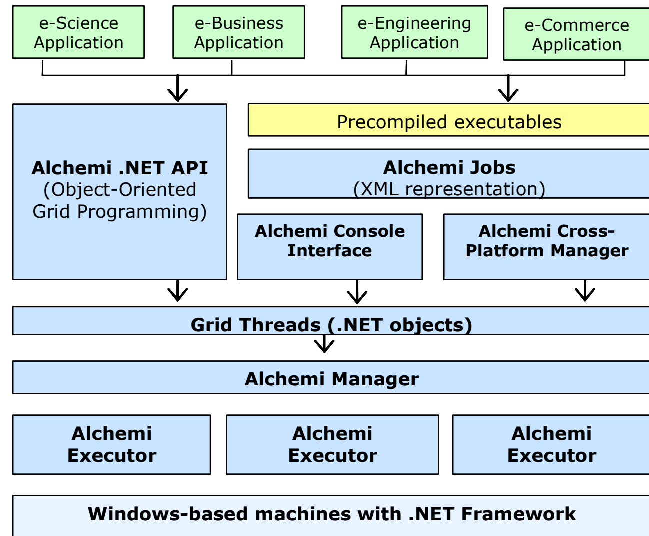

Figure 1. A layered architecture for an enterprise grid computing environment.

2.1 Application Models

Two models for parallel application composition are supported by Alchemi.

2.1.1 Grid Thread Model

Minimizing the entry barrier to writing applications for a grid environment is one of Alchemi's key goals. This goal is served by an object-oriented programming environment via the Alchemi .NET API which can be used to write grid applications in any .NET-supported language.

The atomic unit of independent parallel execution is a grid thread with many grid threads comprising a grid

application (hereafter, ‘applications’ and ‘threads’ can be taken to mean grid applications and grid threads respectively, unless stated otherwise). The two central classes in the Alchemi .NET API are `GThread` and `GApplication`, representing a grid thread and grid application respectively. There are essentially two parts to an Alchemi grid application. Each is centered on one of these classes:

- “Remote code”: code to be executed remotely i.e. on the grid (a grid thread and its dependencies) and
- “Local code”: code to be executed locally (code responsible for creating and executing grid threads).

A concrete grid thread is implemented by writing a class that derives from `GThread`, overriding the `void Start()` method, and marking the class with the `Serializable` attribute. Code to be executed remotely is defined in the implementation of the overridden `void Start()` method.

The application itself (local code) creates instances of the custom grid thread, executes them on the grid and consumes each thread’s results. It makes use of an instance of the `GApplication` class which represents a grid application. The modules (.EXE or .DLL files) containing the implementation of this `GThread`-derived class and any other dependency types that not part of the .NET Framework must be included in the `Manifest` of the `GApplication` instance. Instances of the `GThread`-derived class are asynchronously executed on the grid by adding them to the grid application. Upon completion of each thread, a ‘thread finish’ event is fired and a method subscribing to this event can consume the thread’s results. Other events such as ‘application finish’ and ‘thread failed’ can also be subscribed to. Thus, the programmatic abstraction of the grid in this manner described allows the application developer to concentrate on the application itself without worrying about "plumbing" details.

A sample list of applications created using Alchemi’s Grid thread programming model include: parallel Mandelbrot set generator, high performance encryption/decryption [34], and spreadsheet processing [35].

2.1.2 Grid Job Model

Traditional grid implementations have offered a high-level, abstraction of the "virtual machine", where the smallest unit of parallel execution is a process. In this model, a work unit is typically described by specifying a command, input files and output files. In Alchemi, such a work unit is termed ‘job’ with many jobs constituting a ‘task’.

Although writing software for the “grid job” model involves dealing with files, an approach that can be complicated and inflexible, Alchemi’s architecture supports it for the following reasons:

- grid-enabling existing applications; and
- interoperability with grid middleware that can leverage Alchemi via the Cross Platform Manager web service

Tasks and their constituent jobs are represented as XML files conforming to the Alchemi task and job schemas. Figure 2 shows a sample task representation that contains two jobs to execute the `Reverse.exe` program against two input files.

```
<task>
  <manifest>
    <embedded_file name="Reverse.exe"
location="Reverse.exe" />
  </manifest>

  <job id="0">
    <input>
      <embedded_file name="input1.txt"
location="input1.txt" />
    </input>
    <work run_command="Reverse.exe input1.txt >
result1.txt" />
    <output>
      <embedded_file name="result1.txt"/>
    </output>
  </job>

  <job id="1">
    <input>
      <embedded_file name="input2.txt"
location="input2.txt" />
    </input>
    <work run_command="Reverse input2.txt >
result2.txt" />
    <output>
      <embedded_file name="result2.txt"/>
    </output>
  </job>
</task>
```

Figure 2. Sample XML-based task representation.

Before submitting the task to the Manager, references to the ‘embedded’ files are resolved and the files themselves are embedded into the task XML file as Base64-encoded text data. When finished jobs are retrieved from the Manager, the Base64-encoded contents of the ‘embedded’ files are decoded and written to disk. It should be noted that tasks and jobs are represented internally as grid applications and grid threads respectively. Thus, any discussion that applies to ‘grid applications’ and ‘grid threads’ applies to ‘grid tasks’ and ‘grid jobs’ as well.

A sample list of applications created using Alchemi's Grid job model include: **BLAST** (Basic Local Alignment Search Tool) used in identifying similarities between biological sequences and Gridbus broker [33] that supports integration/utilisation of Alchemi-based enterprise grids as nodes within global grids.

2.2 Distributed Components

Four types of nodes (or hosts) take part in enterprise grid construction and application execution (see Figure 3). An Alchemi enterprise grid is constructed by deploying a Manager node and deploying one or more Executor nodes configured to connect to the Manager. One or more Users can execute their applications on the cluster by connecting to the Manager. An optional component, the Cross Platform Manager provides a web service interface to custom grid middleware. The operation of the Manager, Executor, User and Cross Platform Manager nodes is described below.

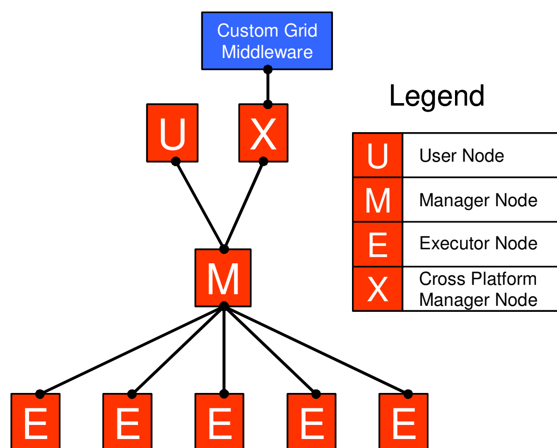

Figure 3. Distributed components and their relationships.

2.2.1 Manager

The Manager provides services associated with managing execution of grid applications and their constituent threads. Executors register themselves with the Manager, which in turn monitors their status. Threads received from the User are placed in a pool and scheduled to be executed on the various available Executors. A priority for each thread can be explicitly specified when it is created or submitted. Threads are scheduled on a Priority and First Come First Served (FCFS) basis, in that order. The Executors return completed threads to the Manager which are subsequently collected by the respective users. A scheduling API is provided that allows custom schedulers to be written.

2.2.2 Executor

The Executor accepts threads from the Manager and executes them. An Executor can be configured to be dedicated, meaning the resource is centrally managed by the Manager, or non-dedicated, meaning that the resource is managed on a volunteer basis via a screen saver or explicitly by the user. For non-dedicated execution, there is one-way communication between the Executor and the Manager. In this case, the resource that the Executor resides on is managed on a volunteer basis since it requests threads to execute from the Manager. When two-way communication is possible and dedicated execution is desired the Executor exposes an interface so that the Manager may communicate with it directly. In this case, the Manager explicitly instructs the Executor to execute threads, resulting in centralized management of the resource where the Executor resides. Thus, Alchemi's execution model provides the dual benefit of:

- flexible resource management i.e. centralized management with dedicated execution vs. decentralized management with non-dedicated execution; and
- flexible deployment under network constraints i.e. the component can be deployment as non-dedicated where two-way communication is not desired or not possible (e.g. when it is behind a firewall or NAT/proxy server).

Thus, dedicated execution is more suitable where the Manager and Executor are on the same Local Area Network while non-dedicated execution is more appropriate when the Manager and Executor are to be connected over the Internet.

2.2.3 User

Grid applications are executed on the User node. The API abstracts the implementation of the grid from the user and is responsible for performing a variety of services on the user's behalf such as submitting an application and its constituent threads for execution, notifying the user of finished threads and providing results and notifying the user of failed threads along with error details.

2.2.4 Cross-Platform Manager –Web Services

The Cross-Platform Manager is a web services interface that exposes a portion of the functionality of the Manager in order to enable Alchemi to manage the execution of grid jobs (as opposed to grid applications utilizing the Alchemi grid thread model). Jobs submitted to the Cross-Platform Manager are translated into a form that is accepted by the Manager (i.e. grid threads), which are then scheduled and

executed as normal in the fashion described above. In addition to support for the grid-enabling of legacy applications, the Cross-Platform Manager allows custom grid middleware to interoperate with and leverage Alchemi on any platform that supports web services.

3 Design and Implementation

Figure 4 and Figure 5 provide an overview of the design and implementation by way of a deployment diagram and class diagram (showing only the main classes without attributes or operations) respectively.

Web services were considered briefly for this purpose, but were decided against due to the relatively higher overheads involved with XML-encoded messages, the inherent inflexibility of the HTTP protocol for the requirements at hand and the fact that each component would be required to be configured with a web services container (web server). However, web services are used for the Cross-Platform Manager's public interface since cross-platform interoperability was the primary requirement in this regard.

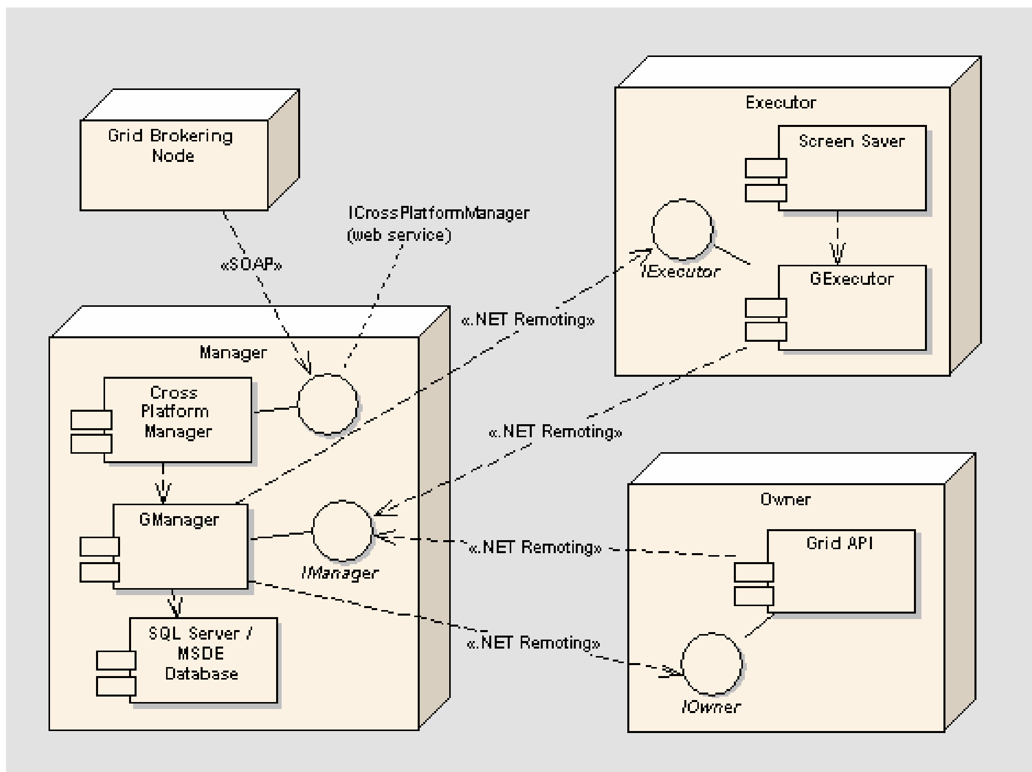

Figure 4. Alchemi architecture and interaction between its components.

3.1 Overview

The .NET Framework offers two mechanisms for execution across application domains – Remoting and web services (application domains are the unit of isolation for a .NET application and can reside on different network hosts). .NET Remoting allows an object to be "remoted" and expose its functionality across application domains. Remoting is used for communication between the four Alchemi distributed grid components as it allows low-level interaction transparently between .NET objects with low overhead (remote objects are configured to use binary encoding for messaging).

The objects remoted using .NET Remoting within the four distributed components of Alchemi, the Manager, Executor, Owner and Cross-Platform Manager are instances of `GManager`, `GExecutor`, `GApplication` and `CrossPlatformManager` respectively.

It should be noted that classes are named with respect to their roles vis-à-vis a grid application. This discussion therefore refers to an 'Owner' node synomymously with a 'User' node, since the node where the grid application is being submitted from can be considered to "own" the application.

The prefix 'I' is used in type names to denote an interface, whereas 'G' is used to denote a 'grid node' class. **GManager**, **GExecutor**, **GApplication** derive from the **GNode** class which implements generic functionality for remoting the object itself and connecting to a remote Manager via the **IManager** interface.

The Manager executable initializes an instance of the **GManager** class, which is always remoted and exposes a public interface **IManager**. The Executor executable creates an instance of the **GExecutor** class. For non-dedicated execution, there is one-way communication between the Executor and the Manager. Where two-way communication is possible and dedicated execution is desired, **GExecutor** is remoted and exposes the **IExecutor** interface so that the Manager may communicate with it directly. The Executor installation provides an option to install a screen saver, which initiates non-dedicated execution when activated by the operating system.

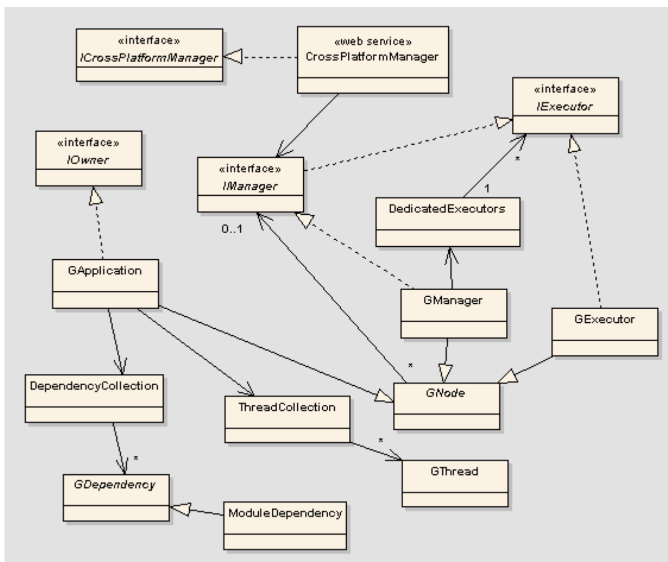

Figure 5. Main classes and their relationships.

The **GApplication** object in Alchemi API communicates with the Manager in a similar fashion to **GExecutor**. While two-way communication is currently not used in the implementation, the architecture caters for this by way of the **IOwner** interface.

The Cross-Platform Manager web service is a thin wrapper around **GManager** and uses applications and threads internally to represent tasks and jobs (the **GJob** class derives from **GThread**) via the public **ICrossPlatformManager** interface.

3.2 Grid Application Lifecycle

To develop and execute a grid application the developer creates a custom grid thread class that derives from the abstract **GThread** class. An instance of the **GApplication** object is created and any dependencies required by the application are added to its **DependencyCollection**. Instances of the **GThread**-derived class are then added to the **GApplication**'s **ThreadCollection**.

The **GApplication** serializes and sends relevant data to the Manager, where it is persisted to disk and threads scheduled. Application and thread state is maintained in a SQL Server / MSDE database. Non-dedicated executors poll for threads to execute until one is available. Dedicated executors are directly provided a thread to execute by the Manager.

Threads are executed in .NET application domains, with one application domain for each grid application. If an application domain does not exist that corresponds to the grid application that the thread belongs to, one is created by requesting, desterilizing and dynamically loading the application's dependencies. The thread object itself is then desterilized, started within the application domain and returned to the Manager on completion.

After sending threads to the Manager for execution, the **GApplication** polls the Manager for finished threads. A user-defined **GThreadFinish** delegate is called to signify each thread's completion and once all threads have finished a user-defined **GApplicationFinish** delegate is called.

4 Performance Evaluation

4.1 Alchemi Enterprise Grid

Testbed

The testbed is an Alchemi cluster consisting of six Executors (Pentium III 1.7 GHz desktop machines with 512 MB physical memory running Windows 2000 Professional). One of these machines is additionally designated as a Manager.

Test Application & Methodology

The test application is the computation of the value of Pi to n decimal digits. The algorithm used allows the computation of the p'th digit without knowing the previous digits [27]. The application utilizes the Alchemi grid thread model. The test was performed for a range of workloads (calculating 1000, 1200, 1400, 1600, 1800, 2000 and 2200 digits of Pi), each with one to six Executors enabled. The workload was

sliced into a number of threads, each to calculate 50 digits of Pi, with the number of threads varying proportionally with the total number of digits to be calculated. Execution time was measured as the elapsed clock time for the test program to complete on the Owner node.

Resource	Location	Configuration	Middleware	Jobs Completed
maggie.cs.mu.oz.au [Windows cluster]	University of Melbourne	6 * Intel Pentium IV 1.7 GHz	Alchemi	21
quidam.ucsd.edu [Linux cluster]	Univ. of California, San Diego	1 * AMD Athlon XP 2100+	Globus	16
belle.anu.edu.au [Linux cluster]	Australian National University	4 * Intel Xeon 2	Globus	22
koume.hpcc.jp [Linux cluster]	AIST, Japan	4 * Intel Xeon 2	Globus	18
brecca-2.vpac.org [Linux cluster]	VPAC Melbourne	4 * Intel Xeon 2	Globus	23

Table 1. Grid resources.

Results

Figure 6 shows a plot between thread size (the number of decimal places to which Pi is calculated to) and total time (in seconds taken by the all threads to complete execution) with varying numbers of Executors enabled.

At a low workload (1000 digits), there is little difference between the total execution time with different quantity of Executors. This is explained by the fact that the total overhead (network latency and overheads involved in managing a distributed execution environment) is in a relatively high proportion to the actual total computation time. However, as the workload is increased, there is near-proportional difference when higher numbers of executors are used. For example, for 2200 digits, the execution time with six executors (84 seconds) is nearly 1/5th of that with one executor (428 seconds). This is explained by the fact that for higher workloads, the total overhead is in a relatively lower proportion to the actual total computation time.

4.2 Alchemi as a Global Grid Node

Testbed

A global grid was used for evaluating Alchemi in a cross-platform environment with the Gridbus Grid Service Broker managing five grid resources (see **Table 1**). One of the grid nodes was powered by Alchemi while the other resources by Globus 2.4 [7]. The Gridbus resource brokering mechanism used in

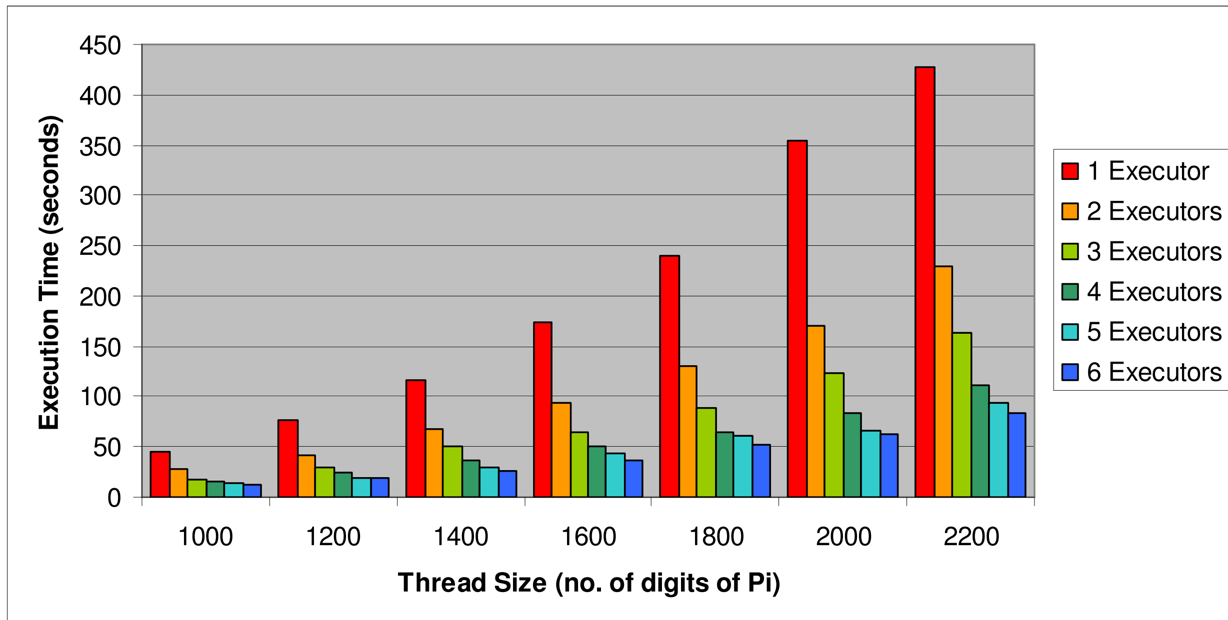

Figure 6: A plot of thread size vs. execution time on an enterprise desktop grid.

this test obtains the users' application requirements and evaluates the suitability of various resources for the purpose. It then schedules the jobs to various resources in order to satisfy those requirements.

```
#Parameter definition
parameter X integer range from 1 to 100 step 1;
parameter Y integer default 1;
#Task definition
task main
   #Copy necessary executables depending on node type
   copy calc.$OS node:calc
   #Execute program with parameter values on remote node
   node:execute ./calc $X $Y
   #Copy results to home node with jobname as extension
   copy node:output ./output.$jobname
endtask
```

Figure 7: Parametric job specification.

Test Application & Methodology

For the purpose of evaluation, we used an application that calculates mathematical functions based on the values of two input parameters. The first parameter *X*, is an input to a mathematical function and the second parameter *Y,* indicates the expected calculation complexity in minutes plus a random deviation value between 0 to 120 seconds—this creates an illusion of small variation in execution time of different parametric jobs similar to a real application. A plan file modeling this application as a parameter sweep application using the Nimrod-G parameter specification language [12] is shown in Figure 7. The first part defines parameters and the second part defines the task that is to be performed for each job. As the parameter *X* varies from values 1 to 100 in step of 1, this plan file would create 100 jobs with input values from 1 to 100.

Results

The results of the experiment shown in Figure 8 show the number of jobs completed on different Grid resources at different times. The parameter calc.$OS directs the broker to select appropriate executables based a target Grid resource architecture. For example, if the target resource is Windows/Intel, it selects calc.exe and copies to the grid node before its execution. It demonstrates the feasible to utilizing Windows-based Alchemi resources along with other Unix-class resources running Globus.

5 Related Work

A number of enterprise Grid systems have been developed by in academia and industries. They include Condor [18], SETI@home [9][14] , Entropia [16], GridMP [21], and XtermWeb [15]. Compared to these related systems, Alchemi has several distinguished features (see Table 2) in addition to its implementation based on service-oriented architecture based state-of-the-art Web services technologies such as XML, .Microsoft's .NET framework and platform.

6 Summary and Future Work

We have discussed a .NET-based grid computing framework that provides the runtime machinery and object-oriented programming environment to easily construct enterprise grids and develop grid applications. Its integration into the global cross-platform grid has been made possible via support for execution of grid jobs via a web services interface and the use of a broker component.

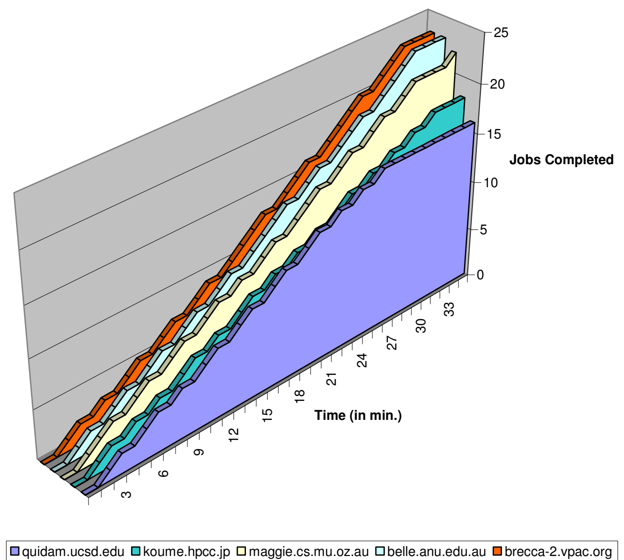

Figure 8. A plot of the number of jobs completed on different resources versus the time.

Property \ System	Alchemi	Condor	SETI@home	Entropia	XtermWeb	Grid MP
Architecture	Hierarchical	Hierarchical	Centralized	Centralized	Centralized	Centralized
Implementation Technologies	C#, Web Services, & .NET Framework	C	C++, Win32	C++, Win32	Java, Linux	C++, Win32
Multi-Clustering	Yes	Yes	No	No	No	Yes
Global Grid Brokering Mechanism	Yes (via Gridbus Broker)	Yes (via Condor-G)	No	No	No	No
Thread Programming Model	Yes	No	No	No	No	No
Level of integration of application, programming and runtime environment	Low (general purpose)	Low (general purpose)	High (single purpose, single application environment)	Low (general purpose)	Low (general purpose)	Low (general purpose)
Web Services Interface	Yes	No	No	No	No	Yes

Table 2. Comparison of Alchemi and some related enterprise grid systems.

We plan to extend Alchemi in a number of areas. Firstly, support for additional functionality via the API including inter-thread communication is planned. Secondly, we are working on support for multi-clustering with peer-to-peer communication between Managers. Thirdly, we plan to support utility-based resource allocation policies driven by economic, quality of services, and service-level agreements. Fourthly, we are investigating strategies for adherence to OGSI (Open Grid Services Infrastructure) standards by extending the current Alchemi job management interface. This is likely to be achieved by its integration with .NET-based low-level grid middleware implementations (e.g., University of Virginia's WSRF.NET [31] platform) that conform to grid standards such as WSRF (Web Services Resource Framework), which is an extension of OGSI (Open Grid Services Infrastructure) [23][30]. Finally, we plan to provide data grid capabilities to enable resource providers to share their data resources in addition to computational resources.

7 References

[1] Ian Foster and Carl Kesselman (editors), *The Grid: Blueprint for a Future Computing Infrastructure*, Morgan Kaufmann Publishers, USA, 1999.

[2] Larry Smarr and Charlie Catlett, Metacomputing, *Communications of the ACM Magazine*, Vol. 35, No. 6, pp. 44-52, ACM Press, USA, Jun. 1992.

[3] Microsoft Corporation, *.NET Framework Home*, http://msdn.microsoft.com/netframework/ (accessed November 2003)

[4] Piet Obermeyer and Jonathan Hawkins, *Microsoft .NET Remoting: A Technical Overview*, http://msdn.microsoft.com/library/en-us/dndotnet/html/hawkremoting.asp (accessed November 2003)

[5] Microsoft Corp., *Web Services Development Center*, http://msdn.microsoft.com/webservices/ (accessed November 2003)

[6] D.H. Bailey, J. Borwein, P.B. Borwein, S. Plouffe, The quest for Pi, Math. Intelligencer 19 (1997),pp. 50-57.

[7] Ian Foster and Carl Kesselman, Globus: A Metacomputing Infrastructure Toolkit, *International Journal of Supercomputer Applications*, 11(2): 115-128, 1997.

[8] Ian Foster, Carl Kesselman, and S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International Journal of Supercomputer Applications*, 15(3), Sage Publications, 2001, USA.

[9] David Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, Dan Werthimer, SETI@home: An Experiment in Public-Resource Computing, *Communications of the ACM,* Vol. 45 No. 11, ACM Press, USA, November 2002.

[10] Yair Amir, Baruch Awerbuch, and Ryan S. Borgstrom, The Java Market: Transforming the Internet into a Metacomputer, *Technical Report CNDS-98-1*, Johns Hopkins University, 1998.

[11] Peter Cappello, Bernd Christiansen, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu, Javelin: Internet-Based Parallel Computing Using Java, *Proceedings of the 1997*

ACM Workshop on Java for Science and Engineering Computation, June 1997.

[12] Rajkumar Buyya, David Abramson, Jonathan Giddy, *Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid,* Proceedings of 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2000), Beijing, China, 2000.

[13] Rajkumar Buyya, Economic-based Distributed Resource Management and Scheduling for Grid Computing, *Ph.D. Thesis*, Monash University Australia, April 2002.

[14] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, D. Anderson, *A new major SETI project based on Project Serendip data and 100,000 personal computers*, Proceedings of the 5th International Conference on Bioastronomy, 1997.

[15] Cecile Germain, Vincent Neri, Gille Fedak and Franck Cappello, *XtremWeb: building an experimental platform for Global Computing*, Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000), Bangalore, India, Dec. 2000.

[16] Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia, Entropia: architecture and performance of an enterprise desktop grid system, *Journal of Parallel and Distributed Computing*, Volume 63, Issue 5, Academic Press, USA, May 2003.

[17] Andy Oram (editor), *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, O'Reilly Press, USA, 2001.

[18] M. Litzkow, M. Livny, and M. Mutka, *Condor - A Hunter of Idle Workstations*, Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS 1988), January 1988, San Jose, CA, IEEE CS Press, USA, 1988.

[19] N. Nisan, S. London, O. Regev, and N. Camiel, *Globally Distributed computation over the Internet: The POPCORN project*, International Conference on Distributed Computing Systems (ICDCS'98), May 26 - 29, 1998, Amsterdam, The Netherlands, IEEE CS Press, USA, 1998.

[20] Y. Aridor, M. Factor, and A. Teperman, *cJVM: a Single System Image of a JVM on a Cluster*, Proceedings of the 29th International Conference on Parallel Processing (ICPP 99), September 1999, Fukushima, Japan, IEEE CS Press, USA.

[21] Intel Corporation, *United Devices' Grid MP on Intel Architecture,* http://www.ud.com/rescenter/files/wp_intel_ud.pdf (accessed November 2003)

[22] Ardaiz O., Touch J. *Web Service Deployment Using the Xbone*, Proceedings of Spanish Symposium on Distributed Systems SEID 2000.

[23] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*, January 2002.

[24] P. Cauldwell, R. Chawla, Vivek Chopra, Gary Damschen,Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev, *Professional XML Web Services,* Wrox Press, 2001.

[25] E. O'Tuathail and M. Rose, Using the Simple Object Access Protocol (SOAP) in Blocks Extensible Exchange Protocol (BEEP), *IETF RFC 3288*, June 2002.

[26] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 1.1.W3C Note 15*, 2001. www.w3.org/TR/wsdl.

[27] World Wide Web Consortium, *XML Schema Part 0:Primer: W3C Recommendation*, May 2001.

[28] Fabrice Bellard, *Computation of the n'th digit of pi in any base in O(n^2)*, http://fabrice.bellard.free.fr/pi/pi_n2/pi_n2.html (accessed June 2003).

[29] C. Kruskal and A. Weiss, Allocating independent subtasks on parallel processors, *IEEE Transactions on Software Engineering*, 11:1001--1016, 1984.

[30] Global Grid Forum (GGF), *Open Grid Services Infrastructure (OGSI) Specification 1.0*, https://forge.gridforum.org/projects/ogsi-wg (accessed January 2004).

[31] Glenn Wasson, Norm Beekwilder and Marty Humphrey, *OGSI.NET: An OGSI-Compliant Hosting Container for the .NET Framework,* University of Virginia, USA, 2003. http://www.cs.virginia.edu/~humphrey/GCG/ogsi.net.html (accessed Jan 2004).

[32] M. Mutka and M. Livny, The Available Capacity of a Privately Owned Workstation Environment, *Journal of Performance Evaluation, Volume 12, Issue 4,* **,** 269-284pp, Elsevier Science, The Netherlands, July 1991.

[33] Srikumar Venugopal, Rajkumar Buyya, and Lyle Winton, A Grid Service Broker for Scheduling Distributed Data-Oriented Applications on Global Grids, Proceedings of the 2nd International Workshop on Middleware for Grid Computing (Co-located with Middleware 2004, Toronto, Ontario - Canada, October 18, 2004), ACM Press, 2004, USA.

[34] Agus Setiawan, David Adiutama, Julius Liman, Akshay Luther and Rajkumar Buyya, *GridCrypt: High Performance Symmetric Key using Enterprise Grids*, Proceedings of the 5th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2004, December 8-10, 2004, Singapore), Springer Verlag Publications (LNCS Series), Berlin, Germany.

[35] Krishna Nadiminti, Yi-Feng Chiu, Nick Teoh, Akshay Luther, Srikumar Venugopal, and Rajkumar Buyya, *ExcelGrid: A .NET Plug-in for Outsourcing Excel Spreadsheet Workload to Enterprise and Global Grids*, Proceedings of the 12th International Conference on Advanced Computing and Communication (ADCOM 2004, December 15-18, 2004), Ahmedabad, India.

1. Computational Grids Computational Grids account for a lion share of Grid Computing usage now and will certainly retain this lead in the near future to the least. In a nutshell, the idea behind it is very simple: if you have a task that is executing unacceptably long time, you can split this task into multiple sub-tasks, execute each sub-task in parallel on a separate computer, combine results from the sub-tasks and get the original task's result O(N)-times faster, where N is a number of.Â You solve the scalability but not performance, i.e. you can execute more tasks without slowing down, but you won't be executing each task faster (in many cases you will be executing each task actually slower due to extra work for off-loading task to a remote computer, remote computer being less powerful, etc.).