GTPE: A Thread Programming Environment for the Grid

Harold Soh, Shazia Haque, Weili Liao, Krishna Nadiminti and Rajkumar Buyya

Grid Computing and **D**istributed **S**ystems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne,
ICT Building, 111 Barry Street,
Carlton, VIC 3053, Australia

{ shsoh, haques, wlliao}@students.cs.mu.oz.au, {kna, raj}@cs.mu.oz.au

Abstract

Grid computing has emerged as a viable method for solving computational and data intensive problems, applicable over various domains from business computing to scientific research. However, grid environments are largely heterogeneous, distributed and dynamic, increasing the complexities involved in developing grid applications. Several software constructs have been developed to provide programming environments that hide these complexities, simplifying grid application development. In this paper, we present the Grid Thread Programming Environment (GTPE) for the Gridbus Broker [1], a software grid resource broker developed at the GRIDS Lab, University of Melbourne. GTPE is implemented in pure Java and consists of a thread library that interacts with the Gridbus broker to provide transparent access to grid services. As such, GTPE provides a finer level of application control while freeing the developer from the complexities introduced by grid resource management. This paper describes architecture, overall design, implementation and performance evaluation of the grid thread programming environment.

1. INTRODUCTION

Grid applications are the next-generation network applications for solving the world's computational and data-intensive intensive problems and support integrated and secure use of a variety of shared and distributed resources such as high-performance computers, workstations, data repositories, instruments, and even sensors. However, the heterogeneous and dynamic nature of the grid requires that grid applications not only be high performing but also robust and fault-tolerant. Designing and implementing applications that possess such features from the ground up is often difficult. As such, several middleware programming environments have been developed to provide grid management services, with the aim to reducing the complexities involved with grid application development. One such programming environment is provided by the Gridbus Broker [1] through an extensive Java Application Programming Interface (API).

   The Gridbus Broker, developed at the GRIDS Lab at the University of Melbourne, is a software component that permits access heterogeneous resources transparently, providing services such as resource discovery, secure access, scheduling, monitoring, and job submission [3]. Although application development using the Gridbus Broker's object-orientated API is straightforward, there exist certain drawbacks when working directly with the API. The coarse-grain nature of jobs, the units of work assigned to a grid node, requires them to be implemented as separate executable binaries and copied over to the nodes for execution. Programmers using the Gridbus API are often required to develop two programs; one to interact with the Gridbus broker and another that performs the actual computational work. The application interfacing to the Gridbus broker is required to specify low-level constructs such as copy, execute commands and program arguments. While the task of specifying these parameters is not difficult, it may be tedious for applications utilizing a large number of different job programs and detracts focus from the original problem task. Additionally, since the grid is a heterogeneous environment, it may be necessary to provide different recompilations of the job programs (unless they are in Java bytecode) or develop conversion plug-ins.

   We argue that for certain applications it is more intuitive to be able to think of the computations as functions or threads that could be "executed" from within the application. Using a thread library would allow programmers to develop a self-contained grid application, distributing threads for execution instead of programs. This frees the programmer from having to specify application-descriptors or lower-level

commands. Threads also afford the programmer an additional amount of flexibility as it is possible to work with thread level objects using methods not possible with external job programs.

This paper presents the Grid Thread Programming Environment (GTPE), a programming environment implemented in Java utilizing the Gridbus Broker API. GTPE further abstracts the task of grid application development, automating grid management while providing a finer level of logical program control. In the following section, we describe the design and architecture of the GTPE system. Section 3 discusses the GTPE implementation, with an emphasis on the services provided. This is followed by a section on performance testing utilizing a sample application developed using GTPE. This paper concludes with known limitations of the current system and considerations for future work.

2. ARCHITECTURE

In this section, we present a high-level overview of Grid-Thread Programming Environment that uses services provided by the Gridbus Resource Broker for deploying threads on global Grids. The GTPE is architected with the following primary design objectives:

1. **Usability and Portability.** The heterogeneous nature of resources on the grid requires that programs running on them be largely processor and architecture independent. Hence, a grid programming environment should be geared towards designing applications that are able to run successfully on different machines.
2. **Flexibility.** The grid programming environment needs to support a large class of grid applications and impose as few restrictions as possible.
3. **Performance.** One of the main advantages of working on the grid is the high performance that can be obtained by parallel execution. A grid programming environment should ensure that quality of service is maintained in a dynamic environment.
4. **Fault Tolerance.** Resources are not globally administered in the grid and hence, can join and exit the grid at any time. There is also exists a non-zero probability that a resource will fail during computation. As such, grid programming environments have to provide mechanisms for detecting changes to the grid environment and recovery services.
5. **Security**. Grid applications will normally be executed on remote servers across multiple administrative domains. Security is hence a concern and it is necessary to ensure secure access to these resources and the protection of information during transport of code and data.

To support these design objectives, GTPE was implemented in pure Java as a layer on top of the Gridbus Broker API. GTPE is responsible for thread management and interfacing to the broker which provides grid services. Figure 1 illustrates an architecture block-diagram of GTPE. GTPE consists of two main components, the *GridApplication* class and the *GridThread* class.

The *GridThread* object forms the atomic unit of remote, independent work. All user defined grid threads derive from the *GridThread* abstract base class. The subclass has to override the *start* and the *callback* methods. The *start* method is executed on the remote nodes and hence, the computational work intended for remote execution should be defined in this method. The *callback* method is executed at the local client node once the thread has finished executing on the remote node and has been transported back. The *callback* method can be used for a variety of functions including the aggregation of results and the reporting of thread completion to the user.
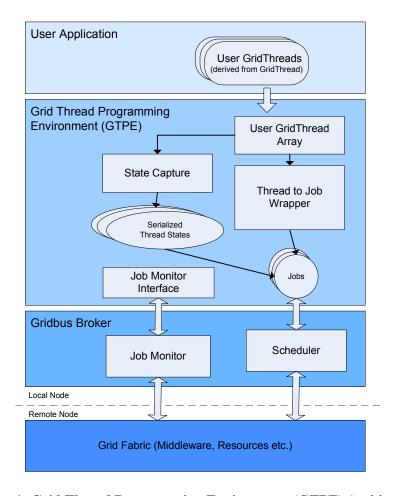
Figure 1: Grid Thread Programming Environment (GTPE) Architecture.

The *GridApplication* object is responsible for thread management and providing near-transparent access to the grid via the Gridbus broker. The class presents a single point of control to the programmer. *GridThreads* are added to a *GridApplication* object for execution on remote nodes. The GridApplication object provides mechanisms to capture and restore thread states, as well as job wrapping and thread monitoring services.

The Gridbus-Thread Programming Environment architecture is relatively simple and supports the aforementioned design objectives. GTPE is implemented in pure Java and as such, benefits from its "write-once-run-anywhere" model. User derived grid threads are inherently portable and able to run on any system which provides access to a Java Virtual Machine. Performance is supported via dynamic scheduling and modern Java compilers, which are able to able to generate Java code capable of execution speeds comparable to traditional high-performance languages [4]. Secure access and job submission to remote nodes is supported via the Gridbus broker and thread monitoring provides a mechanism for detecting thread failures on remote nodes.

3. DESIGN AND IMPLEMENTATION

We now present the implementation of GTPE layer based on the architecture described in the previous section. Figure 2 illustrates the main components of GTPE and the methods implemented in each class.
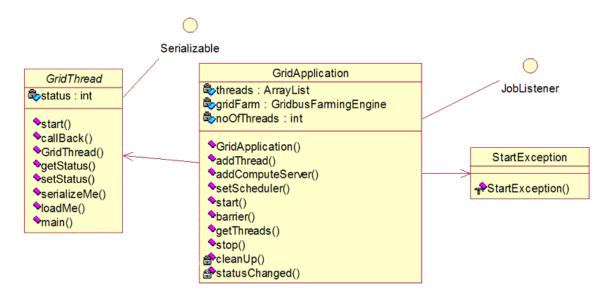
3

Figure 2: UML Diagram of the GTPE Main Components.

3.1. Resource Discovery and Access

The *GridApplication* class provides the *addComputeServer* method that permits users to specify applicable grid resources. Version 2.0 of the Gridbus broker (and hence, GTPE) supports the following a range of middleware for computational resources (Globus v2.4 and v3.2, Alchemi v0.8, and Unicore Gateway v41) and data resources (SRB v3.x and Globus Replica Catalog) [3]. If no resources are specified, a set of servers is loaded from the resources.xml file, which specifies default resources and their attributes. The Gridbus broker manages access to these systems, providing secure access via proxies and credentials [3].

3.2. Thread Object State Capture and Job Wrapping

To capture objects into a form which can be distributed, GTPE utilizes Java *serialization*. The *GridThread* base class implements the *java.io.Serializable* interface and two static methods, *serializeMe* (to write the thread object to a state file) and *loadMe* (to load an object from a serialized state file)[1]. As such, all derived subclasses are serializable by default. Serialization is automated by the GridApplication class and is transparent to the user application. A Java ArrayList, *threads*, to utilized store and manage the GridThreads. When a thread is added to the ArrayList, it is immediately serialized to a state file with a unique filename. This state file is grouped together with the user derived GridThread class file (obtained using Java class inspection), and the GridThread class file and wrapped into a Gridbus broker job along with the appropriate low-level copy and execute commands. The job object is then added to the Gridbus broker for scheduling and submission to a suitable grid node.

3.3. Thread Scheduling

The Gridbus broker utilizes the concept of a computational economy [5] and version 2.0 provides five different scheduling types; cost-optimized, time-optimized, cost-and-time-optimized, cost-and-data-optimized and time-and-data-optimized [3]. It is possible to set the scheduling algorithm used via the *setScheduler* method providing the application developer with the flexibility of selecting the most optimal scheduling approach for the task. If no scheduler is specified, the GridApplication defaults to using the cost-optimized approach.

---

[1] Future work involves permitting serializeMe and loadMe member functions to be overridden in subclasses. This would allow developers to specify more optimized serialization methods.

3.5. Thread Execution and Monitoring

Thread scheduling, distribution, execution and monitoring is started by calling the *GridApplication.start* method. The *start* method can only be called once per session to prevent the spawning of multiple resource brokers. At the remote node, the *main* function in the GridThread base class detects the appropriate user defined subclass and instantiates the appropriate thread object via Java *reflection*. The *loadMe* method is called to restore the thread's serialized state and the thread's *start* method is invoked. When the *start* method returns, the thread's state is serialized to a file on disk via the *serializeMe* method and transported back to the local node.

To provide thread monitoring services, the GridApplication class implements the *Gridbus.broker.event.JobListener* interface. Each GridThread has an associated status variable which stores one of four possible execution states; *notsubmmited*, *running*, *finished* or *failed*. The default status is the *notsubmmited* state and remains unchanged while it is waiting for transport. When the thread has been submitted to the remote node, its status variable is updated to *running*. If a thread successfully completes, its state in the *threads* ArrayList is updated by de-serializing the finished thread state file, its status is set to *finished* and the thread's *callback* method is called. If a thread fails during execution, an error report is generated and its status is changed to *failed*.

3.6. Additional Functionality

GTPE provides additional functionality to minimize the effort necessary to work with grid threads. A *barrier* function is implemented allowing users to synchronize threads and the *getThreads* method is available for retrieving threads that have been added to the threads array. These methods are especially useful when aggregating results or performing some final analysis which requires all threads to have completed execution. Hence, unlike regular broker jobs, it is possible to work with updated threads after execution on a remote node. Additionally, the *stop* method is provided to terminate the scheduling and distribution of threads.

4. GTPE PERFORMANCE EVALUATION

To evaluate the performance characteristics of applications utilizing GTPE, we created a sample application, PrimeFinder, which computes the number of primes smaller than a supplied parameter N utilizing T number of threads. For our purposes, PrimeFinder was not heavily optimized and distributes work among threads in the following naïve fashion: For a given N and T, the thread T_i (where $i = 2, 3…, T$) performs a primality test all numbers in the set:

$$\{2i – 1+ 2jT \text{ for } j = 0, 1, 2, 3…, k \text{ where } (2i - 1 + 2kT) \leq N\}.$$

For example, for N = 10 and T = 2, thread T_0 evaluates $\{1, 5, 9\}$ and T_1 evaluates $\{3, 7\}$. Figure 3 shows the basic algorithm of the PrimeFinder sample application[2].

```
int startPoint = 2*threadID - 1;
int stopPoint = N;
int step = 2*numberOfThreads;

public void start() {
        total = 0;
        for (int i=startPoint; i<=stopPoint; i=i+step){
                if (isPrime(i)) total++;
        }
}

public boolean isPrime(int N) {
```

---

[2] We acknowledge that the number 2 is always skipped when using this algorithm. However, since we are only interested in the total count of primes smaller than N, we take into account this special case by evaluating 1 as prime.

```
        int max = (int) Math.sqrt(N);
        for (int div = 2; div <= max; div++) {
                if (N % div ==0)
                        return false;
        }
        return true;
}
```

Figure 3: Basic algorithm of the PrimeFinder sample application

Our test bed consisted of two resources, belle.cs.mu.oz.au and manjra.cs.mu.oz.au, both based in the GRIDS laboratory, at the Department of Computer Science and Software Engineering, University of Melbourne. Table 1 lists the configuration of both resources.

Server Name	Configuration	Grid Middleware
belle.cs.mu.oz.au	IBM eServer with 4 IA-32 CPUs.	Globus v2.4
manjra.cs.mu.oz.au	Linux Cluster with 13 IA-32 CPUs	Globus v2.4

Table 1: Testbed resources

We then evaluated the performance of the grid-enabled sample application by recording and comparing the execution times for varying values of N and T. Note that if T is one, then GridThreads are not utilized and the algorithm is run locally on a single resource (belle.cs.mu.oz.au). Table 2 below lists the performance results obtained from our tests and Figure 3 graphs our results for the PrimeFinder with T = 1, 2, 4, 8 and 14. Our results show that for values of N larger than 50 million, a performance increase of approximately 300 to 360 percent (as compared to the non-threaded performance results) when utilizing 8 or more GridThreads.

Threads	N (Maximum Search Value)					
	20000000	30000000	40000000	50000000	60000000	70000000
1*	97.43	173.45	261.86	359.55	466.62	581.59
2	87.96	124.24	166.32	229.61	279.61	337.17
4	67.19	88.99	117.91	146.96	175.01	198.82
6	74.09	95.63	124.41	153.85	175.35	217.88
8	74.32	82.60	105.37	120.50	136.45	167.21
10	88.90	97.28	120.51	136.08	167.21	190.71
12	74.98	90.08	105.82	134.82	151.39	173.92
14	92.69	106.69	107.39	116.97	137.90	161.15

*Run locally without using GTPE.

Table 2: PrimeFinder execution time (seconds) with increasing N utilizing varying number of threads.

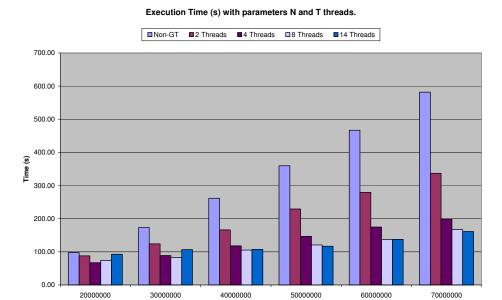**Execution Time (s) with parameters N and T threads.**

Figure 3: Bar Graph of PrimeFinder execution time (seconds) with increasing N utilizing 1, 2, 4, 8 and 14 threads.

We also observe that the performance gains obtained from utilizing a larger number of threads is non-linear and in certain cases, detrimental. This result is can be explained by the fact that increasing the number of threads also increases the overhead associated with thread transport and management. Hence, there exists an optimal number of threads for each input N, after which the addition of more threads would only serve to decrease overall performance. We expect this result to be applicable across all applications developed with GTPE.

5. RELATED WORK

There has been a significant amount of research devoted to building grid programming environments. Similar work involving Java distributed threads include a thread extension [6] to KaRMI [7] and JavaParty [8]. KaRMI is an optimized implementation of Java's Remote Method Invocation (RMI) and serialization. RMI provides communication across distributed virtual machines, allowing Java applications to reference and access remotely exported objects. JavaParty provides remote objects to Java by declaration, simplifying multi-threaded cluster programming in Java. Other projects which use a middleware library to implement distributed computing in Java include Ibis [9], FarGo [10], JavaSymphony [11] and J-Orchestra [12]. A grid-enabled message passing variant utilizing Java is G-JavaMPI [13].

Another approach has been to provide JVMs which are inherently aware of distributed resources. A Distributed JVM (DJVM) offers a single system image view to Java threads and can provide parallel execution for regular Java Threads. Projects involving DJVMs include JESSICA2 [14], cJVM[15], Java/DSM [16]. The main drawback of this method is that the DJVM has to be installed on every node for it to be effective. Hence, this work is more applicable to locally administered clusters.

Other notable grid programming environments include Alchemi [17], The Grid Application Toolkit (GAT) [18], GridRPC [2] and Grid Superscalar [19]. Alchemi is a Microsoft .NET grid computing framework developed at the GRIDS lab which provides a thread programming environment similar to that provided by GTPE. The Grid Application Toolkit (GAT) provides a set of coordinated, generic and flexible APIs that can be accessed from a variety of programming languages. GridRPC is a Remote Procedure Call (RPC) model and API providing standard RPC services on grids. Grid Superscalar is a relatively new programming environment which automatically parallelizes a sequential application by automatically detecting task concurrency and dependencies.

6. CONCLUSIONS AND FUTURE WORK

The main objective of implementing GridThreads library for Gridbus Broker was to minimize the entry barriers associated with grid applications development. Implemented as a pure Java layer on top of the Gridbus broker API, the GTPE programming environment combines the services provided by the broker with the flexibility and portability of Java threads. This gives developers greater application control while avoiding the finer-level complexities associated with grid resource management.

Although GTPE is a currently a stable working environment, it is very much preliminary work and we plan to extend its functionality to provide developers with a fully-featured programming environment. Our plans for future work on GTPE include:

1. **Usability improvements.** It may be possible to eliminate the GridApplication class, integrating its functionality into the GridThread class or the Gridbus Broker, thereby simplifying the thread model. We also plan to explore the possibility of extending the model to include thread grouping for delivery to the same resource.
2. **Flexibility improvements.** Threads built utilizing GTPE currently have to be self contained. We plan to augment GTPE to automatically discover file dependencies.
3. **Performance improvements.** The serialization methods used in GTPE could be optimized and enhanced to provide better performance. As stated in Section 2, we plan to allow developers to override the serialization methods and provide optimal implementations that best fit their applications.
4. **Fault tolerance.** GTPE currently does not provide much functionality for the recovery or resubmission of failed threads.
5. **Additional features.** An interesting area of work would be to develop an efficient method of inter-thread communication both locally and globally. This can be perhaps be achieved using Java's RMI or a more efficient implementation, such as KaRMI.

References

[1]   S. Venugopal, R. Buyya and L. Winton, "A Grid Service Broker for Scheduling Distributed Data-Oriented Applications on Global Grids", Technical Report, GRIDS-TR-2004-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, February 2004.
[2]   C. Lee and D. Talia. "Grid programming models: Current tools, issues, and directions.", In *Grid Computing: Making The Global Infrastructure a Reality*, F. Berman, A. Hey, and G. Fox, editors, John Wiley & Sons, 2003.
[3]   Krishna Nadiminti, Srikumar Venugopal, Hussein Gibbins, and Rajkumar Buyya, The Gridbus Grid Service Broker and Scheduler (2.0) User Guide, Technical Report, GR IDS-TR-2005-4, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, Apri l 22, 2005.
[4]   J. Bull, L. Smith, L. Potttage, and R. Freeman. "Benchmarking Java against C and Fortran for Scientific Applications.", In *ACM 2001 Java Grande/ISCOPE Conf.*, pp 97-105, 2001.
[5]   R. Buyya, D. Abramson, and J. Giddy, An Economy Driven Resource Management Architecture for Global Computational Power Grids, presented at Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000), June 26-29, 2000, Las Vegas, USA, CSREA Press, USA, 2000.
[6]   B. Haumacher, T. Moschny, J. Reuter, and W.F. Tichy. Transparent Distributed Threads for Java, in *Proceedings of the 5th International Workshop on Java for Parallel and Distributed Computing in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS 2003)*, 2003, p. 136, Nice, France, IEEE Computer Society, ISBN-0769-5192-61.
[7]   M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7), pp. 495–518, May 2000.
[8]   M. Philippsen and M. Zenger. JavaParty - transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11), pp. 1225–1242, 1997.
[9]   R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an Efficient Java-based Grid Program ming Environment. In *Joint ACM Java Grande – ISCOPE 2002 Conference*, pages 18–27, Seattle, Washington, USA, November 2002.
[10]   O. Holder, I. Ben-Shaul, and H. Gazit, "Dynamic layout of distributed applications in FarGo." In *International Conference on Software Engineering*, pp. 163–173, 1999.

[11] T. Fahringer, "JavaSymphony: A system for development of locality-oriented distributed and parallel Java applications." In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000)*, 2000.

[12] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java application artitioning." In B. Magnusson, editor, *ECOOP 2002 -Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pp. 178–204.
Springer-Verlag, 2002.

[13] L. Chen, C. Wang, and F.C. Lau, "A grid middleware for distributed Java computing with MPI binding and process migration supports." *J. Comput. Sci. Technol.* 18, 4 (Jul. 2003), pp. 505-514, 2003

[14] Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. "Jessica2: A distributed java virtual machine with transparent thread migration support.", In *IEEE Fourth International Conference on Cluster Computing*, Chicago, USA, September 2002.

[15] Y. Aridor, M. Factor, and A. Teperman, "CJVM: A Single System Image of a JVM on a Cluster.", In *International Conference on Parallel Processing*, pp. 4–11, 1999.

[16] W. Yu and A. L. Cox. "Java/DSM: A Platform for Heterogeneous Computing.", *Concurrency - Practice and Experience*, 9(11), pp. 1213–1224, 1997.

[17] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan & Srikumar Venugopal, Alchemi: A .NET-based Grid Computing Framework and its Integration into Global Grids, Technical Report, GRIDS-TR-2003-8, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, December 2003.

[18] Gridlab Grid Application Toolkit: http://www.gridlab.org/gat

[19] R.M. Badia, J. Labarta, R. Sirvent, J.M. Perez, J.M. Cela, and R. Grima, "GRID superscalar: a programming paradigm for GRID applications", CEPBA_IBM Research Institute, UPC, Spain, January, 2004.