ALiCE: A Scalable Runtime Infrastructure for High Performance Grid Computing

Yong-Meng Teo[1,2] and Xianbing Wang[2]

[1]Department of Computer Science, National University of Singapore, Singapore 117543
{teoym, wangxb}@comp.nus.edu.sg
[2]Singapore-Massachusetts Institute of Technology Alliance, Singapore 117576

Abstract. This paper discusses a Java-based grid computing middleware, ALiCE, to facilitate the development and deployment of generic grid applications on heterogeneous shared computing resources. The ALiCE layered grid architecture comprises of a core layer that provides the basic services for control and communication within a grid. Programming template in the extensions layer provides a distributed shared-memory programming abstraction that frees the grid application developer from the intricacies of the core layer and the underlying grid system. Performance of a distributed Data Encryption Standard (DES) key search problem on two grid configurations is discussed.

1 Introduction

Grid computing [4, 8] is an emerging technology that enables the utilization of shared resources distributed across multiple administrative domains, thereby providing dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [5] in a collaborative environment. Grids can be used to provide computational, data, application, information services, and consequently, knowledge services, to the end users, which can either be a human or a process.

Grid computing projects can be hierarchically categorized as *integrated grid systems*, *application(s)-driven* efforts and *middleware* [1]. NetSolve [3] is one example of an integrated grid system. It is a client/server application designed to solve computational science problems in a wide-area distributed environment. A NetSolve client communicates, using Matlab or the Web, with the server, which can adopt any scientific package in the computational kernel. The European DataGrid [12] is a highly distinguished instance of an application-driven grid effort. Its objective is to develop a grid dedicated to the analysis of large volumes of data obtained from scientific experiments, and to establish productive collaborations between scientific groups based in different geographical locations. Middlewares developed for grid computing include Globus [6], Legion [11]. The Globus metacomputing toolkit attempts to facilitate the construction of computational grids by providing a *metacomputing abstract machine*: a set of loosely coupled basic services that can be used to implement higher-level components and applications. Globus is realigning its toolkit with the emerging OGSA grid standard [7]. Legion is

H. Jin et al. (Eds.): NPC 2004, LNCS 3222, pp. 101-109, 2004.

a metacomputing toolkit that treats all hardware and software components in the grid as objects that are able to communicate with each other through method invocations. Like Globus, Legion pledges to provide users with the vision of a single virtual machine.

This paper presents ALiCE (*Adaptive and scaLable Internet-based Computing Engine*), a grid computing *core middleware* designed for secure, reliable and efficient execution of distributed applications on any Java-compatible platform. Our main design goal is to provide developers of grid applications with a user-friendly programming environment that does away with the hassle of implementing the grid infrastructure, thus enabling them to concentrate solely on their application problems. The middleware encapsulates services for compute and data grids, resource scheduling and allocation, and facilitates application development with a straightforward programming template [15, 16].

The remainder of this paper is structured as follows. Section 2 describes the design of ALiCE including its architecture and runtime system. Section 3 discusses the ALiCE template-based distributed shared-memory programming model. Section 4 evaluates the performance of ALiCE using a key search problem. Our concluding remarks are in Section 5.

2 System Design

2.1 The Objective of ALiCE

Several projects, such as Globus and Legion, attempt to provide users with the vision of a single abstract machine for computing by the provision of core/user-level middleware encapsulating fundamental services for inter-entity communications, task scheduling and management of resources. Likewise, ALiCE is a portable middleware designed for developing and deploying general-purpose grid applications and application programming models. However, unlike Globus toolkit which is a collection of grid tools, ALiCE is a grid system.

ALiCE is designed to meet a number of design goals. ALiCE achieves *flexibility* and *scalability* through its capability to support the execution of multiple applications concurrently and the presence of multiple clients within the grid. ALiCE enables grid applications deployment on all operating systems and hardware platforms due to its implementation in the *platform independent* Java language, unlike systems such as Condor [9], which is C-based and executes only on WinNT and Unix platforms. ALiCE also offers an API to achieve *generic* runtime *infrastructure support*, allowing the deployment of any distributed application: this is a major feature a middleware has to provide, which distinguishes itself from application-driven efforts that are problem-specific, like SETI@Home [14].

2.2 Architecture

The AliCE grid architecture as shown in Figure 1 comprises of three constituent layers, *ALiCE Core*, *ALiCE Extensions* and *ALiCE Applications and Toolkits*, built

upon a set of Java technologies and operating on a grid fabric. The ALiCE system is written in Java and implemented using *Java technologies* including Sun Microsystems' Jini™ and JavaSpaces™ [13] for resource discovery services and object communications within a grid. It also works with GigaSpaces™ [10], an industrial implementation of JavaSpaces.

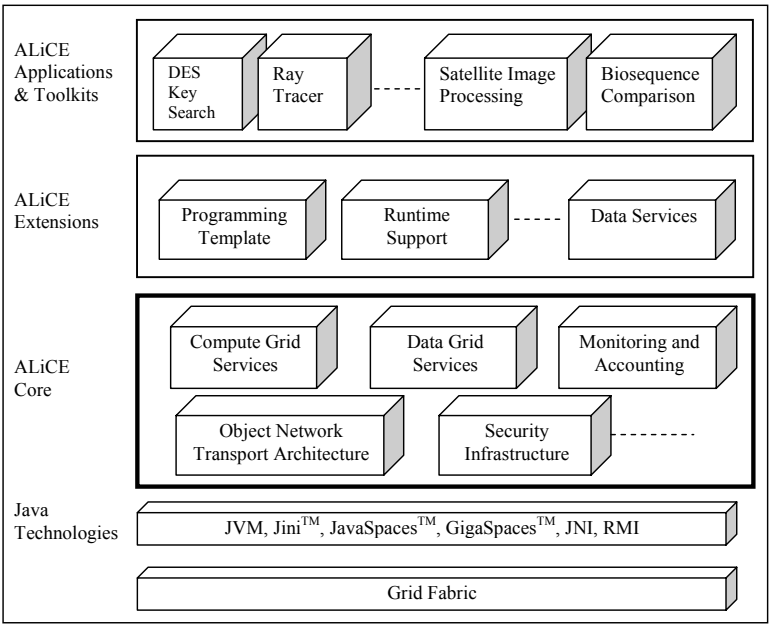

Figure 1: ALiCE Layered Grid Architecture

The *ALiCE core* layer encompasses the basic services used to develop grids. Compute Grid Services include algorithms for resource management, discovery and allocation, as well as the scheduling of compute tasks. Data Grid Services are responsible for the management of data accessed during computation, locating the target data within the grid and ensuring multiple copy updates where applicable. The security service is concerned with maintaining the confidentiality of information within each node and detecting malicious code. Object communication is performed via our Object Network Communication Architecture that coordinates the transfer of information-encapsulated objects within the grid. Besides these grid foundation services, a monitoring and accounting service is also included.

The *ALiCE extensions* layer encompasses the ALiCE runtime support infrastructure for application execution and provides the user with a distributed-shared memory programming template for developing grid applications at an abstract level. Runtime support modules are provided for difficult programming languages and machine platforms. Advanced data services are also introduced to enable users to customize the means in which their application will handle data, and this is especially useful in problems that work on uniquely formatted data, such as data retrieved from

specialized databases and in the physical and life sciences. This is the layer that application developers will work with.

The *ALiCE applications and toolkits* layer encompasses the various grid applications and programming models that are developed using ALiCE programming template and it is the only layer visible to ALiCE application users.

2.3 Runtime System

Figure 2 shows ALiCE runtime system. It adopts a three-tiered architecture, comprising of three main types of entities: *consumer*, *producer* and *resource broker*, as described in the following:

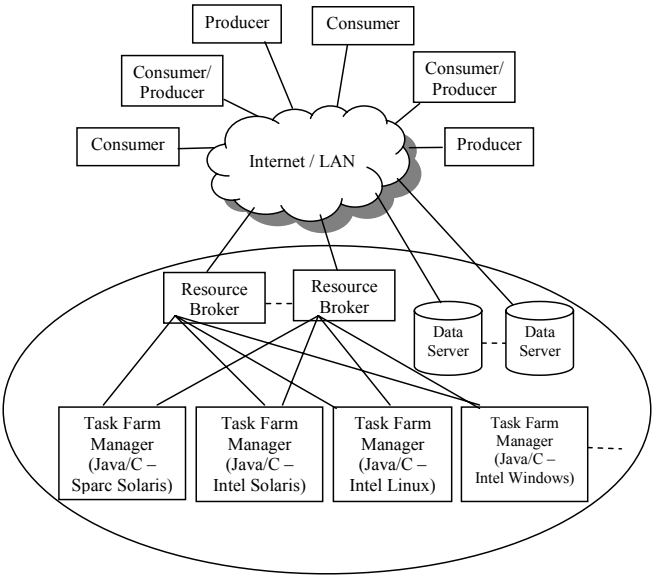

Figure 2: ALiCE Runtime System

- ***Consumer***. This submits applications to the ALiCE grid system. It can be any machine within the grid running the ALiCE consumer/producer components. It is responsible for collecting results for the current application run, returned by the tasks executed at the producers, and is also the point from which new protocols and new runtime supports can be added to the grid system.
- ***Resource broker***. This is the core of the grid system and deals with resource and process management. It has a *scheduler* that performs both *application* and *task* scheduling. Application scheduling helps to ensure that each ALiCE application is able to complete execution in a reasonable turnaround time, and is not constrained by the workload in the grid where multiple applications can execute concurrently. Task scheduling coordinates the dissemination of compute tasks, thereby controlling the utilization of the producers. The default task scheduling algorithm adopted in ALiCE is *eager scheduling* [2].

- ***Producer***. This is run on a machine that volunteers its cycles to run ALiCE applications. It receives tasks from a resource broker in the form of serialized live objects, dynamically loads the objects and executes the encapsulated tasks. The result of each task is returned to the consumer that submitted the application. A producer and a consumer can be run concurrently on the same machine.
- ***Task Farm Manager***. ALiCE applications are initiated by the Task Farm Manager and the tasks generated are then scheduled by the resource broker and executed by the producers. The task farm manager is separated from the resource broker for two principal reasons. Firstly, ALiCE supports non-Java applications that are usually platform-dependent, and the resource broker may not be situated on a suitable platform to run the task generation codes of these applications. Secondly, for reasons of security and fault tolerant the execution of alien code submitted by consumers is isolated from the resource broker.

3 Grid Programming

ALiCE adopts the *TaskGenerator-ResultCollector* programming model. This model comprises of four main components: *TaskGenerator*, *Task*, *Result* and *ResultCollector*. The consumer first submits the application to the grid system. The *TaskGenerator* running at a task farm manager machine generates a pool of *Task*s belonging to the application. These *Task*s are then scheduled for execution by the resource broker and the producers download the tasks from the task pool. The results of the individual executions at the producers are returned to the resource broker as *Result* object. The *ResultCollector*, initiated at the consumer to support visualization and monitoring of data collects all *Result* objects from the resource broker.

The template abstracts methods for generating tasks and retrieving results in ALiCE, leaving the programmers with only the task of filling in the task specifications. The Java classes comprising the ALiCE programming template are:

a. *TaskGenerator*. This is run on a task farm manager machine and allows tasks to be generated for scheduling by the resource broker. It provides a method process that generates tasks for the application. The programmer merely needs to specify the circumstances under which tasks are to be generated in the main method.
b. *Task*. This is run on a producer machine, and it specifies the parallel execution routine at the producer. The programmer has to fill in only the execute method with the task execution routine.
c. *Result*. This models a result object that is returned from the execution of a task. It is a generic object, and can contain as many user-specified attributes and methods, thus permitting the representation of results in the form of any data structure that are serializable.
d. *ResultCollector*. This is run on a consumer machine, and handles user data input for an application and the visualization of results thereafter. It provides a method collectResult that retrieves a *Result* object from the resource broker. The programmer has to specify the visualization components and control in the collect method.

4 Performance Evaluation

We have developed several distributed applications using ALiCE. These include life science applications such as biosequence comparison and progressive Multiple Sequence Alignment [16], satellite image processing [15], distributed equation solver, etc. In this paper, we present the results of the DES (*Data Encryption Standard*) key search [18]. DES key search is a mathematical problem, involving the use of a brute force method to identify a selected encryption key in a given key space. A DES key consists of 56 bits that are randomly generated for searching, and 8 bits for error detection. In the algorithm, a randomly selected key, *K*, is used to encrypt a known string into a ciphertext. To identify *K*, every key in the key space is used to encrypt the same known string. If the encrypted string for a certain key matches with the ciphertext, then the algorithm converges and the value of *K* is returned. This problem requires immense computational power as it involves exhaustive search in a potentially huge key space.

The test environment consists of a homogeneous cluster and a heterogeneous cluster with all nodes running RedHat Linux. The 64-node homogeneous cluster (*Cluster I*) consists of dual processors Intel Xeon 1.4GHz processors with 1GB of memory. The nodes are connected by a Myrinet network. The 24-node heterogeneous cluster (*Cluster II*) consists of sixteen nodes Pentium II 400MHz with 256MB of RAM, and eight nodes Pentium III 866MHz with 256MB of RAM. These nodes are connected via 100Mbps Ethernet switch.

Our performance metric is the execution time to search the *entire* key space. The sequential execution time grows exponentially with increasing key sizes. The DES key problem can be partitioned into varying number of tasks with a task size measured by the number of keys and its execution time can be estimated using the time from the sequential run. Table 1 shows the task characteristics for varying task sizes and problem sizes. The table was used to select an appropriate task size for the experiments to be carried out in the two grid configurations.

task size (keys)	32-bit Key			36-bit Key		40-bit Key	
	no. of tasks	Est. Time/Task (secs)		no. of tasks	est. time/task (secs)	no. of tasks	est. time/task (secs)
		cluster I	cluster II		cluster I		cluster I
5,000,000	859	20.8	32.9	13,744	20.8	219,902	430.3
10,000,000	429	41.1	65.4	6,872	43.4	109,951	862.4
30,000,000	143	122.9	196.6	2,291	127.8	36,650	2587.7
50,000,000	86	201.9	322.0	1,374	211.4	21,990	4314.3
100,000,000	43	395.4	641.2	687	420.1	10,995	8629.5

Table 1: Estimated Task Execution Times for Varying Task Sizes

For our experiments conducted, we selected a task size of 50 million keys per task and a problem size of 36-bit keys for Cluster I and 32-bit keys for Cluster II. Table 2 shows the results for 4 to 32 producer nodes. The execution time for key search reduces significantly with increasing number of nodes, resulting in greater speedup.

No. of Producers	Cluster I (36-bit Key)	Cluster II (32-bit Key)
1 (Est. Sequential)	78 hr 23 min	8 hr 43 min
4	23 hr 36 min	3 hr 43 min
8	11 hr 6 min	2 hr 7 min
10	8 hr 34 min	1 hr 42 min
12	7 hr 21 min	1 hr 26 min
16	5 hr 11 min	1 hr 7 min
32	2 hr 29 min	-

Table 2: Execution Time for Varying Number of Producer Nodes

We define *speedup* as T_s/T_p, where T_s is the execution time of the sequential program and T_p is the execution time of the derived parallel program on p processors. As shown in Figure 5, a speedup of approximately 32 is attained for key size 36-bits on Cluster I and 8 for 32-bits on Cluster II. We consider these results highly encouraging, although the performance of key search needs to be further evaluated with more key space sizes and nodes. The effects of using other scheduling algorithms in the resource broker must also be studied, as it may result in different overheads to the execution time.

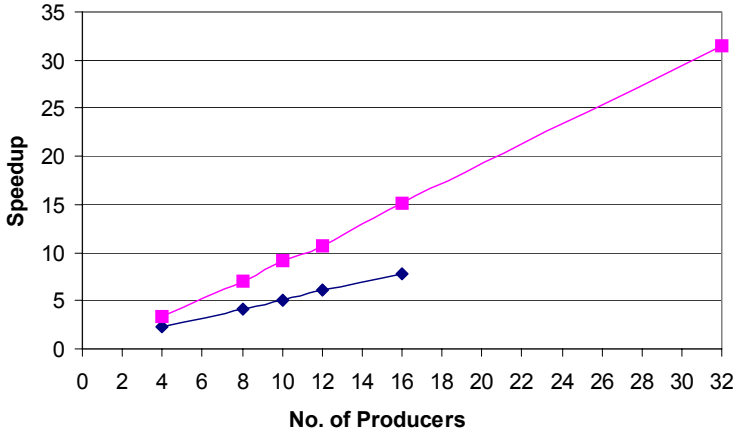

Figure 3: Speedup vs Varying Number of Producers

5 Conclusions and Further Works

We discussed the design and implementation of the Java-based ALiCE grid system. The runtime system comprises of consumers, producers and resource broker. Parallel

grid applications are written using programming template that supports the distributed-shared memory programming model. We presented the performance of ALiCE using the DES key search problem. The result shows that a homogeneous cluster yields greater speedup than on a heterogeneous cluster for the same task size. A homogeneous cluster generally has a better load balance than a heterogeneous cluster which is made up of different platforms and capabilities.

Much work still needs to be done to transform ALiCE into a comprehensive grid computing infrastructure. We are in the process of integrating new resource scheduling techniques and load-balancing mechanisms into the ALiCE core layer to reduce the overhead in running applications [17]. Task migration, pre-emption and check-pointing mechanisms are being incorporated to improve the reliability and fault-tolerance ability of the system.

References

1. Baker, M., Buyya, R. and Laforenza, D., Grids and Grid Technologies for Wide-Area Distributed Computing, *International Journal of Software: Practice and Experience (SPE)*, 32(15), Wiley Press, USA, November 2002.
2. Baratloo. A, Karaul. M, Kedem. Z and Wyckoff. P, Charlotte: Metacomputing on the Web, *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.
3. Casanova, H. and Dongarra, J., NetSolve: A Network Server for Solving Computational Science Problems, *International Journal of Supercomputing Applications and High Performance Computing*, 11(3), 1997.
4. De Roure, D., Baker, M. A., Jennings, N. R. and Shadbolt, N. R., The Evolution of the Grid, Research Agenda, UK National eScience Center, 2002.
5. Foster, I., Computational Grids, Morgan Kaufmann Publishers, 1998.
6. Foster. I and Kesselman. C, Globus: A Metacomputing Infrastructure Toolkit, *International Journal of Supercomputing Applications*, 11(2), pp 115-128, 1997.
7. Foster, I., Kesselman, C., Nick, J. M. and Tuecke, S., The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, *Proceedings of CGF4*, February 2002, http://www.globus.org/research/papers/ogsa.pdf.
8. Foster, I., Kesselman, C. and Tuecke, S., The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International Journal of Supercomputer Applications*, 15(3), 2001.
9. Frey, J., Tannenbaum, T., Foster, I., Livny, M. and Tuecke, S., Condor-G: A Computation Management Agent for Multi-Institutional Grids, *Journal of Cluster Computing*, 5, pp. 237-246, 2002.
10. GigaSpaces Platform White Paper, GigaSpaces Technologies, Ltd., February 2002.
11. Grimshaw, A. and Wulf, W., The Legion Vision of a Worldwide Virtual Computer, *Communications of the ACM*, 40(1), January 1997.
12. Hoschek, W., Jaen-Martinez, J., Samar, A., Stockinger, H. and Stockinger, K., Data Management in an International Data Grid Project, *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (GRID2000)*, Bangalore, India, pp. 17-20, December 2000.
13. Hupfer, S., The Nuts and Bolts of Compiling and Running JavaSpaces Programs, Java Developer Connection, Sun Microsystems, Inc., 2000.
14. SETI@Home: Search for Extraterrestrial Intelligence at Home, http://setiathome.ssl.berkeley.edu.

15. Teo., Y.M., S.C. Tay and J.P. Gozalijo, Geo-rectification of Satellite Images using Grid Computing, Proceedings of the International Parallel & Distributed Processing Symposium, IEEE Computer Society Press, Nice, France, April 2003.
16. Teo Y.M. and Ng Y.K., Progressive Multiple Biosequence Alignments on the ALiCE Grid, *Proceeding of the 6th International Conference on High Performance Computing for Computational Science,* Springer Lecture Notes in Computer Science Series, xx, Spain, June 28-30, 2004 (accepted for publication).
17. Teo Y.M., X. Wang, J.P. Gozali, A Compensation-based Scheduling Scheme for Grid Computing, *Proceedings of the 7th International Conference on High Performance Computing,* IEEE Computer Society Press, Tokyo, Japan, July 2004.
18. Wiener, M., Efficient DES Key Search, Practical Cryptography for Data Internetworks, William Stallings, IEEE Computer Society Press, pp. 31-79, 1996.