Y.M. Teo, Y. Chen and X.B. Wang, *On Grid Programming and MATLAB*G*, Proceedings of 3rd International Conference on Grid and Cooperative Computing, pp. xx, Springer Verlag Lecture Notes in Computer Science, Wuhan, China, October 2004 (submitted).

On Grid Programming and MATLAB*G

Y. M. Teo[1,2], Y. Chen[2] and X.B. Wang[2]

[1]*Department of Computer Science, National University of Singapore, Singapore 117543*
[2]*Singapore-Massachusetts Institute of Technology Alliance, 4 Engineering Drive 3, Singapore 117576*
email: teoym@comp.nus.edu.sg

Abstract

*The main goal of grid programming is the study of programming models, tools and methods that support the effective development of algorithms and applications on grid. This paper discusses the design and implementation of ALiCE object-oriented grid programming template (AOPT). ALiCE is a Java-based grid computing middleware to facilitate the development and deployment of generic grid applications on heterogeneous shared computing resources. The programming template provides a distributed shared-memory programming abstraction based on JavaSpaces that frees the grid application developer from the intricacies of the core layer and the underlying grid system. AOPT is designed for developing grid applications and as a programming tool for grid-enabling domain specific software applications such as MATLAB. In this paper, we discuss the design and implementation of MATLAB*G, a grid-enabled MATLAB using AOPT. The performance results indicate that for large matrix sizes MATLAB*G can be a faster alternative to sequential MATLAB.*

1 Introduction

Grid computing [7, 12] 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 [10] in a collaborative environment. These resources can include supercomputers, storage systems, data sources and special classes of devices. Clustering and using them as a single unified resource forms a networked virtual supercomputer [11] is popularly known as a *computational grid* [10]. 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.

The main goal of grid programming is the study of programming models, tools and methods that support the effective development of portable and high-performance algorithms and applications on grid environments [20]. Grid programming will require capabilities and properties beyond that of simple sequential programming or even parallel and distributed programming. Besides orchestrating simple operations over private data structures, or orchestrating multiple operations over shared or distributed data structures, a grid programmer will have to manage a computation in an environment that is typically open-ended, heterogeneous and dynamic. A programming model can be present in many different forms, e.g., a language, a library API, or a tool with extensible functionality. The most successful programming models will enable both high-performance and the flexible composition and management of resources.

Grid applications tend to be heterogeneous and dynamic, i.e., they will run on different types of resources whose configuration may change during run-time. These dynamic configurations could be motivated by changes in the environment from any available grid resources. Regardless of their cause, a programming model or tool need to present the heterogeneous resources as a common “look-and-feel” to the programmer; hiding their differences while allowing the programmer some control over

each resource type if necessary. To reduce the complexity of grid programming, such transparency needs to be provided by the run-time system. We discuss the design and implementation of MATLAB*G, a grid-based MATLAB developed using AOPT.

In [20], Craig and Domenico have reviewed the grid programming models on current tools, issues and directions. Eight grid programming models have been reviewed and its shortcomings are discussed. In this paper, we design and implement an object-oriented grid programming template based on the distributed shared-memory model, JavaSpaces. We focus on how to efficiently develop portable and high-performance grid applications.

Sun JavaSpaces [17] is a Java-based implementation of the Linda tuplespace concept, in which tuples are represented as serialized objects. The use of Java allows heterogeneous clients and servers to interoperate, regardless of their processor architectures and operating systems. The model used by JavaSpaces views an application as a collection of processes communicating between them by putting and getting objects into one or more *spaces*. A *space* is a shared and persistent object repository that is accessible via network. The processes use the repository as an exchange mechanism to get coordinated, instead of communicating directly with each other. The main operations that processes can do with a *space* are to *put*, *take* and *read* objects. A programmer that wants to build a space-based application should design *distributed data structures* as a set of objects that are stored in one or more *spaces*. The new approach that the JavaSpaces programming model gives to the programmer makes building distributed applications much easier, even when dealing with such dynamic, environments.

Our propose ALiCE Object-oriented grid Programming Template (AOPT) is implemented on ALiCE (*Adaptive 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 [27, 29]. Our main design goal is to grid application developers with a user-friendly programming environment that is transparent of low-level grid infrastructure details, thus enabling them to concentrate solely on the application problems. The middleware encapsulates services for compute and data grids, resource scheduling and allocation, and facilitates application development with a straightforward programming template. Using AOPT, we have demonstrated the ease of programming grid applications [28, 30, 31, 32].

This paper focuses on the use of AOPT as a system programming tool to grid-enabled the domain-specific application package called MATLAB. Performance results indicate that for large matrix sizes MATLAB*G can be a faster alternative to sequential MATLAB. The remainder of this paper is structured as follows. Section 2 introduces ALiCE. Section 3 presents the ALiCE template-based distributed shared-memory programming model. Section 4 uses the programming template to implement MATLAB*G. Section 5 presents the performance evaluation of MATLAB*G. Our concluding remarks are in Section 6.

2 System Design

2.1 Architecture

Several projects, such as Globus [11] and Legion [21], 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 [29]. However, unlike Globus toolkit which is a collection of grid tools, ALiCE is a grid system.

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, which encompasses the physical hardware components and networks within the grid. The ALiCE system is written in Java and implemented using *Java technologies* including Sun Microsystems' Jini™ and JavaSpaces™ [15] for resource discovery services and object communications within a grid. To support the execution of applications regardless of their developmental language, ALiCE uses Java Native Interface (JNI) to enable the runtime infrastructure to invoke non-Java code.

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 (ONTA) that coordinates the transfer of information-encapsulated objects within the grid. Besides these grid foundation services, a monitoring and accounting service is also included [29].

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 different 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 [28, 29].

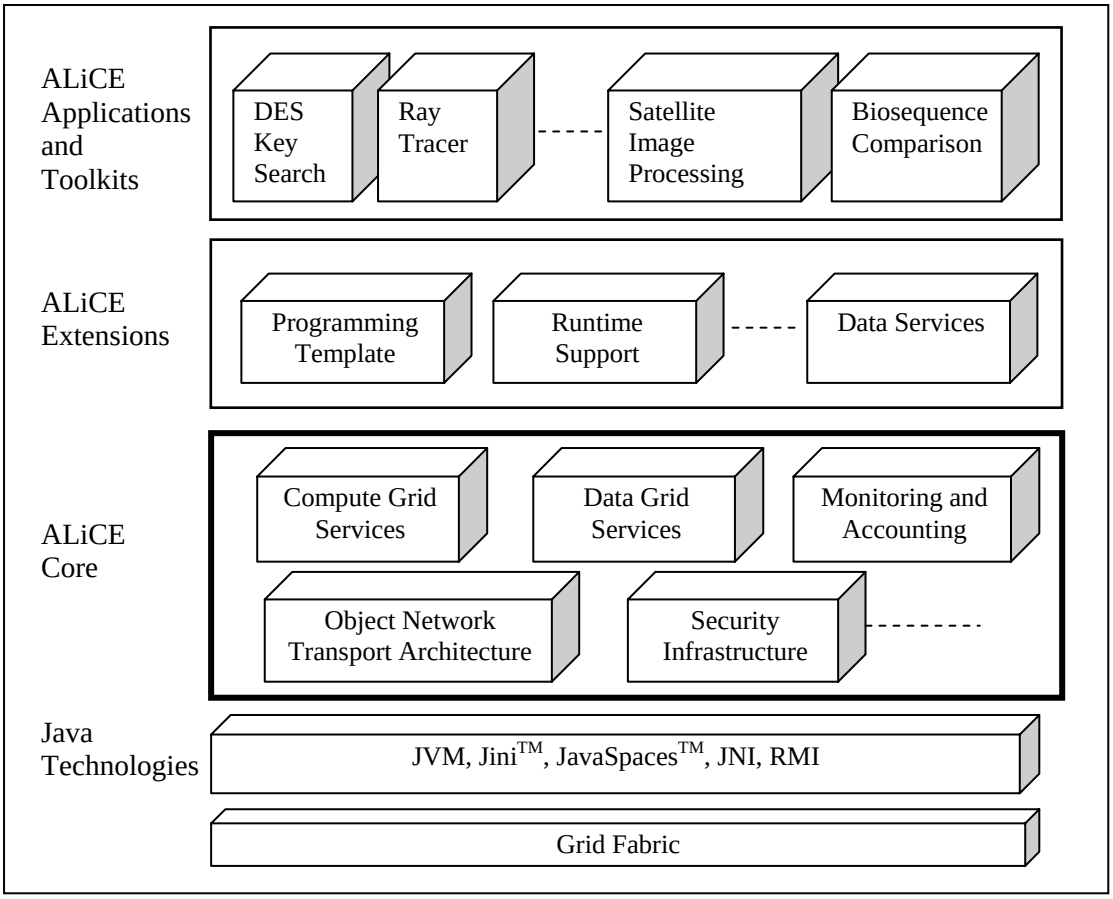

Figure 1: ALiCE Grid Architecture

2.2 Runtime System

The ALiCE runtime system, illustrated in Figure 2, is an integration of the Compute Grid Services and ONTA components from the ALiCE Core layer in the grid architecture. It adopts a three-tiered architecture, and consists of the following four main components:

- ***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]. In addition, there are some objective requirements imposed on ALiCE, since one of its goals is to support execution of applications implemented in programming languages other than Java. These applications may be platform and library dependent. The scheduler must therefore select, amongst the producers in the grid, one that runs the most appropriate platform to execute a given application.
- ***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. Each task farm manager runs either Java code or code compiled for the platform that the task farm manager offers.

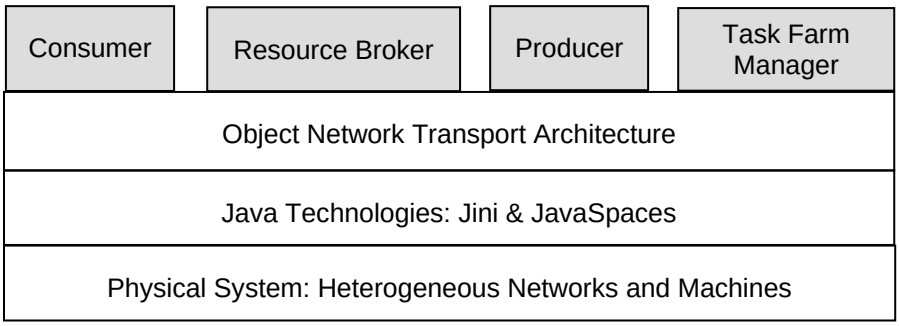

Figure 2: ALiCE Runtime System Framework

In a typical scenario, a user launches an ALiCE application at a consumer, which then submits the application codes to a resource broker in the system. The resource broker directs an application to an appropriate task farm manager that supports the required programming language and platform. The task farm manager initiates the application and creates a pool of tasks. Task references are returned to the resource broker, which schedules the tasks for execution on producers. Results of task execution are then returned to the consumer for visualization. Connections between the system entities are established over a LAN if the environment is a cluster grid or an Intranet, and through the Internet if the grid system is deployed over a WAN.

2.3 Implementation

ALiCE is scalable and comprises of modular components developed using ubiquitous and easy-to-use object-based Java technologies, including Jini and JavaSpaces [15], thereby providing for *cross-platform portability*, *extensibility* and *scalability*. Each ALiCE producer runs a copy of the JVM, allowing different platforms in the grid to share executables.

Jini defines a runtime infrastructure that unifies all JVMs into a single virtual network, enabling homogeneous hardware devices to plug in to form *decentralized communities*. JavaSpaces, a special service of Jini, is a simple, expressive, and powerful technology with the goal of reducing development time in building distributed applications. All processes are loosely coupled across the network, communicating and synchronizing their activities using a persistent object store called a *space*. All ALiCE entities communicate through JavaSpaces. To our knowledge, ALiCE is the *first* grid-computing project that is developed using Sun's Java-Jini and JavaSpaces.

ALiCE provides an alternative communications platform using GigaSpaces [14]. GigaSpaces Synchronization and Coordination Platform is a software infrastructure for information collaboration platform for Enterprise Distributed Applications and Web Services. The major difference between JavaSpaces and GigaSpaces is that the former provides a logical distributed-shared memory [16] but the latter implements distributed-shared memory by coupling together several spaces hosted at different machines. Our tests show that using GigaSpaces results in better performance and reliability than JavaSpaces.

3 Grid Programming

Grid environments are characteristically distributed and dynamic [19]. ALiCE sets out to provide an effective programming model to facilitate the development of grid applications and higher-level specialized programming models. In the ALiCE paradigm, large computations are decomposed into smaller tasks that are then distributed among producers in the network to exploit parallelism as best as possible to achieve a reasonable amount of speedup.

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 in the form of a .jar file encapsulating the application codes. The *TaskGenerator* running at a task farm manager machine generates a pool of *Task*s belonging to the application. Subsequently, these *Task*s are scheduled for executing 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. For batch job, result objects are collected at the resource broker.

Parallel applications development are written using ALiCE programming template. The template allows the programmers to transparently exploit the distributed nature of the ALiCE grid, i.e., without prior knowledge of the underlying technologies for communications, dynamic code linking, etc. 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. Figure 3 shows the ALiCE programming template.

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.

TaskGenerator Template

```
import alice.consumer.*;
import alice.data.*;
public class TASKGEN_CLASSNAME extends TaskGenerator {
   public TASKGEN_CLASSNAME() {}
   public void init() {
      //Place your initialisation code here
   }

   /* Main method - entry point */
   public void main(String args[]) {
     // This is where the tasks are generated, usually in a loop

     // This should be called for each task
     TASK_CLASSNAME t = new TASK_CLASSNAME();
     process(t);

     // To open a data file, read and write from/to it
     DataFile f = Data.openFile("file_name",this);
     READ_BUFF = f.read(POSITION, LENGTH);
    f.write( WRITE_BUFF, POSITION, LENGTH);

     // To send/receive an object
     OBJECT_CLASSNAME obj = new  OBJECT_CLASSNAME();
     sendObject(obj, "snd_str_id");
     OBJECT_CLASSNAME rcvObj = (OBJECT_CLASSNAME)
                                    requestObject("rcv_str_id");

     // To receive a string message from the result collector:
    String msg = getStringMessage();
   }
}
```

Task Template

```
import alice.consumer.*;
import java.io.*;
public class TASK_CLASSNAME extends Task {
  // Place variables here
  public TASK_CLASSNAME () {
  }

  public Object execute () {
    // This is where you do your computations. The results can be any kind of
    // objects

    // You can generate and send a new task to be produced
    O_TASK_CLASSNAME t = new O_TASK_CLASSNAME();
    process(t);

    // To open a data file, read and write from/to it
    DataFile f = Data.openFile("file_name",this);
    READ_BUFF = f.read(POSITION, LENGTH);
    f.write( WRITE_BUFF, POSITION, LENGTH);

    // To send/receive an object
    OBJECT_CLASSNAME obj = new OBJECT_CLASSNAME();
    sendObject(obj, "snd_str_id");
    OBJECT_CLASSNAME rcvObj =(OBJECT_CLASSNAME)
                                    requestObject("rcv_str_id");
  }
}
```

Result Template

```
import java.io.*;

public class MyResult implements Serializable {
   public DATA_TYPE var;
   public MyResult() {
      var=NULL;
   }
}
```

ResultCollector Template

```
import alice.result.*;
public class RESCOL_CLASSNAME extends ResultCollector {
   // Place Variables Here

   public RESCOL_CLASSNAME() {
   }

   public void collect() {
     // Place here the result collection and processing code to obtain
     // number of results ready call
     int resReady = getResultsNoReady()

      // To get a new result call
     RES_CLASSNAME res = (RES_CLASSNAME)collectResult();
   }
}
```

Figure 3: ALiCE Programming Template

4 MATLAB*G

MATLAB is popular mathematical software that provides an easy-to-use interface for various science computations. Computation intensive MATLAB applications can benefit from faster execution if parallelism is provided. With the increasing popularity of distributed computing technology, up to now, at least twenty-seven parallel MATLABs are available [6].

In this paper we present the design, implementation and experimental results of MATLAB*G, a grid-based MATLAB on the ALiCE Grid, which exploits distributed matrix computation using task parallelism and job parallelism. Firstly, we introduce related existing parallel MATLABs which are classified into different categories according to two criteria: First, whether they provide implicit or explicit parallelism; second, the method used for inter-processor communication.

4.1 Implicit Parallelism vs. Explicit Parallelism

In order to execute a program which exploits parallelism, the programming language must supply the

means to identify parallelism, to start and stop parallel executions, and to coordinate the parallel executions. Thus from the programming language level, the approaches to parallel processing can be classified into implicit parallelism and explicit parallelism [13]:

- *Implicit parallelism* allows programmers to write their programs without any concern about the exploitation of parallelism. Exploitation of parallelism is instead automatically performed by the compiler or the runtime system. Parallel MATLABs in this category include RTExpress [26], CONLAB Compiler[4], and MATCH[22]. All of these parallel MATLABs take MATLAB scripts and compile them into executable code. The advantage is that the parallelism is transparent to the programmer. However, extracting parallelism implicitly requires much effort for the system developer.
- *Explicit parallelism* is characterized by the presence of explicit constructs in the programming language, aimed at describing the way in which the parallel computation will take place. Most parallel MATLABs use explicit parallelism, like MATLAB*P [5], MATmarks [23], and DP-Toolbox [8]. The main advantage of explicit paralleism is its considerable flexibility, which allows the user to code a wide variety of patterns of execution. However the management of the parallelism, a quite complex task, is left to the programmer.

MATLAB*P is an explicitly parallel MATLAB designed at MIT. MATLAB*P handles communication and synchronization for the user and requires the user to explicitly indicate the matrices which are to be distributed. MATLAB*G is also an explicitly parallel MATLAB and is similar to MATLAB*P, in that it handles the communication and synchronization details for the user. However, while users are not required to indicate the matrices to be distributed, they have to explicitly specify the MATLAB computations to be parallelized.

4.2 Inter-processor Communication

In designing a parallel system, processors must have the ability to communicate with each other in order to cooperatively complete a task. There are two methods of inter-processor communication, each suitable for different system architectures:

The first is *Distributed Memory Architectures.* It employs a scheme in which each processor has its own memory module. Each component is connected with a high-speed communications network. Processors communicate with each other over the network. Well-known packages such as MPI [24] provide a message passing interface between machines. Most parallel MATLABs are built upon distributed memory architecture, e.g. MATLAB*P, Cornell Multitasking Toolbox for MATLAB, and Distributed and Parallel Application Toolbox, etc. One advantage of these parallel MATLABs is that MPI and PVM are mature standards which have been available for several years and offers a high degree of functionality. However, almost all of these parallel MATLABs exploit standard message passing interface, which means they can only run on homogenous clusters.

The second is *Distributed Shared Memory* systems which have two main architectures [9]:

- *Shared Virtual Memory (SVM)* systems share a single address space, thereby allowing processor communication through variables stored in the space. For example, MAT*marks*, an environment that allows users to run several MATLAB programs in parallel using the shared memory programming style is built on top of TreadMarks, a virtual SVM which provides a global shared address space across the different machines on a cluster. The environment extends the MATLAB language with several primitives to enable shared variables and synchronization primitives.
- *Object-based Distributed Shared Memory (DSM):* Processes on multiple machines share an abstract space filled with shared objects. The location and management of the objects is handled automatically by the runtime system. Any process can invoke any object's methods, regardless of where the process and object are located. It is the job of the operating system and runtime system to make the act of invoking work no matter where the process and the object are located. DSM has a few advantages over SVM: (i) it is more modular and more flexible because accesses are controlled, and (ii) synchronization and access can be integrated together cleanly.

MATLAB*G is currently the only parallel MATLAB built on object-based DSM. A shared memory interface is more desirable than a message-passing interface from the application programmer's viewpoint, as it allows the programmer to focus more on algorithmic development rather than on managing communication. But such a parallel MATLAB depends on another system

providing shared memory upon different machines. Furthermore, whether a parallel MATLAB can be run in a heterogeneous environment depends on whether the DSM supports heterogeneous machines.

4.3 System Design

In MATLAB, a normal matrix addition can be performed as in Figure 4. The first line creates a 10-by-10 matrix A, The second lines creates a 10-by-10 matrix B. The third line creates a 10-by-10 matrix C, and lets it have the value: A+B. To parallelize the matrix addition, a user can write a MATLAB*G program as shown in Figure 5. The third line creates a 10-by-10 matrix C, performs parallel matrix addition between A and B, and returns the result to C. From the user's point of view, these two programs are equivalent because after execution the resulting matrix C in both programs has the same value in both programs. However, the executions of these two programs are quite different: the first program is executed purely inside MATLAB environment, but the second exploits parallelism provided by MATLAB*G.

```
1: A=randn(10,10);
2: B=randn(10,10);
3: C=plus (A, B);
```

Figure 4 MATLAB Matrix Addition

```
1: A=randn(10,10);
2: B=randn(10,10);
3: C=mm('plus', 2, A, B);
```

Figure 5 MATLAB*G Matrix Addition

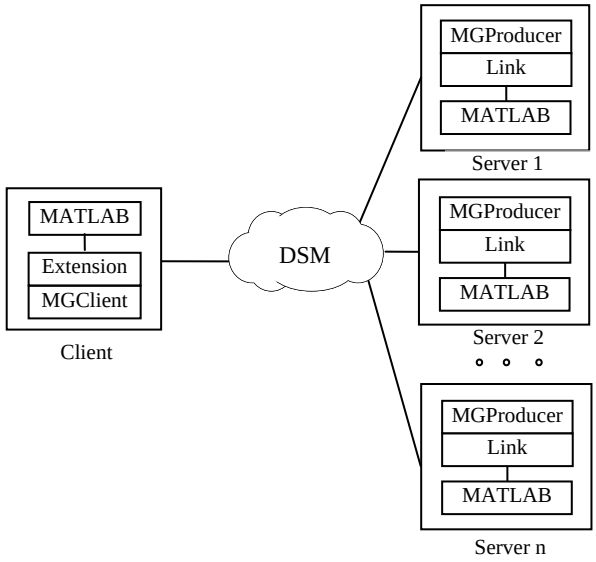

Figure 6. MATLAB*G Client-Server Architecture

To achieve parallelism, MATLAB*G exploits a client-server model using Object-based Distributed-Shared Memory. User submits job interactively from MATLAB environment. The client gets the job, divides it into a number of tasks, sends tasks into the DSM, and polls DSM for the result. A server always tries to get a task from DSM. After receiving a task, the server processes it, and returns the result to DSM. On the client side, after getting all results from servers, it assembles them into a complete result and returns it to the user. The system architecture is shown in Figure 6.

4.3.1 The Client

As shown in Figure 3, the client side consists of two components: *Extension* and *MGClient*. A more detailed diagram for the MATLAB*G client architecture is shown in Figure 7.

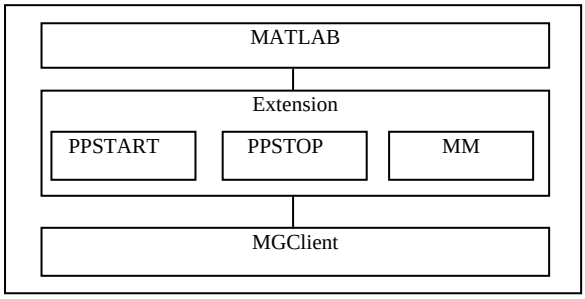

Figure 7. MATLAB*G Client

Extension includes a few MATLAB M files. It provides user interfaces for parallelism and links MATLAB with *MGClient.:*

a. *ppstart:* This is a MATLAB function introduced by MATLAB*G Extension. When the user calls *ppstop*(*n*), *n* servers are initialized and reserved for future computations.

b. *ppstop:* This function releases the reservations by a prior *ppstart*.

c. *mm:* This function lets the user assign a parallel job. The syntax of *mm* is: *A=mm*(*'fname', tasknum, matrices*). *fname* is the computation that the user wants to execute, *matrices* are the arguments for this computation and *tasknum* is the task number specified by the user. The user can decide the task granularity according to the complexity of the computation and the size of matrices. We anticipate that in the next version, the *tasknum* argument will be removed and the task number will be generated by the system automatically according to certain algorithms. Finally, *A* is the result of computation.

Another component, *MGClient*, includes a number of Java classes, and provides two main functionalities:

a. Communicate with all the servers. *MGClient* communicates with the servers through DSM; it does not need to know their locations.

b. Distribute tasks and assemble results. According to the user's input program, *MGClient* generates a number of tasks, which can be executed on the servers. *MGClient* also has to assemble all results from servers into one complete and correct result and return it to the user.

MGClient can be invoked only by *Extensions*, which makes the command set simpler. Pseudocode for *MGClient* is shown in Figure 8 below:

```
Switch of command passed in by Extension:
Case: ppstart
   Send ppstart into DSM;
Case: ppstop
   Send ppstop into DSM;
Case: mm
   Partition matrices;
   Marshal message into a number of tasks;
   Send tasks into DSM;
   Wait for result by polling DSM;
   Return result;
```

Figure 8. *MGClient* Pseudocode

Besides *Extension* and *MGClient*, a running instance of MATLAB is also required on the client side. This MATLAB session provides a programming environment to the user and thus lets the user invoke function calls in *Extension*.

4.3.2 The Servers

As communication latency is quite unpredictable on a grid system, it would be costly to pass data frequently among the compute nodes. Thus currently only embarrassingly parallel mode of computation is supported, whereby each server receives a work package, performs computation without coordination with other servers, and sends results back to the client. The Server consists of two main components: *MGProducer* and *Link*.

MGProducer runs on a server and waits for tasks from DSM. On receiving a *ppstart* from DSM, *MGProducer* starts a MATLAB session at the backend through *Link*. Similarly, on receiving a *ppstop*, *MGProducer* terminates the MATLAB session. Upon receiving a computation task, *MGProducer* performs calculation and sends the result back to DSM. The pseudocode for *MGProducer* is shown in Figure 9.

```
Loop Forever
  If take PPSTART message fails, go to 1
  Start MATLAB
  Acknowledge the client
  Loop Forever
    Wait for message from the client
    If message is PPSTOP, then
       Stops MATLAB
       Break
    Else
       Process message
       Acknowledge completion of task and send result to the client
    End If
  End Loop
End Loop
```

Figure 9. *MGProducer* Pseudocode

Link is another component on the server. It is used by *MGProducer* to start a MATLAB session, stop a MATLAB session, and execute MATLAB programs. To implement *Link*, we make use of an existing Java interface to the MATLAB engine called JMatLink [18].

4.3.3 DSM

A tuplespace is a shared datastore for simple list data structures (tuples) [3]. A simple model is used to access the tuplespace, usually consisting of the operations *write*, *take* and *read*. A tuplespace provides DSM if every data inside it is an object. In MATLAB*G, communication between processors is handled through a tuplespace where processors post and read objects. Submatrices are deposited into space for server nodes to retrieve. The server nodes then perform computations on submatrices and return the results back to space.

4.4 System Implementation

MATLAB*G is written in Java and implemented on the ALiCE Grid.

4.4.1 Mapping MATLAB*G onto ALiCE

The mapping from MATLAB*G onto ALiCE is illustrated in Figure 10. Shaded boxes are MATLAB*G components. A user submits a job through ALiCE Consumer. In response to the submission of a job, the ALiCE Resource Broker will instantiate a MATLAB*G Client, and ALiCE Producers will instantiate MATLAB*G Servers.

4.4.2 ALiCE Program Generated by MATLAB*G

ALiCE does not provide any interface to allow a user to run a MATLAB*G client directly on the Task Manager or run a MATLAB*G server on a Producer. An application has to be submitted in the form of an ALiCE Program through the Consumer interface.

An ALiCE program template consists of following elements: *Task*, which runs on the Producer; *TaskGenerator*, which runs on the *Resource Broker*; and *ResultCollector* which runs on the Consumer to collect the results obtained from the execution of tasks generated by the Task Manager. Thus our job is just to add MATLAB*G code into proper templates to generate customized ALiCE program elements.

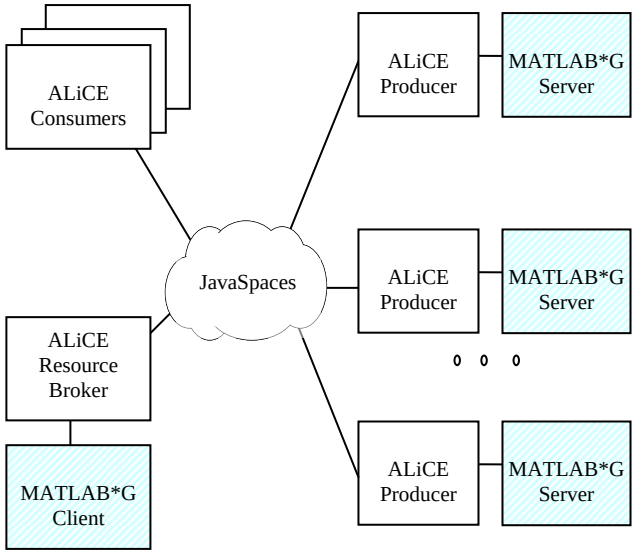

Figure 10. Mapping between MATLAB*G and ALiCE components

1) *MGTaskGenerator*

Besides the client side code, the user's MATLAB program is also embedded in the Task Generator template to create *MGTaskGenerator*. *MGTaskGenerator* first starts a MATLAB session, and then initiates *n* tasks by issuing command *ppstart(n)* to MATLAB. It then asks MATLAB to run the user's MATLAB programs. When finished, it issues a *ppstop* command to terminate tasks. *MGTaskGenerator* sends output it receives from MATLAB to *MGResultCollector* as the result from the computation. The pseudocode of *MGTaskGenerator* is as in Figure 11.

```
1. Start MATLAB
2. Starts the required number (N) of Tasks
3. Issue "PPSTART(N)" to MATLAB
4. Issue command to MATLAB to run user program
5. Issue "PPSTOP" to MATLAB
6. Stop MATLAB
7. Return result
```

Figure 11. *MGTaskGenerator* Pseudocode

2) MGTask

MGTask is created by adding the server side code into ALiCE Task template. Each *MGTask* instantiates an *MGProducer* and runs it. The pseudocode for *MATLAB*G Task* is as in Figure 12.

```
1. Instantiates MGProducer
2. Execute the run method in MGProducer
```

Figure 12. *MATLAB*G Task* Pseudocode

3) MGResultCollector

MGResultCollector extends the ALiCE Result Collector template. Running on the ALiCE Resource Broker, it simply waits for the result from JavaSpaces. An ALiCE program is created when we compile these elements together. The structure of such a program is described in Figure 13.

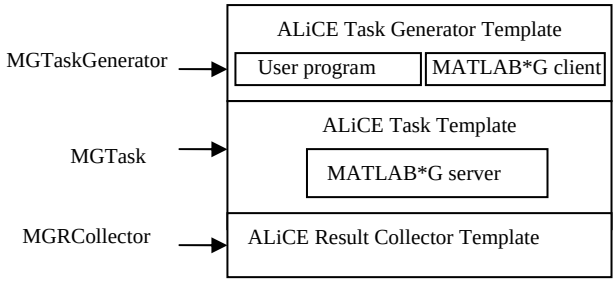

Figure 13. Structure for a ALiCE Program generated by MATLAB*G

After an ALiCE program is submitted to ALICE, the system dynamically finds available resources to join in the parallel computation, and each component is dynamically loaded by various machines as shown in Figure 14. Shaded boxes are ALiCE program elements for MATLAB*G.

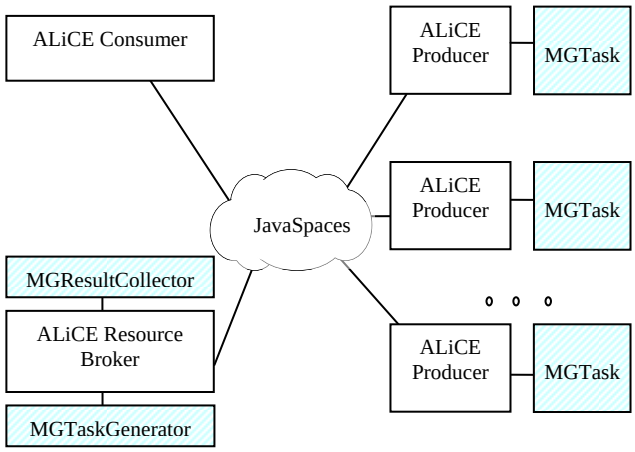

Figure 14. MATLAB*G running on ALiCE

4.4.3 Batch Mode

ALiCE supports two types of applications. Batch applications are non-interactive applications and involve minimum user intervention. This mode is for executing large jobs. After submitting the application, the Consumer can disconnect itself and later reconnect for collecting the results. The result collection mechanism is implemented at the Resource Broker. Interactive applications require User/Consumer intervention during execution process. In this mode, Users/Consumers can program a graphical user interface (GUI) to visualize the progress of the execution. Results of executing individual tasks generated by the Task Manager are returned to the Consumer.

The current MATLAB*G implementation supports batch applications: the user submits a complete MATLAB program instead of entering commands interactively at a MATLAB environment. This is accomplished by embedding the MATLAB program into *MGTaskGenerater* so that it can be submitted to the ALiCE Resource Broker.

5 Experimental Results

We compare the performance of MATLAB*G with sequential MATLAB on the ALiCE Grid. The experiments are conducted on ALiCE Grid Cluster with twenty-four nodes connected by 100 Mbps Ethernet. Four nodes are used, each of which is a PIII 866MHz, 256 MB RAM machine running Linux 2.4.

The current implementation of MATLAB*G can exploit two forms of parallelism. The first is task parallelism. When a user wants to perform computation involving matrices, the computation can be divided into a number of tasks. Each task has the computation name and parameter sub-matrices. Tasks are sent to space and each producer gets a task from space and performs computation on its sub-matrices. The number of tasks a matrix computation should be split into is largely influenced by

the complexity of the computation. A simple matrix computation (e.g. matrix addition) should be split into a small number of tasks so that the communication overhead does not dominate the computation time. Conversely, a complex matrix computation (e.g. computation of eigenvalues) should be split into a relatively large number of tasks. In general the number of tasks should be larger the number of Producers for load balancing consideration, as each node in the grid may have different computation ability and different network latency.

The second is job parallelism. When there are a number of matrix computations (jobs) to be executed one after the other, the user can specify for them to be executed in parallel. This will result in each matrix computation being executed on a single producer.

We perform experiments to discover the performance of the MATLAB*G implementation on each type of parallelism. Specifically, we measure performance in terms of the time elapsed on the client side from submission of application to receipt of results.

5.1 Task Parallelism

The designer of MATLAB has previously stated that one reason for not developing a parallel MATLAB is that it takes much more time to distribute the data than perform the computation because a matrix that fits into the host's memory would not be large enough to make efficient use of the parallel computer [25]. However this is true only for functions provided by MATLAB itself, which can be performed quite fast by MATLAB. For some MATLAB scripts written by a user, it is possible that the computation time for a normal size matrix is long enough that we can benefit from doing it in parallel.

For example, Figure 15 is a simple but compute intensive function that computes for 1000 times the exponential for each element of a matrix A.

```
Function result=Exp_1(A)
For (i=1:1000)
exp(A);
End;
Result=A;
```

Figure 15. A User-defined Matlab Function

We time this program in MATLAB and in MATLAB*G on various input matrix size and reproduce the results in Figure 16.

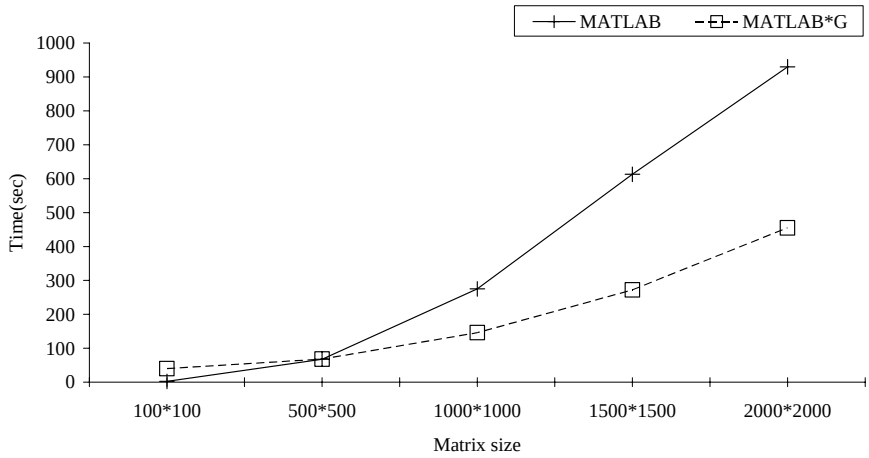

Figure 16. Task Parallelism - Varying Matrix Size

It can be seen that for small matrix size (e.g. 100x100), the elapsed time for sequential MATLAB is still less than that of MATLAB*G. This phenomenon is attributed to the communication and partitioning overhead which is much larger than the computation time. However, as matrix size

increases, the performance of MATLAB*G improves relative to sequential MATLAB, eventually overtaking it at the cross-point of approximately 500x500.

5.2 Job Parallelism

E = pinv(*X*) is the pseudo-inverse function provided by MATLAB. If a user has to perform *pinv*() on a few matrices, he can perform *pinv*() on each matrix one by one; alternatively he can parallelize these jobs as shown in Figure 17.

```
A1=randn(1000);
A2=randn(1000);
A3=randn(1000);
A4=randn(1000);
E1=pinv(A1);
E2=pinv(A2);
E3=pinv(A3);
E4=pinv(A4);
```

Figure 17.a. Compute *pinv*()s Sequentially

```
A1=randn(1000);
A2=randn(1000);
A3=randn(1000);
A4=randn(1000);
X(1:1000, :)=A1;
X(1001:2000,:)=A2;
X(2001:3000, :)=A3;
X(3001:4000,:)=A4;
Y=mm('pinv', 4, X);
E1=Y(1:1000,:);
E2=Y(1001:2000,:);
E3=Y(2001:3000,:);
E4=Y(3001:4000,:);
```

Figure 17.b. Compute *pinv*()s in Parallel

We time for the sequential program as in Figure 17.a and the parallel program as in Figure 17.b on various matrix sizes and reproduce the results in Figure 18.

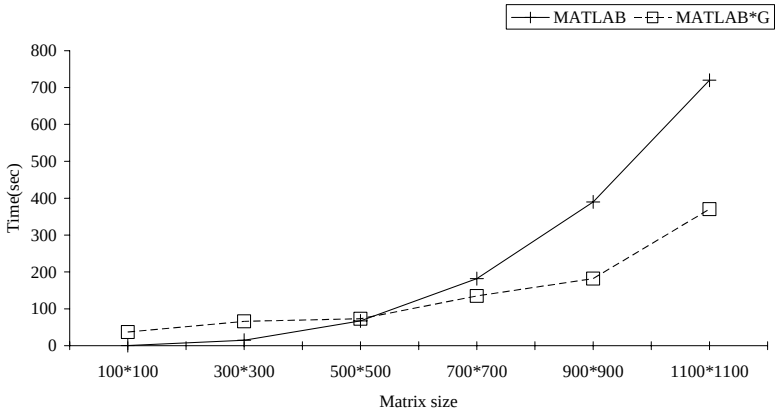

Figure 18. Job Parallelism - Varying Matrix Size

Once again we see that for small matrix size the elapsed time for sequential MATLAB is still less than that of MATLAB*G. But as matrix size exceeds 500x500, MATLAB*G outperforms sequential MATLAB.

6 Conclusion and Further Works

We discussed the design and implementation of the ALiCE object-oriented grid programming template that supports the distributed-shared memory programming model. We use the grid programming template as a system programming tool to develop a grid parallel MATLAB called MATLAB*G. Currently two types of parallelism for matrix computation are implemented: *task parallelism* and *job parallelism*. Performance results show that for large matrix sizes MATLAB*G can be a faster alternative to sequential MATLAB. Future work includes exploiting MATLAB *for*-loop parallelism, one of the most time-consuming computations in many MATLAB programs.

Optimizations are also required to reduce overheads such as communication latency and matrix partitioning.

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 ALiCE [27]. We have used AOPT as the underlying programming abstraction to develop domain-specific grid programming environment such as GLAD (Grid Life sciences Applications Developer) [28]. As part of GLAD, we are developing tools and techniques for debugging grid applications.

References

1. M. Baker, R. Buyya, and D. Laforenza, 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. A. Baratloo, M. Karaul, Z. Kedem and P. Wyckoff, Charlotte: Metacomputing on the Web, *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.
3. N. Carriero, D. Gelernter, "Linda in Context," CACM 32/4, pp. 444-458, 1984
4. Cornell Multitasking Toolbox for MATLAB. [Online]. Available: http://www.tc.cornell.edu/Services/Software/CMTM/
5. R. Choy, "MATLAB*P 2.0: Interactive Supercomputing Made Practical." Master of Science Thesis, EECS, MIT, Sep 2002.
6. R. Choy, (2003, Oct. 12). Parallel MATLAB Survey [Online]. Available: http://theory.lcs.mit.edu/~cly/survey.html.
7. D. De Roure, M. A. Baker, N. R. Jennings, and N. R. Shadbolt, The Evolution of the Grid, Research Agenda, UK National eScience Center, 2002.
8. Distributed and Parallel Application Toolbox (DP-Toolbox). [Online]. Available: http://www-at.e-technik.uni-rostock.de/dp/
9. DOSMOS project. [Online]. Available: http://www.ens-lyon.fr/~llefevre/DOSMOS/dosmos.html
10. I. Foster, Computational Grids, Morgan Kaufmann Publishers, 1998.
11. I. Foster, and C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, *International Journal of Supercomputing Applications*, 11(2), pp 115-128, 1997.
12. I. Foster, C. Kesselman, and S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International Journal of Supercomputer Applications*, 15(3), 2001.
13. Vincent W. Freeh. (1994, Aug). A Comparison of Implicit and Explicit Parallel Programming. Journal of Parallel and Distributed Computing. [Online].
14. GigaSpaces Platform White Paper, GigaSpaces Technologies, Ltd., February 2002.
15. S. Hupfer, The Nuts and Bolts of Compiling and Running JavaSpaces Programs, Java Developer Connection, Sun Microsystems, Inc., 2000.
16. A. Itzkovitz, and A. Schuster, Distributed Shared Memory: Bridging the Granularity Gap, *Proceedings of the First ACM Workshop on Software Distributed Shared Memory (WSDSM)*, Greece, June 1999.
17. JavaSpace™ Specification, June 27, 1997. [Online]. Available: http://java.sun.com
18. JMatLink. [Online]. Available: http://www.held-mueller.de/JMatLink/index.html
19. Lee, Matsuoka, Talia, Sossman, Karonis, Allen and Thomas, A Grid Programming Primer Programming Models Working Group, Grid Forum 1, Amsterdam, 2001.
20. C. Lee, and D. Talia, "Grid Programming Models: Current Tools, Issues and Directions", in Grid Computing: Making the Global Infrastructure a Reality, F. Berman, G. Fox and T. Hey (eds.), Wiley, chap. 21, pp. 555-578, 2003.
21. M. Lewis, and A. Grimshaw, The Core Legion Object Model, *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.
22. MATCH. [Online]. Available: http://www.accelchip.com
23. MATmarks [Online]. Available: http://polaris.cs.uiuc.edu/matmarks/
24. Message Passing Interface. [Online]. Available: http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
25. C. Moler, Why There Isn't a Parallel Matlab. [Online]. Available: http://www.mathworks.com/company/newsletter/pdf/spr95cleve.pdf
26. Parallel MATLAB® Development for High Performance Computing, [Online] Available: http://www.rtexpress.com/isi/rtexpress/

Y.M. Teo, Y. Chen and X.B. Wang, *On Grid Programming and MATLAB*G*, Proceedings of 3rd International Conference on Grid and Cooperative Computing, pp. xx, Springer Verlag Lecture Notes in Computer Science, Wuhan, China, October 2004 (submitted).

27. Y.M. Teo, X. Wang, and 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.
28. Y.M. Teo, X. Wang and Y.K. Ng, *GLAD: A System for Developing and Deploying Large-scale Bioinformatics Grid*, Technical Report, Department of Computer Science, National University of Singapore, 2004.
29. Y.M. Teo and X. Wang, ALiCE: A Scalable Runtime Infrastructure for High Performance Grid Computing, Proceedings of IFIP International Conference on Network and Parallel Computing, Springer-Verlag Lecture Notes in Computer Science Series, ??, China, October 18-20, 2004 (submitted).
30. Y.M. Teo,Y.K. Ng and X. Wang, *Progressive Multiple Biosequence Alignments on the ALiCE Grid*, Proceeding of the 6th International Conference on High Performance Computing for Computational Science, Springer-Verlag Lecture Notes in Computer Science Series, ??, Spain, June 28-30, 2004.
31. Y.M. Teo, 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.
32. D.P. Ho, Y.M. Teo, J.P. Gozali, Solving the N-body Problem on the ALiCE Grid System, 7th Asian Computing Science Conference, Lecture Notes in Computer Science 2250, pp. 87-97, Springer-Verlag, Hanoi, Vietnam, December 2002.