MATLAB*G: A Grid-Based Parallel MATLAB

Ying Chen [1] and Suan Fong Tan [2]

[1] Singapore-MIT Alliance, E4-04-10, 4 Engineering Drive 3, Singapore-117576

[2] Department of Computer Science, National University of Singapore, Singapore-117534

***Abstract*—This paper describes the design and implementation of MATLAB*G, a parallel MATLAB on the ALiCE Grid. ALiCE (Adaptive and scaLable internet-based Computing Engine), developed at NUS, is a lightweight grid-computing middleware. Grid applications in ALiCE are written in Java and use the distributed shared memory programming model. Utilizing existing MATLAB functions, MATLAB*G provides distributed matrix computation to the user through a set of simple commands. Currently two forms of parallelism for distributed matrix computation are implemented: task parallelism and job parallelism. Experiments are carried out to investigate the performance of MATLAB*G on each type of parallelism. Results indicate that for large matrix sizes MATLAB*G can be a faster alternative to sequential MATLAB**.

***Index Terms*—MATLAB, DSM, Grid**

I. Introduction

MATLAB is a popular mathematical software that provides an easy-to-use interface for scientists and students to compute and visualize various computations. Computation intensive MATLAB applications can benefit from faster execution if parallelism is provided by MATLAB. With the increasing popularity of distributed computing, researchers have been building support for parallel computing into MATLAB. Up to now there are at least twenty-seven parallel MATLABs available [1].

With commodity computers getting more powerful and more affordable, and with more people connecting to the Internet, distributed computing is becoming more popular. A distributed system can be classified as (i) a cluster system, which is characterized by homogeneous compute nodes, fast networks, and central management; or (ii) a computational grid, which consists of a heterogeneous set of computing machines without central management.

Grid computing is defined as coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations [2], where a virtual organization is a collection of compute nodes who share their resources. There is no central control, the compute nodes are usually heterogeneous, and the communication cost between any two nodes varies depending on the nodes and time of communication. The system is dynamic because existing nodes may become unavailable without warning, and new nodes may join the grid.

One example of grid computing is Seti@Home, where the idle CPU times of desktop machines on the Internet are shared to analyze radio signals in the search for extra-terrestrial life; and file sharing systems like Napster and Kazaa, where disk storage is the resource that is being shared [3].

A grid middleware is a set of tools that can be used to build a grid system. For example, the Condor System [4] and the Globus System [5] are both middlewares. ALiCE (Adaptive and scaLable Internet-based Compute engine) is a grid computing middleware developed at NUS [6].

In this paper we present the design, implementation and experimental results of MATLAB*G, a parallel MATLAB on the ALiCE Grid, which can perform distributed matrix computation using task parallelism and job parallelism.

The remainder of the paper is organized as follows: classification and comparison among parallel MATLABs is given in section II; the design of MATLAB*G is illustrated in section III; in section IV the implementation of MATLAB*G on ALiCE is shown; experimental results are related in section V and some recommendations for future work and conclusions are presented in Section VI.

II. Related Works

In this section, we classify existing parallel MATLABs into different categories according to two criteria: First, whether they provide implicit or explicit parallelism. Second, the method used for inter-processor communication.

A. 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 [7]:

Manuscript received November 4, 2003.

Ying Chen is with Singapore-MIT Alliance, E4-04-10, 4 Engineering Drive 3, National University of Singapore, Singapore 117576 (phone: 65-68744366; e-mail: smacy@nus.edu.sg).

Suan Fong Tan, is with the Department of Computer Science, National University of Singapore, Singapore 117534. (e-mail: tansuanf@comp.nus.edu.sg).

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 [8], CONLAB Compiler[9], and MATCH[10]. 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 [11], MATmarks [12], and DP-Toolbox [13].

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.

For example, MAT*marks* extends the MATLAB language with commands to enable shared variables and process synchronization. In order to perform a parallel computation in MAT*marks*, the user must explicitly write the required communication and synchronization code. This is in contrast with an implicitly parallel MATLAB, where the system would handle the communication and synchronization "behind the scenes".

MATLAB*P is an explicitly parallel MATLAB designed at MIT. Unlike MAT*marks*, MATLAB*P handles communication and synchronization for the user. Where it differs from implicitly parallel MATLABs however, is that MATLAB*P requires the user to explicitly indicate the matrices which are to be distributed.

MATLAB*G is an explicitly parallel MATLAB. It 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.

B. 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:

Distributed Memory Architectures employ 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 [14] 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.

Distributed Shared Memory systems have two main architectures [15]:

- *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. MATLAB*G is designed for ALiCE Grid which uses Sun's Jini and JavaSpaces technologies.

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.

III. SYSTEM DESIGN

In MATLAB, a normal matrix addition can be performed by executing the program in Figure 1.

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

Fig. 1. A MATLAB program for matrix addition.

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 2.

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

Fig. 2. A MATLAB*G program for the parallel matrix addition.

The third line creates a 10-by-10 matrix C, performs parallel matrix addition between A and B, ad 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.

To achieve parallelism, MATLAB*G exploits a client-server model using Object-based Distributed Shared Memory (DSM). User submits his job interactively from MATLAB environment. The client gets the job, divides it into a number of tasks, sends tasks into the DSM, and then keeps polling the DSM for the result. A server always tries to get a task from DSM. After receiving a task, the server processes it, and then sends the result back 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 3.

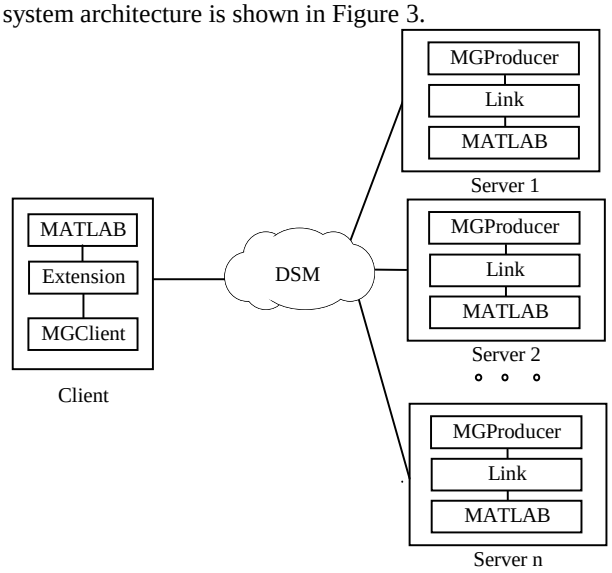

Fig. 3. MATLAB*G Client-Server Architecture

A. 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 4.

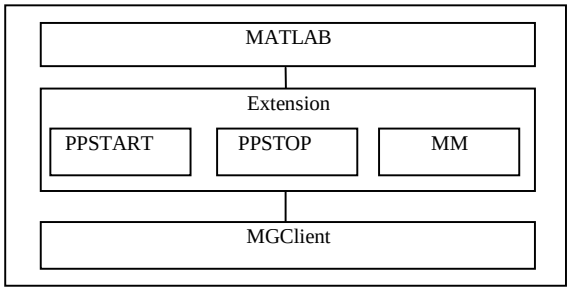

Fig. 4. MATLAB*G Client

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 5 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;
```

Fig. 5. *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*.

B. 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 6.

```
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
```

Fig. 6. *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 [16].

C. DSM

A tuplespace is a shared datastore for simple list data structures (tuples) [17]. 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.

IV. System Implementation

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

*A. Mapping MATLAB*G onto ALiCE*

There are two main components in ALiCE: a runtime system that handles management of grid resources, and a set of programming templates for users to develop applications on ALiCE. Figure 7 illustrates the ALiCE architecture.

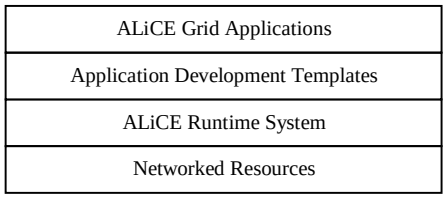

Fig. 7. Layers in ALiCE Architecture

The ALiCE Runtime System consists of three components: Consumer, Producers, and a Resource Broker. The Consumer provides a user interface for job submission, monitoring and collections of results. The Producers interface with local schedulers of shared grid resources. The Resource Broker manages the applications, scheduling of resources, communications and security of the system. The bottom two layers can be visualized in detail in Figure 8.

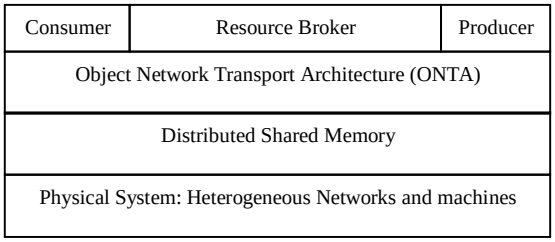

Fig. 8. ALiCE Runtime and Network Resources

The Object Network Transport Architecture (ONTA) provides a layer for communication between JavaSpaces [18] and the Consumer/Resource-Broker/Producer layer.

The JavaSpaces provides a distributed persistence and object exchange mechanism for code written in Java. Processes are loosely coupled and communicate and synchronize their activities using a persistent object store called a *space*, rather than through direct communication.

The mapping from MATLAB*G onto ALiCE is

illustrated in Figure 9. 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.

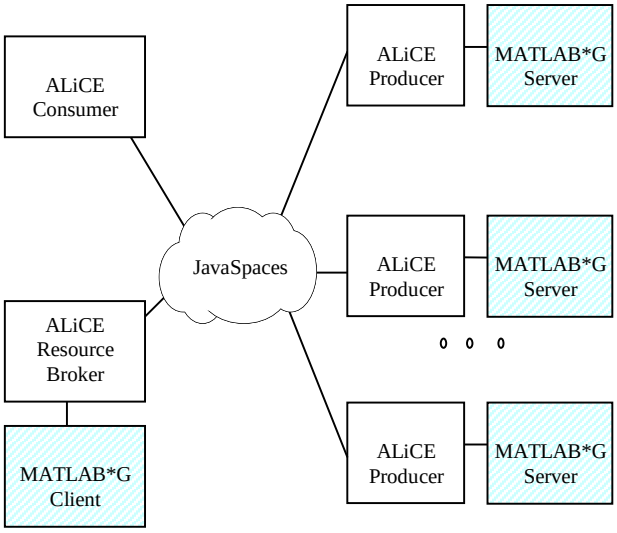

Fig. 9. Mapping between MATLAB*G and ALiCE components

B. *ALiCE Program Generated by MATLAB*G*

ALiCE does not provide any interface to allow a user to directly run a MATLAB*G client 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.

Templates for these three elements are provided by ALiCE. Thus our job is just to add MATLAB*G code into proper templates to generate customized ALiCE program elements.

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 10.

```
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
```

Fig. 10. *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 11.

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

Fig. 11. *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 12.

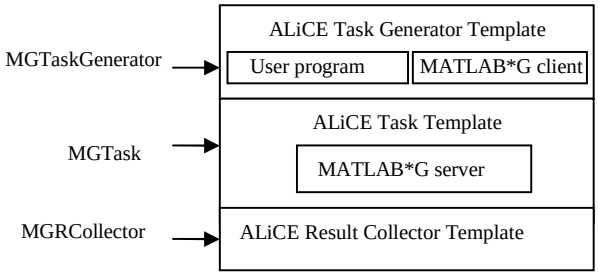

Fig. 12. Structure for a ALiCE Program generated by MATLAB*G

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 13. Shaded boxes are ALiCE program elements for MATLAB*G.

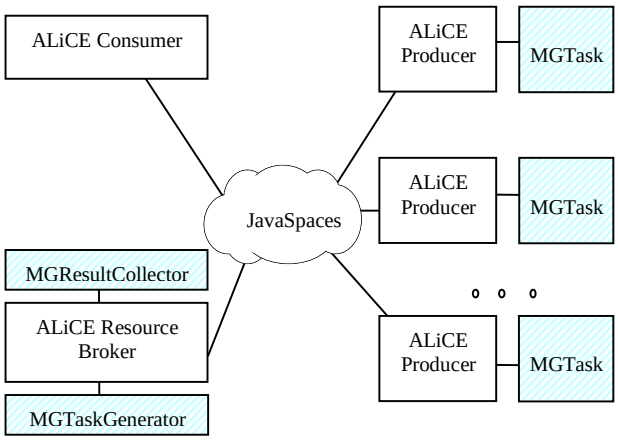

Fig. 13. MATLAB*G running on ALiCE

C. 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.

V. 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 0.866GHz, 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 submatrices. Tasks are sent to space and each producer gets a task from space and performs computation on its submatrices. 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.

A. 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 [19]. 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, we have a user program as in Figure 14.

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

Fig. 14. An example computation intensive program
exp(X) is the exponential of the elements of X, *e* to the X

We time this program in MATLAB and in MATLAB*G on various input matrix size and reproduce the results in Figure 15.

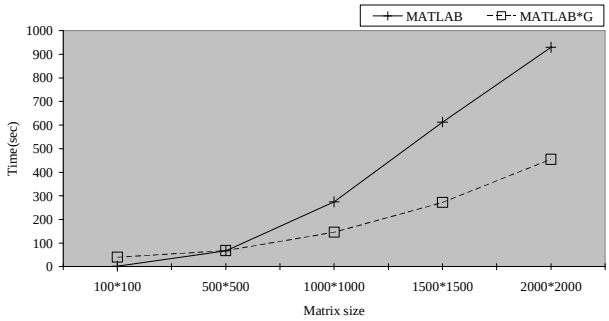

Fig. 15. Timing for Task Parallelism on varying matrix size

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 crosspoint of approximately 500x500.

B. Job Parallelism

E = pinv(X) is the pseudoinverse 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 16.

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

Fig. 16.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,:);
```

Fig. 16.b. Compute *pinv()*s in Parallel

We time for the sequential program as in Figure 16.a and the parallel program as in Figure 16.b on various matrix size and reproducer the results in Figure 17.

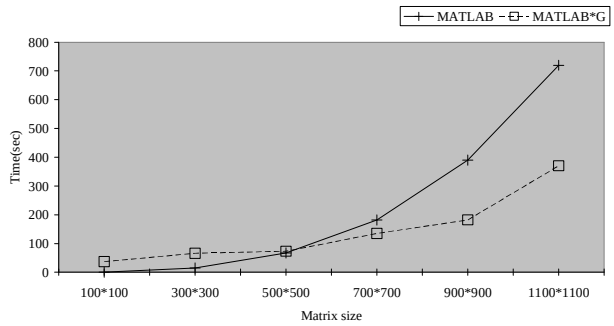

Fig. 17. Timing for Job Parallelism on varying matrix size

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.

VI. Conclusion and Future Work

MATLAB*G provides parallel MATLAB for the ALiCE Grid users. Currently two types of parallelism for matrix computation are implemented: task parallelism and job parallelism. Results of its performance indicate that for large matrix sizes MATLAB*G can be a faster alternative to sequential MATLAB. One direction for future work is to implement *for*-loop parallelism as they are one of the most time-consuming parts in many MATLAB programs. Optimizations are also required to reduce the cost of overheads such as communication time and matrix partitioning.

Acknowledgment

We would like to thank Prof. Yong Meng Teo for his invaluable support, guidance and patience; Yih Lee for his original conception, design and implementation of MATLAB*G; Ron Choy and Prof. Alan Edelman for their creation of MATLAB*P, from which MATLAB*G was inspired.

References

[1] R. Choy, (2003, Oct. 12). *Parallel MATLAB Survey* [Online]. Available: http://theory.lcs.mit.edu/~cly/survey.html.

[2] I Foster, C Kesselman, S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*. Intl. J. of Supercomputer Applications, 2001. Available: http://www.globus.org/research/papers/anatomy.pdf

[3] SETI@Home Project. [Online]. Available: http://setiathome.ssl.berkeley.edu/

[4] Condor Project. [Online]. Available: http://www.cs.wisc.edu/condor/

[5] Globus Project. [Online]. Available: http://www.globus.org/

[6] ALiCE Grid Computing Project. [Online]. Available: http://www.comp.nus.edu.sg/~teoym/alice.htm

[7] Vincent W. Freeh. (1994, Aug). A Comparison of Implicit and Explicit Parallel Programming. *Journal of Parallel and Distributed Computing*. [Online].

[8] Parallel MATLAB® Development for High Performance Computing, [Online] Available: http://www.rtexpress.com/isi/rtexpress/

[9] Cornell Multitasking Toolbox for MATLAB. [Online]. Available: http://www.tc.cornell.edu/Services/Software/CMTM/

[10] MATCH. [Online]. Available: http://www.accelchip.com

[11] R. Choy, "MATLAB*P 2.0: Interactive Supercomputing Made Practical." Master of Science Thesis, EECS, MIT, Sep 2002.

[12] MATmarks [Online]. Available: http://polaris.cs.uiuc.edu/matmarks/

[13] Distributed and Parallel Application Toolbox (DP-Toolbox). [Online]. Available: http://www-at.e-technik.uni-rostock.de/dp/

[14] Message Passing Interface. [Online]. Available: http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[15] DOSMOS project. [Online]. Available: http://www.ens-lyon.fr/~llefevre/DOSMOS/dosmos.html

[16] JMatLink. [Online]. Available: http://www.held-mueller.de/JMatLink/index.html

[17] N. Carriero, D. Gelernter, "Linda in Context," CACM 32/4, pp. 444-458, 1984

[18] JavaSpace™ Specification, June 27, 1997. [Online]. Available: http://java.sun.com

[19] C. Moler. *Why There Isn't a Parallel Matlab*. [Online]. Available: http://www.mathworks.com/company/newsletter/pdf/spr95cleve.pdf

Ying Chen obtained her B.E. and M.E. in computer science from Xian Jiaotong University in China in 1999 and 2002 respectively, and MSc in computer science from Singapore-MIT Alliance in 2003. She is currently a research assistant in Singapore-MIT Alliance. Her current research interests include distributed systems, parallelism and grid computing. Author's Present Address: Singapore-MIT Alliance, E4-04-10, 4 Engineering Drive 3, National University of Singapore, Singapore 117576, smacy@nus.edu.sg.

Suan Fong Tan is an undergraduate pursuing a B.Sc. in computer science from the National University of Singapore. His current research interests include distributed systems, parallelism and grid computing. Author's Present Address: Computer Systems Laboratory, S14-06-17, 3 Science Drive 2, National University of Singapore, Singapore 117543, tansuanf@comp.nus.edu.sg.