MATLAB*P 2.0: A unified parallel MATLAB

Ron Choy, Alan Edelman
Laboratory for Computer Science,
Massachusetts Institute of Technology,
Cambridge, MA 02139
Email: cly@mit.edu, edelman@math.mit.edu

***Abstract*— MATLAB [?] is one of the most widely used mathematical computing environments in technical computing. It is an interactive environment that provides high performance computational routines and an easy-to-use, C-like scripting language. Mathworks, the company that develops MATLAB, currently does not provide a version of MATLAB that can utilize parallel computing [?]. This has led to academic and commercial efforts outside Mathworks to build a parallel MATLAB, using a variety of approaches. In a survey [?], 26 parallel MATLAB projects utilizing four different approaches have been identified.**

MATLAB*P [?] is one of the 26 projects. It makes use of the backend support approach. This approach provides parallelism to MATLAB programs by relaying MATLAB commands to a parallel backend. The main difference between MATLAB*P and other projects that make use of the same approach is in its focus. MATLAB*P aims to provide a user-friendly supercomputing environment in which parallelism is achieved transparently through the use of objected oriented programming features in MATLAB. One advantage of this approach is that existing scripts can be run in parallel with no or minimal modifications.

This paper describes MATLAB*P 2.0, which is a complete rewrite of MATLAB*P. This new version brings together the backend support approach with embarrassingly parallel and MPI approaches to provide the first complete parallel MATLAB framework.

I. Background

MATLAB [**?**] is one of the most widely used tools in scientific and technical computing. It started in 1970s as an interactive interface to EISPACK [**?**] and LINPACK [**?**], a set of eigenvalue and linear system solution routines. It has since grown to a feature rich product utilizing modern numerical libraries like ATLAS [**?**] and FFTW [**?**], and with toolboxes in a number of application areas, for example, financial mathematics, neural networks, and control theory. It has a built-in interpreted language that is similar to C and HPF, and the flexible matrix indexing makes it very suitable for programming matrix problems. It is a very useful tool to a wide audience. For example, it comes in very handy as a matrix calculator for simple problems, because of its easy-to-use command syntax. Its strong graphical capabilities makes it a good data analysis tool. Also researchers have been known to build very complex systems using MATLAB scripts.

Because of its roots in serial numerical libraries, MATLAB has always been a serial program. However, as modern engineering and simulation problems become more and more complex, space and computing time requirements for solutions skyrocketed, and a serial MATLAB is no longer able to handle them. Compared to similar software like Maple and Mathematica, MATLAB is known to have a larger number of users. For example, at MIT, MATLAB is the dominant computing software used by the student body.

The above factors sparked a lot of interest in creating a parallel MATLAB. In a recent survey [**?**], 26 different parallel MATLAB projects have been identified. These projects vary in their scope: some are one-man projects that provide basic embarrassingly parallel capabilities to MATLAB; some are university or government lab research projects; while some are commercial products that enables the use of MATLAB in product development. Also their approaches to making MATLAB parallel are different: some compile MATLAB script into parallel native code; some provide a parallel backend to MATLAB, using MATLAB as a graphical frontend; and some others coordinate multiple MATLAB processes to work in parallel. These projects also vary widely in their status: some are now defunct and exist only in Google web cache, while some are entering their second or third revision.

II. MATLAB*P 2.0 Overview

MATLAB*P 2.0 is a parallel MATLAB using the backend support approach, aimed at widespread circulation among a general audience. In order to achieve this, we took ideas from the approaches used by other software found in the survey. For example, the *embarrassingly parallel* approach allow simplistic, yet useful, division of work into multiple MATLAB sessions. The *message passing* approach, which is a superset of the embarrassingly parallel approach, allows finer control between the MATLAB sessions.

In order to make MATLAB*P useful for a wider range of audience, these other approaches are incorporated into the system as well.

A. Structure of MATLAB*P 2.0 system

MATLAB*P 2.0 is a complete rewrite of MATLAB*P in C++. The code is organized in a way to ensure easy extension of the software.

The server itself is divided in four self-contained parts:

1) Client Connection Manager
   Client Connection Manager is responsible for communications with the client. It provides functions for reading commands and arguments from the client and sending the results back to the client. It is only used in the head server process.
2) Server Connection Manager

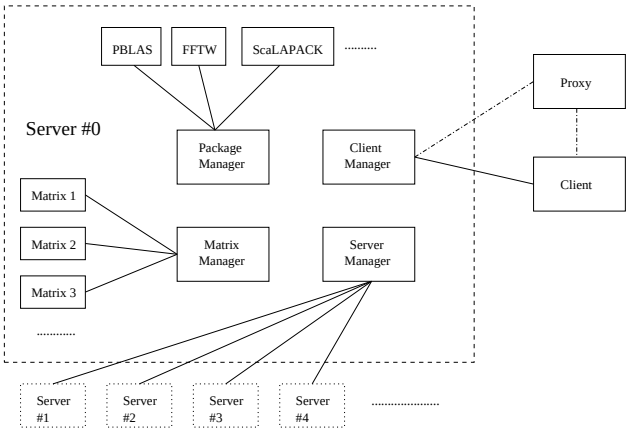

Fig. 1. Structure of MATLAB*P 2.0

Server Connection Manager takes care of commucation between server processes. It mainly controls broadcasting of commands and arguments from head process to the slave processes, and collection of results and error codes. Also it provides rank and size information to the processes.

3) Package Manager
   Package Manager is responsible for maintaining a list of available packages and functions provided by them. When initiated by the server process, Package Manager will also perform the actual call to the functions.
4) Matrix Manager
   Matrix Manager contains all the functions needed to create, delete and change the matrices on the server processes. It maintains a mapping from client-side matrix identifiers to actual matrices on the server. It is also responsible for performing garbage collection.

This organization offers great advantages. First of all, debugging is made easier because bugs are localized and thus are much easier to track down. Also, this compartmentized approach allows easier extension of the server. For example, the basic Server Connection Manager makes use of MPI (Message Passing Interface) as the means of communication between server processes. However, one could write a Server Connection Manager that uses PVM (Parallel Virtual Machine) instead. As long as the new version implements all the public functions in the class, no change is needed in any other part of the code.

Similar extensions can be made to Client Connection Manager as well. The basic Client Connection Manager uses TCP socket. An interesting replacement would be to make a Client Connection Manager that act as an interface to a language like C++ or Java.

III. Features of MATLAB*P 2.0

A. Parallelism through Polymophism - *p

The key to the system lies in the *p variable. It is an object of *layout* class in MATLAB. Through the use of the *p variable, matrices that are distributed on the server could be created. For example,

```
X = randn(8192*p,8192);
```

The above creates a row distributed, 8192 x 8192 normally distributed random matrix on the server. X is a handle to the distributed matrix, identified by MATLAB as a *ddense* class object. By overloading randn and many other built-in functions in MATLAB, we are able to tie in the parallel support transparent to the user. This is called *parallelism through polymorphism*

```
e = eig(X);
```

The command computes the eigenvalues of X by calling the appropriate ScaLAPACK routines, and store the result in a matrix e, which resides on the server. The result is not returned to the client unless explicitly requested, to reduce data traffic.

```
E = pp2matlab(e);
```

This command returns the result to MATLAB.

The use of the *p variable along with overloaded MATLAB routines enable existing MATLAB scripts to be reused. For example,

```
function H = hilb(n)

J = 1:n;
J = J(ones(n,1),:);
I = J';
E = ones(n,n);
H = E./(I+J-1);
```

The above is the built-in MATLAB routine to construct a Hilbert matrix. Because the operators in the routine (colon, ones, subsasgn, transpose, rdivide, +, -) are overloaded to work with *p, typing

```
H = hilb(16384*p)
```

would create a 16384 by 16384 Hilbert matrix on the server. By exploiting MATLAB's object-oriented features in this way, many existing scripts would run in parallel under MATLAB*P without any modification.

B. Data Parallel/Embarassingly Parallel Operations

One of the goals of the project is to make the software to be useful to as wide an audience as possible. In order to achieve this, we found that it would be fruitful to combine other parallel MATLAB approaches into MATLAB*P, to provide a unified parallel MATLAB framework.

In conjuction with Parry Husbands, we developed a prototype implementation of a MultiMATLAB-like, distributed MATLAB package in MATLAB*P, which we call the PPEngine. With this package and associated m-files, we can run multiple MATLAB processes on the backend and evaluate MATLAB functions in parallel on dense matrices.

The system works by starting up MATLAB engine instances on each node through calls to the MATLAB engine interface. From that point on, MATLAB commands can be relayed to the MATLAB engine.

Examples of the usage of the PPEngine system:

```
>> % Example 1
>> a = 1:100*p;
>> b = mm('chi2rnd',a);
```

The first example creates a distributed matrix of length 100, then fill it with random values from the chi-square distribution through calls to the function chi2rnd from MATLAB statistics toolbox.

```
>> % Example 2
>> a = rand(100,100*p);
>> b = rand(100,100*p);
>> c = mm('plus',a,b);
```

This example creates two column distributed matrices of size 100x100, adds them, and puts the result in another matrix. This is the slow way of doing the equivalent of:

```
>> a = rand(100,100*p);
>> b = rand(100,100*p);
>> c = a+b;
```

Besides providing embarrassingly parallel capabilities like in the above examples, we are also interested in integrating MPI-like functionalities into the system. We do so by making use of the MatlabMPI [**?**] package. This package has the advantage of being written in pure MATLAB code, so integrating simply means making calls to this package through PPEngine.

```
>> % Example 3
>> piapprox = mmmpi('cpi',100);
```

This example calls cpi.m with 100. This calls a MATLAB version of the popular pi-finding routine distributed with MPICH, cpi.c. All the MPI calls are simulated within MATLAB through MatlabMPI.

C. Complex Number Support

MATLAB*P 1.0 only supports real matrices. In the new version complex number support is added. This is particularly important for FFT routines and eigenvalue routines.

Adding this support is the main reason why it is necessary to rewrite the code from scratch. Interaction between real and complex matrices, and hooks for complex number routine calls to numerical libraries are extremely error prone if they were to be added on top of MATLAB*P 1.0.

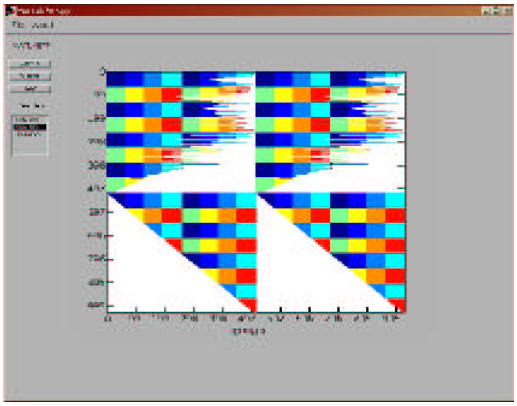

Fig. 2. ppspy on 1024x1024 matrix on eight nodes

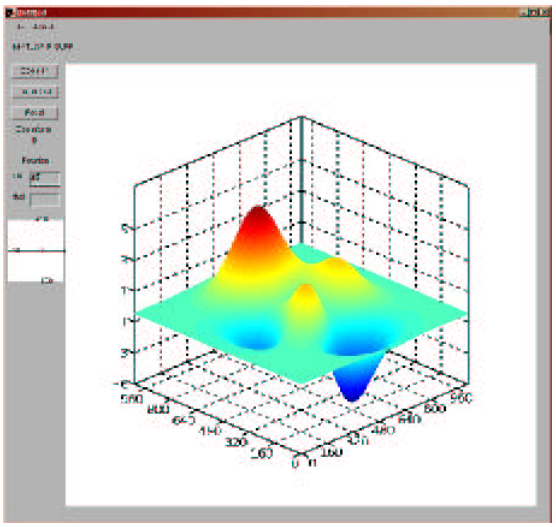

Fig. 3. ppsurf on the 1024x1024 'peaks' matrix

D. Visualization Package

This visualization package was written by Bruning, Holloway and Sulejmanpasic, under supervision of Ron Choy, as a term project for the class 6.338/18.337 - Applied Parallel Computing at MIT. It has since then been merged into the main MATLAB*P source.

This package adds *spy*, *surf*, and *mesh* routines to MATLAB*P. This enable visualization of very large matrices. The rendering is done in parallel, using the Mesa OpenGL library.

Figure **??**, **??**, **??** shows the routines in action.

All three routines allow zooming into a portion of the matrix. Also, ppsurf and ppmesh allow changing of the camera angle, just as supported in MATLAB.

E. Two-dimensional Block Cyclic Distribution

MATLAB*P 1.0 supports row and column distribution of dense matrices. These distributions assign a block on contiguous row/column of a matrix to successive processes.

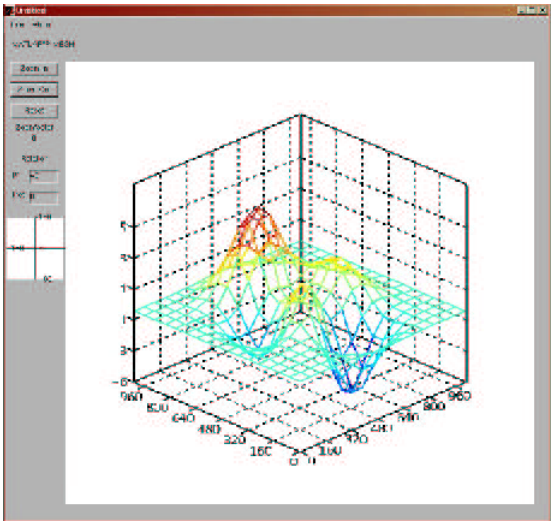

Fig. 4. ppmesh on the 1024x1024 'peaks' matrix

Each process receives only one block of the matrix. These distribution suffers from load balancing issues.

A good example that illustrate the load balancing problem is Gaussian elimiination. Recall the Gaussian elimiation algorithm solves a square linear system by 'zeroing out' the subdiagonal entries.

```
for i = 1:n
  for j = i+1:n
    A(j,i:n) = A(j,i:n) -
A(i,i:n)*A(j,i)/A(i,i);
  end
end
```

The algorithm proceeds column by column from left to right. For column distribution, as the algorithm go past the columns stored in process i, process 0, ..., i will be idle for the rest of the computation. Row distribution suffers from a similar deficiency.

Two-dimensional block cyclic distribution avoid this problem by 'dealing out' blocks of the matrix to the processes in a cyclic fashion. This is illustrated in figure **??**. The figure shows the two-dimensional block cyclic distribution for four processes in a 2x2 grid. This way all processes remains busy throughout the computation, thus achieving load balancing. Furthermore, this approach carries the additional advantages of allowing BLAS level-2 and BLAS level-3 operations on local blocks. A detailed discussion of the advantages of two dimensional block cyclic distribution can be found in Dongarra, Van De Geijn and Walker [**?**].

IV. Benchmarks

We compare the performance of MATLAB*P, MATLAB, and ScaLAPACK on a Beowulf cluster running Linux.

1	2	1	2	1	2	1	2
3	4	3	4	3	4	3	4
1	2	1	2	1	2	1	2
3	4	3	4	3	4	3	4
1	2	1	2	1	2	1	2
3	4	3	4	3	4	3	4
1	2	1	2	1	2	1	2
3	4	3	4	3	4	3	4

Fig. 5. Two-dimensional block cyclic distribution

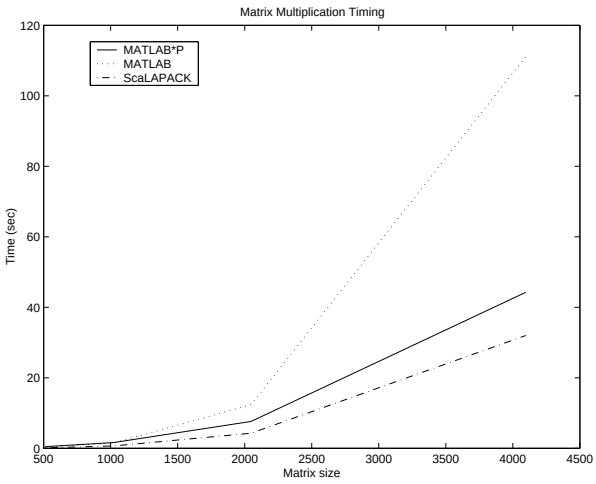

Fig. 6. Matrix multiplication timing results

A. Test Platform

- Beowulf cluster with 9 nodes (2 nodes are used in the tests).
- Dual-processor nodes, each with two 1.533GHz Athlon MP processors.
- 1GB DDR RAM on each node. No swapping occurred during benchmarks.
- Fast ethernet (100Mbps/sec) interconnect. Intel Etherfast 410T switch.
- Linux 2.4.18-4smp

B. Timing Results

See graphs of matrix multiplication timing results and linear system solve timing results.

C. Interpretation

*1) MATLAB*P and MATLAB:* From the results, MATLAB*P on 4 processors begin to outperform MATLAB on single processor when the problem size is 2048 and upward. This shows that for smaller problems, one should use plain MATLAB instead of MATLAB*P. When the problem size is large, MATLAB*P offers two advantages:

- Better performance
- Distributed memory, enabling larger problems to fit in memory.

And all these come at close to zero effort on the user's part, once the system is installed and configured.

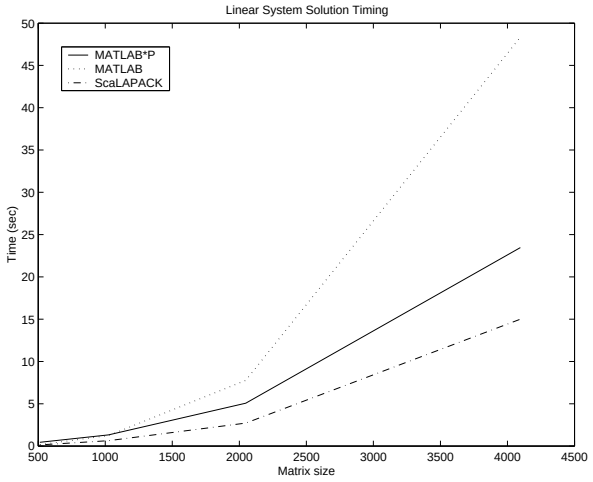

Fig. 7. Linear system solve timing results

*2) MATLAB*P and ScaLAPACK:* Comparing the timing results of MATLAB*P and ScaLAPACK, we see that ScaLAPACK is always faster than MATLAB*P, although the gap narrows at larger problem size. This should be obvious from the fact that MATLAB*P uses ScaLAPACK for matrix multiplication and linear system solution.

The difference in the timing results come from both overhead incurred by MATLAB*P and the design of the benchmark itself:

- Timing for MATLAB*P is done inside MATLAB, using tic/toc on the MATLAB call. The MATLAB call includes memory allocation and matrix copying on the server side. Timing for ScaLAPACK is done in C++ using clock(), and only the actual computation routine is timed.
- There is a messaging overhead from the MATLAB client to the server. As the MATLAB call yields multiple calls to the server, this messaging overhead is multiplied.
- In linear system solution, ScaLAPACK overwrites the input matrix. In MATLAB*P, in order to mimic standard MATLAB behaviour, the input matrix is copied into another matrix which is used in the ScaLAPACK call. This incurred additional overhead.

V. Future work

A. Further work on unified framework

Work could be done to improve on the portability and generality of the data parallel operation. Currently there are certain restrictions on the type and size of inputs to the data parallel mode that only exists because of technical difficulties.

Also, example applications that demonstrate the power of this unified framework should be found.

B. Sparse Matrix Routines

Because of time constraints, sparse matrix routines are not included in the release of MATLAB*P 2.0. If there is sufficient interest they will be added.

C. MATLAB*P on Grids of Beowulfs

Recently there has been a surge on interest in Beowulf-class supercomputers. Beowulfs are clusters of machines made from commodity-off-the-shelf (COTS) hardware. Example of COTS hardware includes 80x86 processors, DDR memory, and IDE hard drives.

The advantage of Beowulfs over traditional supercomputers like SGI Origin or IBM SP2 is that Beowulfs are much cheaper to build and maintain. For example, a Beowulf with 18 processors (1.533GHz) and 9GB RAM costs around $15000 at time of writing. On the other hand, a Sun Enterprise 250 Server, with 2 processors and 2GB RAM, costs around $16000. As Beowulfs generally run open source operating systems like Linux, the software cost is also cut substantially.

Along with the popularity of Beowulfs, there has also been rising interest in the concept of computational grids [**?**]. Its focus is on large-scale resource sharing, namely, resource sharing among geographically distributed individuals and institutions. The resources shared could be applications, databases, archival storage devices, visualization devices, scientific instruments, or CPU cycles. One of the most popular software that implements the Grid architecture is the Globus Toolkit [**?**].

We are interested in combining the above two very popular ideas, to see how MATLAB*P would run on a Grid of Beowulfs. Both of these projects failed. Both of them failed at the point of setting up a working MPICH-G2 on the clusters. In order to have MATLAB*P run on the grid, it is necessary to build it upon a special, Grid-aware version of MPI. All the Beowulf clusters we used have private networks, meaning only designated 'front end' nodes are exposed to the outside world, while the other nodes are all hidden and form its own network. This caused MPICH-G2 to fail, because the MPI processes started up by MPICH-G2 were located in different private networks and were unable to contact one another.

This is very unfortunate, because we believe that Grids of Beowulf clusters is a very powerful and cost-effective architecture. When we turned to the Globus discussion mailing list for help, we were told that this could be achieved by making MPICH-G2 aware of local MPI implementations in each Beowulf cluster and making use of them, but documentations on this setup is scarce and we were unable to proceed with this approach.

D. Parallel MATLAB Toolbox

One problem with the current state of MATLAB*P 2.0 is the difficulty in setting up the softare. MATLAB*P 2.0 depends on a number of established libraries for communication and computational routines: BLACS, MPI, PBLAS, ScaLAPACK, PARPACK, FFTW and SPRNG. Unless the system MATLAB*P 2.0 is running on already has these libraries up and running, installing MATLAB*P 2.0 would often be a painful process of tracking down bugs in the installation of libraries totally unfamiliar to the user.

A solution to this problem is to port the entire code into MATLAB scripts. This is best demonstrated by MatlabMPI, which implements basic MPI routines entirely in MATLAB scripts. There are several advantages to this approach.

1) Portability: First of all, the software would run where MATLAB runs. And because MATLAB is a commercial product by Mathworks, it is tested extensively to ensure its portability. By building the software upon MATLAB, we would be able to take advantage of this extensive testing.
2) Easy to install: Installing a system made up of MATLAB scripts would require no compilation. Just set the MATLAB path correctly and the system should be ready.

MatlabMPI is a good example that illustrates the strong points of this approach. Because MATLAB*P is MPI-based, there have been discussions of porting MATLAB*P to a purely MATLAB script code base, using MatlabMPI as the underlying communication layer.

If this were to be pursued, the resulting MATLAB*P would consist of a group of MATLAB processes working together. In addition to developing the equivalent of the current MATLAB*P code in MATLAB scripts, we have identified the following additional functionalities that have to be implemented:

1) Supporting libraries: One of the strong points of MATLAB*P is the leverage of established, high-performance parallel numerical libraries. In moving towards a purely MATLAB code base, since these libraries do not exist in MATLAB, those functions they provide will have to be implemented by ourselves. This means that all or part of ScaLAPACK, PARPACK, and FFTW have to be implemented in MATLAB, debugged and maintained. Also as MATLAB scripts are interpreted instead of compiled, they do not get compiler optimization which optimizes memory access patterns and cache usage. Thus the resulting code would be slower.
2) MPI routines: MatlabMPI supports only basic MPI routines like Send, Recv, Size and Rank. As MATLAB*P utilizes more advanced features like Reduce and Scatter/Gather, they have to be implemented.

These alone would increase the size of the code base dramatically and make code maintenance harder than before, despite the move to MATLAB scripts, which is supposed to be easier to write and understand. Yet still, it is interesting to investigate whether this approach would yield a better MATLAB*P.