MultiMATLAB:
MATLAB on Multiple Processors

Anne E. Trefethen
Cornell Theory Center
aet@tc.cornell.edu
http://www.tc.cornell.edu/~anne

Vijay S. Menon
Computer Science Department, Cornell University
vsm@cs.cornell.edu
http://www.cs.cornell.edu/Info/People/vsm

Chi-Chao Chang
Computer Science Department, Cornell University
chichao@cs.cornell.edu
http://www.cs.cornell.edu/Info/People/chichao/chichao.html

Grzegorz J. Czajkowski
Computer Science Department, Cornell University
grzes@cs.cornell.edu
http://www.cs.cornell.edu/Info/People/grzes/grzes.html

Chris Myers
Cornell Theory Center
myers@tc.cornell.edu
http://www.tc.cornell.edu/CSERG/myers

Lloyd N. Trefethen
Computer Science Department, Cornell University
lnt@cs.cornell.edu
http://www.cs.cornell.edu/home/lnt

Abstract:

MATLAB®, a commercial product of The MathWorks, Inc., has become one of the principal languages of desktop scientific computing. A system is described that enables one to run MATLAB conveniently on multiple processors. Using short, MATLAB-style commands like Eval, Send, Recv, Bcast, Min, and Sum, the user operating within one MATLAB session can start MATLAB processes on other machines and then pass commands and data between between these various processes in a fashion that maintains MATLAB's traditional user-friendliness. Multi-processor graphics is also supported. The system currently runs under MPICH on an IBM SP2 or a network of Unix workstations, and extensions are planned to networks of PCs. MultiMATLAB is potentially useful for education in parallel programming, for prototyping parallel algorithms, and for fast and convenient execution of easily parallelizable numerical computations on multiple processors.

Keywords:

MATLAB, MultiMATLAB, SP2, message passing, MPI, MPICH

1. Introduction

1.1. The Popularity of MATLAB

MATLAB® is a high-level language, and a problem-solving environment, for mathematical and scientific calculations. It originated in the late 1970s with an attempt by Cleve Moler to provide interactive access to the Fortran linear algebra software packages EISPACK and LINPACK. Soon a programming language emerged (programs conventionally have the extension `.m` and are called "m-files") containing dozens of high-level commands such as `svd` (singular value decomposition), `fft` (fast Fourier transform), and `roots` (polynomial zerofinding). Graphical commands were built into the language, and a company called The MathWorks, Inc. was formed in 1984 by Moler and John Little, now based in Natick, Massachusetts.

From the beginning, MATLAB proved greatly appealing to users. The numerical analysis and signal processing communities in the United States took to it quickly, followed by other groups of scientists and engineers in the U.S. and abroad. Roughly speaking, the number of MATLAB users has doubled each year since 1978. According to The MathWorks, there are currently about 300,000 users in fifty countries, and this figure continues to increase rapidly. In many scientific and engineering communities, MATLAB has become the dominant language for desktop numerical computing.

At least six reasons for MATLAB's success can be identified. The first is an exceptionally user-friendly, intuitive syntax, favoring brevity and simplicity at all turns without being so compressed as to interfere with intelligibility. The second is the very high quality of the underlying numerical programs, a result of MATLAB's intimate ties from the beginning with the numerical analysis research community. The third is powerful and user-friendly graphics. The fourth is the high level of the language, which often makes it possible to carry out computations in a line or two of MATLAB that would require dozens or hundreds of lines in Fortran or C. (The ability to link with Fortran or C programs is also provided.) The fifth is MATLAB's easy extensibility via packages of m-files known as Toolboxes. Many Toolboxes have been produced over the years, both by The MathWorks and by others, covering application areas such as optimization, signal processing, fuzzy logic, partial differential equations, and mathematical finance. Finally, perhaps the most interesting reason for MATLAB's success may be that from the beginning, the whole language has been built around real or complex vectors and matrices (including sparse matrices) as the fundamental data type. To computer scientists not involved with numerical computation, such a limitation may seem narrow and capricious, but it has proved extraordinarily fruitful.

It is probably fair to say that one of the three or four most important developments in numerical computation in the past decade has been the emergence of MATLAB as the preferred language of tens of thousands of leading scientists and engineers.

1.2. Single vs Multiple Processors

Curiously, one of the other principal developments of the past decade has been orthogonal to that one. This is the move from single to multiple processors. A new generation of researchers and practitioners have grown up who are accustomed to the principle that high-performance computing means multi-processor computing -- a phenomenon attested to by the success of these Supercomputing conferences. But this development and the emergence of MATLAB have been disjoint events, as MATLAB remains a language tied to a single processor.

Originally, MATLAB was conceived as an educational aid and as a tool for prototyping algorithms, which would then be translated into a "real" language. The justifications for this point of view were presumably that MATLAB's capabilities were limited and that, being interpreted, it could not achieve the performance of a compiled language. Over the years, however, the force of these arguments has diminished. So much MATLAB software is now available that MATLAB's capabilities can hardly be called narrow anymore; and as for performance, many users find that a degradation in speed by a factor between 1 and 10 is more than made up for by an improvement of programming ease by a factor between 10 and 100. In MATLAB, one effortlessly modifies the model, plots a new variable, or reformulates the problem in an interactive fashion. Such rapid real-time exploration is rarely feasible in Fortran or C.

Thus, increasingly, MATLAB has become a language for "real" computing by scientists and engineers. But one sense has remained in which MATLAB is only a system for education and prototyping. If one wants to take advantage of multiple processors, then one must switch to other languages. Experts, such as many of those participating in this conference, are in the habit of doing just this. Others, less familiar with the rapidly-changing complexities of high-performance computing, remain tied to their MATLAB desktops, isolated from the trend towards multiprocessors.

The vision of the MultiMATLAB project has been to bridge this gap. Think of a user who finds him- or herself computing happily in MATLAB, but frustrated by the time it takes to rerun the program for six different boundary conditions, or a dozen different parameter choices, or a hundred different initial guesses. Such a user's problems might be solved by a system that makes it convenient to spawn MATLAB processes on multiple processors of a parallel computer or a network of workstations or PCs. In many cases the needs for communication between the processors are rather small. Convenience of spreading the problem across machines and collecting the results numerically or graphically is paramount.

The MultiMATLAB project is exploring one approach for making this kind of computing possible. We do not at the outset aim for fine-grained parallelism or for peak performance of the kind needed for the grand challenge problems of computational science. Instead, following the philosophy that has made MATLAB so successful, we require reasonable efficiency but put the premium on ease of use. A key principle is that MATLAB itself -- not a home-grown facsimile, which would have little chance of keeping up with the ever-expanding features of the commercial product -- must be run on multiple processors. Our vision is that a user must be able to learn enough in five minutes to become intrigued by the system and begin to use it.

2. Using MultiMATLAB

2.1. Start, Nproc, Eval, ID

Each MultiMATLAB command begins with an initial upper-case letter. We illustrate how the system is used before describing its implementation.

Suppose the first author is sitting at her workstation in the Theory Center, connected to a node of the IBM SP2, running MATLAB. After a time she decides to start MATLAB on five new processors. She types

```
Start(5)
```

MATLAB is then started on five additional processors taken from a predetermined list. Or perhaps the second author is a sitting at a machine connected to Cornell's Computer Science Department network. He types

```
Start(['gemini'; 'orion'; 'rigel'; 'castor'; 'pollux'])
```

Now MATLAB is started on the five processors with the names indicated. (Some names could be repeated, in which case multiple MATLAB processes would be started on a single processor.) In either case, when all the processes are started the message is returned,

```
6 MultiMATLAB processes running.
```

This total number of processors can subsequently be accessed by the MultiMATLAB command `Nproc`.

The standard MultiMATLAB command for executing commands on one or more processors is `Eval`. If the user now types

```
Eval( 'sqrt(2)' )
```

then the MATLAB command `sqrt(2)` is executed on all six processors. The result is six repetitions of `1.4142`, which is not very interesting. On the other hand the command

```
Eval( 'ID' )
```

calls the MultiMATLAB command `ID` on each of the processors running. This command returns the number of the current process, an integer from 0 to `Nproc`-1. Running it on all nodes might give the result

```
ans = 0
ans = 1
ans = 5
ans = 2
ans = 3
ans = 4
```

The ordering of these numbers is arbitrary, since the processors are not synchronized and output is sent to the master process as soon as it is ready. (It is a good idea to type `Eval('format compact')` at the beginning to keep the output from the various processes as condensed as possible.) The command

```
Eval( 'ID^ID' )
```

might produce

```
ans = 1
ans = 1
ans = 256
ans = 3125
ans = 27
ans = 4
```

In the above examples, in keeping with our orientation toward SPMD programming, each command passed to `Eval` was executed on all MATLAB processes. Alternatively, one can select a subset of the processes by passing two arguments to the `Eval` command, the first being a vector of process IDs. Thus

```
Eval( [4 5] , 'cond(hilb(ID))' )
```

might return

```
ans = 1.5514e+04
ans = 4.7661e+05 ,
```

the condition numbers of the Hilbert matrices of dimensions 4 and 5, and

```
Eval( 0:2:4 , 'quad(''exp'',ID,ID+1)' )
```

might return

```
ans = 1.7183
ans = 93.8151
ans = 12.6965
```

the integrals of e^x from n to n+1 for n = 0, 2, and 4. Note how the double quote is used to obtain a string within a string. This calling of the MATLAB command `quad` gives a hint of the high-level power available that is so characteristic of MATLAB. In this case, adaptive numerical quadrature has been carried out to compute the desired integral. MATLAB users are accustomed to treating problems like integration, zerofinding, minimization, and computation of eigenvalues as routine matters to be handled silently by appropriate single-word commands.

None of these examples were costly enough for the use of multiple processors to serve much purpose, but it is easy to devise such examples. Suppose we want to find the spectral radii (maximum of the absolute values of the eigenvalues) of six matrices of dimension 400. The command

```
Eval( 'max(abs(eig(randn(400))))' )
```

does not do the trick; we get six copies of the number `20.8508`, since the random number generators deliver identical results on all processors. Preceding the eigenvalue computation by

```
Eval( 'randn(''seed'',ID)' ),
```

however, leads to the result desired:

```
ans = 20.9729
ans = 20.8508
ans = 21.0364
ans = 21.0312
ans = 21.6540
ans = 20.4072
```

(The spectral radius of an n by n random matrix is approximately the square root of n, for large n.) In a typical experiment this example might run in 23 seconds on six thin nodes of our SP2; the elapsed time would be six times greater if one used a `for` loop on a single machine. Of course, Monte Carlo experiments like this one are always the easiest examples of embarrassingly parallel computations.

For simplicity, the examples above call `Eval` with an explicit MATLAB command as an argument string. For most applications, however, a user will want to execute a program (an m-file) rather than a single line of text. A command such as

```
Eval( 'filename' )
```

achieves this effect.

2.2. Put, Get

So far, we have not communicated between processes except to send screen output to the master process. Of course, a nontrivial MultiMATLAB system depends on such communication.

One form of communication we have implemented is puts and gets, executable solely by the master MATLAB process. For example, the command

```
Put(1:4,'A'),
```

sends the matrix `A` from the master process 0 to processes 1 through 4; an optional argument permits the name of `A` to be changed at the destination. The command

```
Get(3,'B'),
```

gets the matrix `B` back from process 3 to the master.

2.3. Send, Recv, Probe, Barrier

More general point-to-point communication is accomplished by send and receive commands, which can be executed on any of the MATLAB processes. For example, the sequence

```
x = [pi pi^2];
Send(3,x)
Eval(3, 'Recv' )
```

passes a message containing a 2-vector from the master process to process 3, leading to the output

```
3.1416 9.8696
```

An optional argument can be added in `Recv` to specify the source. Another optional argument may be added in both `Send` and `Recv` to specify a message tag so as to ensure that sends and receives are properly matched and to aid in error checking. The command

```
Probe,
```

run on any process, again with optional source process number and message tag, returns 1 (true) if a message has arrived from the indicated source with the indicated tag, otherwise 0 (false).

SPMD programs can be built upon `Send` and `Recv` commands. Typically the program contains `if` and `else` commands that specify different actions for different processes. For example, suppose the m-file `cycle.m` consists of the following program:

```
if ID==0                  % first process: send
   a = 1
   Send(ID+1,a)
elseif ID == Nproc-1      % last process: receive and double
   a = 2*Recv
else                      % middle processes: receive, double, and send
   a = 2*Recv
   Send(ID+1,a)
end;
```

Process 0 creates the variable `a` with value 1 and sends it to process 1. Process 1 receives the message, doubles the value of `a`, and sends it along to process 2; and so on. If there are six processors the command `Eval( 'cycle' )` produces the output

```
a = 1
a = 2
a = 4
a = 8
a = 16
a = 32
```

The processes run asynchronously, but since each `Send` command is only executed after the corresponding `Recv` has completed, the proper sequence of computations and final value 32 are guaranteed so long as all of the nodes are functioning.

Alternatively, a MultiMATLAB command is available for explicit synchronization. The command

```
Barrier
```

returns only when called on each process.

2.4. Bcast, Min, Max, Sum

Although `Send` takes a vector of processor IDs as its destination list, the underlying idea is that of point-to-point communication. For more efficient communication between multiple processes, as well as greater convenience for the programmer, MultiMATLAB also has various commands for collective communication. These commands must be evaluated simultaneously on all processes.

The `Bcast` command is used to broadcast a matrix from one process to all other processes, using a tree-structured algorithm. For example,

```
Eval( 'Bcast(1,ID)' )
```

returns the number 1 on all processes. `Bcast` is much more efficient than a corresponding `Send` and `Recv`.

The same kind of a tree algorithm is used for various computations that reduce data from many processes to one. For example, the commands `Min`, `Max`, and `Sum` compute vectors obtained by reducing data over the copies of a vector or matrix located on all processors. Thus the command

```
Eval( 'Sum(1,[1 ID Nproc])' )
```

executed on six processes will return the vector

```
[6 15 36]
```

to process 1. If the first argument is omitted, the result is returned (broadcast) to all processes.

2.5. Higher-level MultiMATLAB Commands

The MultiMATLAB commands described so far represent communication primitives as they are used in the message-passing paradigm of programming. One of the aims of this project, however, is to provide also an interface at a higher level by building on these routines, hiding as much of the message passing as possible.

We can do this by taking a data-parallel approach in a simplistic fashion. We have developed a number of routines such as `Distribute` and `Collect` that allow a user to distribute a matrix or to collect a set of matrices into one large matrix. These functions operate using a mask that indicates which processors hold which portions of the matrix. This allows us also to develop routines such as `Shift` and `Copy` that are useful in data-parallel computing, keeping the communication to a more abstract level.

Additional geometry routines such as `Grid` and `Coord` have also been constructed that allow the user to create a grid of processors in 1,2 or 3 dimensions. These provide a powerful tool for more sophisticated parallel coding. An optional argument on the communication routines allows communication within a given set of nodes, for example along a column or row of the grid. We do not give further details, as these facilities are under development.

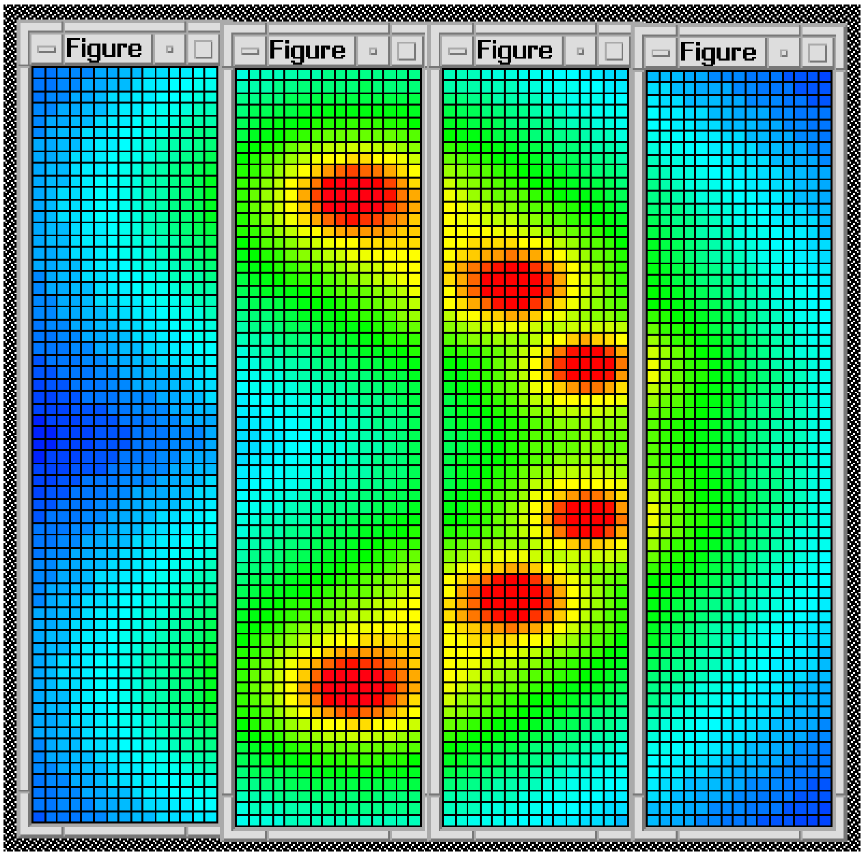
3. Multiprocessor Graphics

One of the great strengths of MATLAB is graphics. A primary goal of the MultiMATLAB project has been to ensure that this strength carries over to multiprocessor computations.

In many applications, the user will find it most convenient to compute on multiple processors but produce plots on the master process, after sending data as necessary. Equally often, however, it may be desirable to produce plots in a distributed fashion that are then sent to the user's screen. This can be particularly useful when one wishes to monitor the progress of computations on several processors graphically.

We have found the following simple method of doing this to be very useful. As mentioned above, many calculations with a geometric flavor divide easily into, say, four or eight subdomains assigned to a corresponding set of processors. We set up a MATLAB figure window in each process and arrange them in a grid on the screen. This is easily done using standard MATLAB handle graphics commands, and we expect shortly to develop MultiMATLAB commands for this purpose that are integrated with the grid operations mentioned earlier.

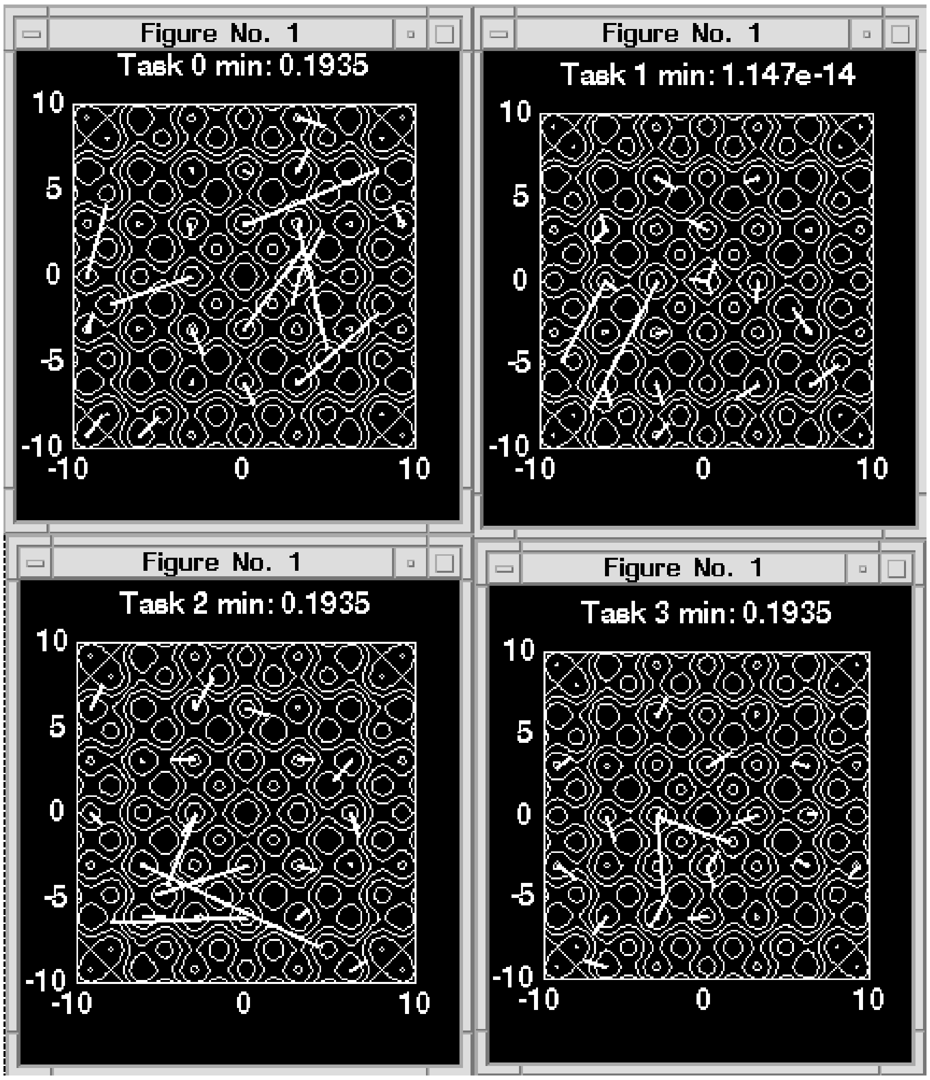
The figure below shows an example of this kind of computing; in this case we have a 4 by 1 grid of windows. In this particular example, what has been computed are the pseudospectra of a 64 by 64 matrix known as the "Grcar matrix" [17]. This is an easy application for MultiMATLAB since the computation requires a very large number of floating point operations (1024 singular value decompositions of dimension 64 by 64) but minimal communication (just the global minimum and maximum of the data with `Min` and `Max`, so that all panels can be on the same scale).



Another kind of application that might benefit from this kind of graphics would be as follows. Suppose we wish to solve the wave equation by an explicit finite difference scheme and watch waves bounce around in the computational domain. It is a straightforward matter to divide the computation into a grid of processors as in the figure above, communicating just one row or column of boundary data to adjacent processors at each step. Waves can then be seen to propagate from one window to another. This kind of visualization can be very convenient for interactive experimentation, and higher-quality plots can be produced at selected time steps as necessary by sending data to a single processor.

Our second computed example illustrates the use of multiple figure windows for monitoring a process of numerical optimization. MATLAB contains powerful programs for minimization of functions of several variables; one of the original such programs is `fminu`. Unfortunately, such programs generally find local minima, not global ones. If one requires the global minimum it is customary to run the search multiple times from distinct initial points, which in many cases might as well be taken to be random. With sufficiently many trials leading to a single smallest minimum found over and over again, one acquires confidence that the global minimum has been found, but the cost of this confidence may be considerable computing time.

Such a problem is easily parallelizable, and the next figure shows a case in which it has been distributed to four processors. A function *f(x,y)* of two variables has been constructed that has many local minima but just one global minimum, the value 0 taken at the origin. On each of four processors, the optimization is carried out from twenty random initial points, and the result is displayed in the corresponding figure window as a straight line from the initial guess to the converged value. The background curves are contours of the objective function *f(x,y)*. Note that in three of the windows, the smallest value obtained is *f(x,y)*=0.1935, whereas the fourth window has found the global minimum *f(x,y)*=0.



In these examples we have set up a grid of windows, one to each processor. As an alternative it might be desirable sometimes to have multiple MATLAB processes all draw to one common window. This arrangement is possible within XWindows, for example. However, it is not possible within MultiMATLAB at present, because a figure's window ID is a read-only property in the current version of MATLAB, which cannot be set or reset by the user.

4. Implementation of MultiMATLAB

MultiMATLAB is built upon MPI (Message Passing Interface), a highly functional and portable message passing standard [7, 13]. Here is a brief description of how the system is put together.

The system is written using MPICH, a popular and freely available implementation of MPI developed at Argonne National Laboratory and Mississippi State University [6]. In particular, MultiMATLAB uses the P4 communication layer within MPICH, allowing it to run over a heterogeneous network of workstations. In building upon MPICH, we believe we have developed a portable and extensible system, in that anyone can freely get a copy of the software and it will run on many systems. Versions of MPICH are beginning to become available that run on PCs running Windows, and we expect soon to experiment with MultiMATLAB on those platforms.

The MultiMATLAB `Start` command builds a P4 process group file of remote hosts, which are either explicitly specified by the user or taken from a default list, and then initializes MPICH. MATLAB processes are then started on the remote hosts. Each process iterates over a simple loop, waiting for and executing commands received from the user's interactive MATLAB process. The user may use a `Quit` command to shut down the slaves and exit MultiMATLAB. Additionally, if the user quits MATLAB during a MultiMATLAB session, the slaves are automatically shut down.

One limitation of MPI, which was not designed for this particular kind of interactive use, is that a running program cannot spawn additional processes. A consequence of this limitation is that once MultiMATLAB is running on multiple processors, it is not possible to add further processors to the list except by quitting and starting again. It is expected that this limitation of MPI will be removed in the extension of MPI under development known as MPI 2.

At the user level, MultiMATLAB consists of a collection of commands such as `Send`, for example. Such a command is written as a C file called `Send.c`, which is interfaced to MATLAB via the standard MATLAB Fortran/C/C++ interface system known as MEX. Within MPI, many variants on sends and receives are defined. MultiMATLAB is currently built upon the standard send and receive variants, which employ buffered communication for most messages and synchronous communication for very large ones. Our underlying MPI sends and receives are both blocking operations, to ensure that no data is overwritten, but to the MultiMATLAB programmer, the semantics is that `Recv` is blocking while `Send` is non-blocking.

Higher-level MultiMATLAB commands are usually built on higher-level MPI commands. For example, `Bcast` and `Min` and `Max` and `Sum` are built on MPI collective communication routines, and `Grid` and `Coord` are built on MPI routines that support cartesian topologies.

It should be stressed that MultiMATLAB allows MPI routines direct access to MATLAB data. As a result, MultiMATLAB does not incur any extra copying costs over MPICH, so it is reasonable to expect that its efficiency should be comparable. Our experiments show that this is indeed approximately the case. Here are the results of a typical experiment:

size of matrix (# of doubles)	round-trip latency (milliseconds) MPICH	MultiMATLAB
25	2.5	4.7
50	2.1	6.7
100	2.8	12.6
200	4.4	15.1
400	9.3	20.0
800	18.2	21.1
1600	35.8	38.4
3200	80.8	81.9
6400	165.8	175.7
12800	339.6	360.8
25600	708.9	698.7
51200	1397.4	1406.0
102400	2744.7	2850.3

The table compares round-trip latencies for a MultiMATLAB code with those for an equivalent C code using MPICH, and reveals that MultiMATLAB does add some overhead to that of MPICH. The timings were obtained on the IBM SP2, not using the high-performance switch. This occurs because MATLAB performs memory allocation for received matrices. It might be possible to alleviate this problem by maintaining a list of preallocated buffers, but we have not pursued this idea.

5. Related Work

Many people must have thought about parallelizing MATLAB over the years. According to Moler's essay "Why there isn't a parallel MATLAB," published in the MathWorks Newsletter in 1995 [14], he was involved with one of the earliest such attempts in the mid-1980s on an Intel iPSC. Of course, a great deal has happened in distributed computing since then.

Our own first experiments were carried out in 1993 (A. E. Trefethen). By making use of a Fortran wrapper based on IBM's message passing environment (MPL), we ran MATLAB on multiple nodes of an IBM SP-1. We were impressed with the power of this system for certain fluid mechanics calculations, and this experience ultimately led to our persuading The MathWorks to support us in initiating the present project.

We are aware of seven projects than have been undertaken elsewhere that share some of the goals and capabilities of MultiMATLAB. We shall briefly describe them.

The longest-standing related project, dating to before 1990, is the CONLAB (CONcurrent LABoratory) system of Kågström and others at the University of Umeå, Sweden [4,10]. CONLAB is a fully-independent system with MATLAB-like notation that extends the MATLAB language with control structures and functions for explicit parallelism. CONLAB programs are compiled into C code with a message passing library, PICL [5], and the node computations are done using LAPACK.

A group at the Center for Supercomputing Research and Development at the University of Illinois has developed FALCON (FAst Array Language COmputatioN), a programming environment that facilitates the translation of MATLAB code into Fortran 90 [2,3]. FALCON employs compile time and run time inference mechanisms to determine variable properties such as type, structure, and size. Although

FALCON does not directly generate parallel code, the future aim of this project is to annotate the generated Fortran 90 code with directives for parallelization and data distribution. A parallelizing Fortran compiler such as Polaris [1] may then use these directives to generate parallel code.

Another project, from the Technion in Israel, is MATCOM [12]. MATCOM consists of a MATLAB-to-C++ translator and an associated C++ matrix class with overloaded operators. At present, MATCOM translates MATLAB only into serial C++, but one might hope to build a distributed C++ matrix class underneath it which would adhere to the same interface as the existing matrix class.

A project known as the Alpha Bridge has been developed by Alpha Data Parallel Systems, Ltd., in Edinburgh, Scotland [11]. Originally, in a system known as the MATLAB-Transputer-Bridge, this group ran a MATLAB-like language in parallel on each node of a transputer. The Alpha Bridge system is an enhancement of this idea in which high-performance RISC processors are linked in a transputer network. A reduced, MATLAB-like interpreter runs on each node of the network under the control of a master MATLAB 4.0 process running on a PC.

A fifth project has been undertaken not far from Cornell at Integrated Sensors, Inc. (ISI) in Utica, NY, a consulting company with close links to the US Air Force Rome Laboratories [9]. Here MATLAB code is translated to C code with parallel library routines. This project (and product) aims at executing MATLAB-style programs in parallel for real-time control and related applications.

The final two projects we shall mention, though not the most fully developed, are the closest to our own in concept. One is a system built by a group at the Universities of Rostock and Wismar in Germany [15,16]. In this system MATLAB is run on various nodes of a network of Unix workstations, with message passing communication via the authors' own system PSI/IPC based on Unix sockets.

Finally, the Parallel Toolbox is a system developed originally by graduate students Pauca, Liu, Hollingsworth, and Martinez at Wake Forest University in North Carolina [8]. This system is based upon the message passing system known as PVM. In the Parallel Toolbox, there is a level of indirection not present in MultiMATLAB between the MATLAB master process and the slaves, a PVM process known as the PT Engine Daemon. Besides handling the spawning of new processes, the PT Engine Daemon also filters input and output, sending error codes to a PT Error Daemon that logs the error messages to a file.

In summarizing these various projects, the main thing to be said is that most of them involve original implementations of a MATLAB-like language rather than the use of the existing MATLAB system itself. There are good reasons for this, if one's aim is high performance and an investigation of what the "ideal" parallel MATLAB-like system might look like. The disadvantage is that the existing MATLAB product is at present so widely used, and so extensive in its capabilities, that it may be unrealistic and inefficient to try to duplicate it. Instead, our decision has been to build upon MATLAB itself and produce a prototype that users can try as an extension to their current work rather than an alternative to it. As mentioned, this approach has also been followed by the Rostock/Wismar and Wake Forest University projects, using PVM or another message passing system rather than MPI.

6. Conclusions

MultiMATLAB can be summarized in a few words. We run MATLAB processes on multiple processors, with full access to all the usual capabilities such as Toolboxes. These processes communicate via simple MATLAB-style commands built on MPI, with all message-passing details hidden as far as possible from the user. Both master/slave and SPMD paradigms are implemented, and attention is paid to multiprocessor graphics. All of this happens without any changes in the MATLAB architecture; indeed, we have not had access to the MATLAB source code.

It is a straightforward matter to install our current software on any network of Unix workstations or SP2 system, provided that all the nodes are licensed to run MATLAB and there is a shared file system. We expect that extensions to networks of PCs running Windows, based on appropriate implementations of MPI, are not far behind. We hope to make our research code publicly available in the near future and will announce this event on the NA-Net electronic distribution list and elsewhere. Based on reactions of users so far, we think that MultiMATLAB will prove appealing to many people, both for enhancing the power of their computations and as an educational device for teaching message passing ideas and parallel algorithms. It gives MATLAB users easy access to message passing, here and now. The parallel efficiency is not always as high as might be achieved, but for many applications it is surprisingly good. We hope to address questions of performance in more detail in a forthcoming technical report.

MultiMATLAB is by no means in its final form. This is an evolving project, and various improvements in functionality, for example in the areas of collective communications and higher-level abstractions, are under development. The current system also needs improvement in the area of robustness with respect to various kinds of errors, and in its documentation. We are guided in the development process by several projects underway in which MultiMATLAB is being used by our colleagues for scientific computations.

As we have mentioned in the text, several projects related to MultiMATLAB are being pursued at other institutions, including CONLAB, FALCON, the Parallel Toolbox, and others. Though the details of what will emerge in the next few years are of course not yet clear, we believe that the authors of all of these systems join us in expecting that it is inevitable that the MATLAB world will soon take the step from single to multiple processors.

References

[1] W. Blume, et al. Effective Automatic Parallelization with Polaris. International Journal of Parallel Programming. May 1995.

[2] L. De Rose, et al. FALCON: An environment for the development of scientific libraries and applications. Proc. First Intl. Workshop on Knowledge-Based Systems for the (re)Use of Program Libraries, Sophia Antipolis, France, November 1995.

[3] L. De Rose, et al. FALCON: A MATLAB interactive restructuring compiler. In Languages and Compilers for Parallel Computing, pp. 269-288. Springer-Verlag. August, 1995.

[4] P. Drakenberg, P. Jacobson, and B. Kågström. A CONLAB compiler for a distributed memory

multicomputer. R. F. Sincovec, et al., eds., Proc. Sixth SIAM Conf. Parallel Proc. for Sci. Comp., v. 2, pp. 814-821. 1993.

[5] G. A Geist, et al. PICL: A portable instrumented communication library. Tech. Rep. ORNL/TM-11130, Oak Ridge Natl. Lab., 1990.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing, to appear.

[7] W. Gropp, E. Lusk, and A. Skjellum. Using MPI. MIT Press. 1994.

[8] J. Hollingsworth, K. Liu, and Paul Pauca. Parallel Toolbox for MATLAB PT v. 1.00: Manual and Reference Pages. Wake Forest University. 1996.

[9] Integrated Sensors, Inc. home page: http://www.sensors.com.

[10] P. Jacobson, B. Kågström, and M. Rännar. Algorithm development for distributed memory multicomputers using CONLAB. Scientific Programming, v. 1, pp. 185-203. 1992.

[11] J. Kadlec and N. Nakhaee. Alpha Bridge, parallel processing under MATLAB. Second MathWorks Conference. 1995.

[12] MATCOM, March 1996 release. http://techunix.technion.ac.il/~yak/matcom.html.

[13] Message Passing Interface Forum. MPI: A message-passing interface standard. Intl. J. Supercomputer Applics., v. 8. 1994.

[14] C. Moler. Why there isn't a parallel MATLAB. MathWorks Newsletter. Spring, 1995.

[15] S. Pawletta, T. Pawletta, and W. Drewelow. Distributed and parallel simulation in an interactive environment. Preprint, University of Rostock, Germany. 1995.

[16] S. Pawletta, T. Pawletta, and W. Drewelow. Comparison of parallel simulation techniques -- MATLAB/PSI. Simulation News Europe, v. 13, pp. 38-39. 1995.

[17] L. N. Trefethen. Pseudospectra of matrices. In D. F. Griffiths and G. A. Watson, Numerical Analysis 1991, Longman, pp. 234--266. 1992.

About the Authors

Anne Trefethen is Associate Director for Scientific Computational Support at the Cornell Theory Center. From 1988 to 1992 she worked at Thinking Machines, Inc., where she was one of the developers of the Connection Machine Scientific Software Library.

Vijay Menon, interested in parallelizing compilers, is a PhD student of Keshav Pingali in the Computer Science Department at Cornell.

Chi-Chao Chang and Greg Czajkowski, interested in runtime systems, are PhD students of Thorsten von Eicken in the Computer Science Department at Cornell.

Chris Myers is a Research Scientist at the Cornell Theory Center. His research interests are in condensed matter physics and scientific computing.

Nick Trefethen, a Professor in the Department of Computer Science at Cornell, has been using MATLAB since 1980. His research interests are in numerical analysis and applied mathematics.

Acknowledgments

For advice and comments concerning both the MultiMATLAB project and this paper, we are grateful to Toby Driscoll, Bill Gropp, Xiaoming Liu, Cleve Moler, Barry Smith, Steve Vavasis, and Thorsten von Eicken.

This research was supported in part by The MathWorks, Inc. It was conducted in part using the resources of the Cornell Theory Center, which receives major funding from the National Science Foundation (NSF) and New York State, with additional support from the Defence Advanced Research Projects Agency (DARPA), the National Center for Research Resources at the National Institutes of Health (NIH), IBM Corporation, and other members of the center's Corporate Partnership Program. Further support has been provided by NSF Grant DMS-9500975 and DOE Grant DE-FGO2-94ER25199 (L. N. Trefethen), NSF Grant CCR 9503199 (support of Menon by Pingali), ARPA Grant N00014-95-1-0977 (support of Czajkowski by von Eicken) and a Doctoral Fellowship (200812/94-7) from the Brazilian Research Council (Chang).