The Wayback Machine - https://web.archive.org/web/20000304122936/http://www.tc.cornell.edu:80/~anne/projects/MM.html

Contents

Intro page

History

MultiMATLAB Version 1

Examples

Wave Calculation

Speckle Reduction

Pseudospectra computation

Distributed Contouring

Related Material

Related projects

Technical Report

A Financial Application

SC '97 Paper

Acknowledgments

MultiMATLAB

Purpose

The aim of the Cornell MultiMATLAB project is to provide a tool that facilitates the use of a parallel computer, or a network of computers, to solve course-grained large-scale problems using MATLAB. Our hope is that MultiMATLAB will shorten the development cycle of a serial MATLAB code to a parallel implementation and thereby provide a bridge to parallel computing, whether on a high-performance computer system or a network of workstations. The MultiMATLAB project is a research project carried out jointly by the Cornell Theory Center and the Computer Science Department at Cornell University, with support from The MathWorks, Inc.

For further information contact Anne E. Trefethen

MultiMATLAB

Purpose

The aim of the Cornell MultiMATLAB project is to provide a tool that facilitates the use of a parallel computer, or a network of computers, to solve course-grained large-scale problems using MATLAB. Our hope is that MultiMATLAB will shorten the development cycle of a serial MATLAB code to a parallel implementation and thereby provide a bridge to parallel computing, whether on a high-performance computer system or a network of workstations. The MultiMATLAB project is a research project carried out jointly by the Cornell Theory Center and the Computer Science Department at Cornell University, with support from The MathWorks, Inc.

For further information contact Anne E. Trefethen

The Wayback Machine - https://web.archive.org/web/20031118185911/http://www.cs.cornell.edu/Info/People/vsm/papers/sc97

MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing

Vijay Menon
Department of Computer Science
Cornell University, Ithaca, NY 14853
vsm@cs.cornell.edu
http://www.cs.cornell.edu/Info/People/vsm

Anne E. Trefethen
Cornell Theory Center
Cornell University, Ithaca, NY 14853
anne@tc.cornell.edu
http://www.tc.cornell.edu/~anne

Abstract:

MATLAB is the most popular scientific computing environment available on uniprocessors today. Unfortunately, no such environment is currently available for multiprocessors. MultiMATLAB [1] is a general extension of the MATLAB environment to any distributed memory multiprocessors. This paper presents a new MultiMATLAB system designed to provide high-performance on multiprocessors while maintaining the functionality and usability of the MATLAB environment. This system will enable users to access high-performance parallel routines from within the MATLAB environment, to extend the environment with new parallel routines, and to use these routines to develop parallel applications with the MATLAB language. We discuss a general MultiMATLAB architecture, present two implementations based upon the MPI communication standard [2], and demonstrate the use of this system. Preliminary results indicate that the MultiMATLAB system can offer the full performance of the underlying multiprocessor to the MATLAB environment.

1 Introduction

MATLAB has established itself as the numerical computing environment of choice on uniprocessors for hundreds of thousands of engineers and scientists. In particular, MATLAB offers several features that simplify numerical computing. First, it provides users with easy access to an extensive library of high quality numerical routines. Second, it is an intuitive, interactive environment that enables users to use these routines in a dynamic manner well suited to scientific problem solving. Third, it provides a high-level matrix based language that permits users to express computations in an exceptionally concise manner via matrix or vector formulations and often with a fraction of the code required in conventional languages such as C or Fortran. Fourth, MATLAB's graphical and visualization tools offer a simple but powerful mechanism to manipulate and analyze data more effectively. Finally, MATLAB enables users to extend the environment either by developing their own routines or applications in the MATLAB language or by acquiring routines or applications developed by others.

As might be expected with any interpreted language, the MATLAB language may not offer the same performance as compiled C or Fortran. However, since most high-level matrix computations in MATLAB are pre-compiled, optimized routines, the difference in performance is strongly application dependent. Users may also write computationally intensive components of their applications in C or Fortran that may be compiled, loaded, and executed from MATLAB. As a result, MATLAB is able to provide high performance in addition to usability and functionality on uniprocessors.

Nevertheless, for many scientific applications, the desired levels of performance are only obtainable on distributed memory multiprocessors. Unfortunately, there is no computing environment that provides the usability and functionality of MATLAB for multiprocessors. We argue that such an environment is realizable on a generic distributed memory multiprocessor using readily available software.

In [1], Trefethen, et al. present MultiMATLAB, a general extension of the MATLAB environment to distributed memory multiprocessors, as a first step in this direction. In particular, we demonstrate that MultiMATLAB achieves the following goals.

- **Familiarity:** The original MATLAB environment is preserved. The MultiMATLAB system is simply an extension of the MATLAB environment. The system does not alter the way users interact with MATLAB; it merely provides new functionality.
- **Interactiveness:** Users are provided interactive access to all processors in a multiprocessor within the MATLAB environment. This permits steering of parallel computation.
- **Programmability:** Users are given the tools to write parallel code purely in the MATLAB language. Moreover, parallel and distributed applications can be developed interactively in the MATLAB environment.
- **Portability:** The system should be easily portable onto any parallel platform that allows MATLAB to run on each processor. This includes any network of workstations supporting MATLAB. Since most modern fine-grain multiprocessors are built using commodity processors, it should also include most modern parallel platforms.

In this paper, we present a new MultiMATLAB architecture redesigned to achieve two further goals.

- **Performance:** The MultiMATLAB architecture now provides direct access from MATLAB to vendor-optimized communication libraries. Users should be able to replace computationally intensive routines within sequential MATLAB code with high-performance parallel routines and expect performance gains.
- **Extensibility:** Users or other third parties may extend the environment with new high-performance parallel routines written in C or Fortran.

In the remainder of this paper, we describe and evaluate the new MultiMATLAB architecture. Section 2 describes the general architecture. Section 3 presents two implementations of the MultiMATLAB system based on the MPI communication standard. Section 4 discusses the integration of high performance parallel routines into MATLAB. Section 5 explores different methods of parallel programming at the MATLAB level. Section 6 presents

examples and performance results. Section 7 discusses related work, and Section 8 discusses future work. Finally, we conclude with a summary of our results.

2 The MultiMATLAB Architecture

Figure 1: MultiMATLAB Architecture

\begin{figure*} \begin{center} \begin{minipage} {6in} \psfig {file=mmarch.ps,width=6.0in}\end{minipage}\end{center}\end{figure*}

In the MultiMATLAB architecture, illustrated in Figure 1, each processor in a parallel platform individually runs a MATLAB process. Each process is then provided with the ability to communicate with other processes through a communication layer that runs over the parallel platform's interconnection network. The user interacts directly with one MATLAB process, called the interactive process, and operates within that process's MATLAB environment. Other MATLAB processes await commands from the interactive process. These other processes are used either by explicitly sending MATLAB commands to them or by executing routines that, in turn, use them.

Figure 2: MATLAB Process Address Space

\begin{figure*} \begin{center} \begin{minipage} {3in} \psfig {file=mmas.ps,width=3.0in}\end{minipage}\end{cer

The key component in this architecture is the MultiMATLAB interface module. The interface module, shown in Figure 2, is responsible for initializing an underlying communication layer, such as MPI [2], PVM [3], BLACS [4], or any other package available on the platform, and exposing it to the rest of the system.

MEX routines, also shown in the Figure 2, are executables originally written in C or Fortran that may be run directly from MATLAB. All parallel functionality in the MultiMATLAB system is provided through MEX routines. As a simple example, the MultiMATLAB system provides users with an Eval routine for parallel evaluation. This routine allows users to execute commands on the other processes. On the interactive process, the Eval routine is implemented as a MEX routine which accepts a vector of MATLAB process IDs and a MATLAB command. This routine sends the commands over the underlying communication layer to the processes corresponding to the specified IDs. On non-interactive processes, a separate top-level MEX routine is immediately run upon initialization of the system. This MEX routine runs in a loop in which it waits for MATLAB commands to arrive from

the interactive process and executes them in its own MATLAB environment. All parallel MEX routines access the underlying communication layer and, hence, the network, through the MultiMATLAB interface module.

It is important to note that the interface module exists in the corresponding MATLAB process's address space. For this reason, the interface module does not impose any additional context switch when parallel MEX routines access the communication layer. This is of particular importance when the parallel platform provides access to the network in user space. This enables parallel applications to access the network without operating system intervention as is often required for best performance. By residing within the MATLAB process's address space, the MultiMATLAB interface module is able to preserve this property for parallel MEX routines.

The MultiMATLAB interface module provides MEX routines access to the underlying communication layer via an indirection table. Upon initialization, the interface module builds a table of pointers to all functions and data in the underlying communication layer that need to be exposed to parallel MEX routines. When a parallel MEX routine is first loaded and executed, it accesses the MultiMATLAB interface module through a function call to MATLAB and is given the location of this table. This is the only time the MEX routine needs to directly access the interface module. Once the table is obtained, the MEX routine has the capability to access any function provided by the underlying communication layer through its corresponding offset in the table. In general, the MultiMATLAB interface module adds an overhead of one load instruction per communication layer function call in a parallel MEX routine. As the execution time of an instruction is minimal compared to the latency of communicating a single packet of data, this overhead should be negligible.

3 System Implementations

We present two implementations of the MultiMATLAB architecture. Both implementations use MPI as their underlying communication substrate. While a specific communication standard is not fundamental to the architecture, we chose MPI due its broad popularity, its suitability to high performance parallel computing, and the virtually universal availability of vendor-optimized versions on modern parallel computers. One implementation of MultiMATLAB, designed to run on a generic network of Unix workstations, is based upon MPICH [5], a popular public domain version of MPI developed at Argonne National Laboratory and Mississippi State University. The second was designed for the IBM SP2, a modern high performance distributed memory multiprocessor. This implementation is based upon MPI-F [6], IBM's proprietary version of MPI specifically optimized for the SP2.

3.1 MPICH: Network of Workstations

The MPICH implementation of MultiMATLAB is based upon the P4 [7] communication subsystem, allowing it to run over a network of workstations connected by TCP/IP. In particular, this implementation should run on any platform providing both MATLAB and MPICH. To date, we have successfully run it on IBM RS6000 and Sun Sparc workstations and the IBM SP2. This implementation is not optimized for any particular platform: on the SP2, for instance, user space access to the high performance switch is not available.

In this implementation, the interactive MATLAB process is started by the user as a normal MATLAB session. The user is provided with a Start command that initializes the MultiMATLAB system. This 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. The remote hosts need not be unique; this implementation permits multiple MATLAB processes on the same processor. MATLAB processes are started on the remote hosts at this point, and wait for commands from the interactive process. A Quit

command enables the user to shut down remote MATLAB processes and deactivate the MultiMATLAB system. In addition, the system is automatically shutdown when either the interactive MATLAB is exited or the system has been idle for specified period.

3.2 MPI-F: IBM SP2

The MPI-F implementation of MultiMATLAB was design specifically for the IBM SP2. MPI-F is a highly optimized implementation of the MPI standard running over the SP2's high performance switch. This implementation permits access to the switch in user space.

In this implementation, all MATLAB processes are started via POE, IBM's Parallel Operating Environment. For each of *p* MATLAB processes, POE assigns an identification number between and *p*-1. The process numbered is considered the interactive MATLAB. All other processes wait for commands from the interactive MATLAB process.

This implementation was designed to demonstrate the performance potential of the MultiMATLAB system. In particular, it provides a platform in which parallel libraries or applications written in MPI can easily be integrated into MATLAB on the SP2.

4 Integrating Parallel Routines into MATLAB

An important goal of the MultiMATLAB system is to allow users to integrate high performance parallel routines into the MATLAB environment. In particular, users should have the ability to replace computationally intensive parts of their programs with corresponding parallel routines. For example, in MATLAB's Partial Differential Equation Toolkit, users may interactively create and solve rather general PDE systems using the finite element method. The PDE toolkit provides an intuitive graphical interface allowing users to specify geometries, differential equations, and boundary conditions. However, the computationally intensive core of this application consists of mesh generation and linear system solutions. With the MultiMATLAB system, these computations could be done with high performance parallel routines.

In this section, we demonstrate how parallel routines written in MPI can be integrated into MATLAB in our MPI-based MultiMATLAB implementations. The general method is write or rewrite routines as parallel MEX routines. As mentioned earlier, MATLAB's MEX interface allows C and Fortran programs to be run directly from MATLAB. The only requirement is that each program provide a gateway function to MATLAB. This gateway function is called directly from MATLAB and manages all communication of data between MATLAB and the MEX routine.

Parallel MEX routines may be run in a SPMD fashion on any subset of MATLAB processes via the Eval command described in Section 2. The MultiMATLAB interface module provides parallel MEX routines with access to the underlying communication layer. In our implementations, MEX routines are granted access to MPI routines. Moreover, the indirection imposed by the interface module is transparent to the programmer. There are macros, provided to all parallel MEX routines, which map MPI calls directly to the appropriate table offset while maintaining the syntax of MPI.

Figure 3: MultiMATLAB Round Trip Latencies on the IBM SP-2

\begin{figure*} \begin{center} \begin{minipage} {3in} \psfig {file=rtl.ps,width=4.5in} \end{minipage} \end{center}\end{figure*}

Figure 3 illustrates the overhead that the MultiMATLAB interface module introduces to communication. We compare the average round trip latency of data on the IBM SP2 between two processors in both a parallel MEX routine and a native C parallel program. The MEX routine accesses the communication layer through the interface module, while the stand-alone program is linked directly to communication layer. We measured latencies for contiguous buffers of data ranging from 8 bytes (or 1 double) to 2 megabytes. The SP2 communicates this data in packets of roughly 256 bytes over its high performance switch. Note that the latencies are virtually identical for the MEX routine and the native C program. These results demonstrate that, as expected, the overhead is negligible.

As a result, programmers familiar with MPI can conveniently develop parallel MEX routines for MATLAB with standard MPI function calls and expect the roughly same performance as similar native C programs. Moreover, libraries and applications already available in MPI can be easily rewritten and recompiled as MEX routines by simply providing a gateway function.

However, modifying and recompiling libraries is not a viable option when source code is not available. As an alternative, parallel libraries could be pushed down to the underlying communication layer. For example, on the IBM SP2, we can push Parallel ESSL [8], a parallel numerical library optimized for that platform, into the communication layer along with MPI-F. As a result, parallel MEX routines on the SP2 may directly call Parallel ESSL functions in addition to MPI functions. In this case, the library is not reimplemented as a MEX routine. Instead, the MultiMATLAB interface module provides access to the library to all parallel MEX routines.

5 Parallel Programming in MATLAB

While parallel MEX routines are capable of providing the same performance as regular MPI programs, they are, unfortunately, just as difficult to write. If users only require parallelism in a few common routines such as a linear system solver or a discrete Fourier transform, this may not be a problem. These are the types of MEX routines that are likely to be either provided with a MultiMATLAB implementation or by a third party. However, if programmers wish to develop new MEX routines, they must write code in either C or Fortran with MPI calls.

While this may be necessary for higher performance, it does not facilitate the use of MATLAB as a programming environment. A critical aspect of the MATLAB environment is the ability to quickly develop programs and steer computations. From the user's point of view, the time saved by programming in the MATLAB language can considerably offset any loss of program performance.

Often the time saved by programming within the MATLAB language offsets any loss of program performance in terms of user productivity.

In this section, we investigate different mechanisms and methods for parallel computing and program development purely within the MATLAB language. It is widely agreed that the MATLAB language simplifies sequential numerical computing. To extend this notion to parallel numerical computing, we provide users with the ability to interactively develop parallel MATLAB computations and functions. In particular, functions written in the MATLAB language, called m-files, may then be used in a fashion similar to parallel MEX routines.

We present three different paradigms of parallel programming in the MATLAB language. First, we present a master/slave paradigm based on communication of MATLAB data structures between the interactive MATLAB process and other processes, providing a very simple mechanism for coarse-grain distributed computing. Second, we present a SPMD paradigm based on the exposure of message passing in the MATLAB language, permitting finer grain communication of MATLAB data structures between any processes within SPMD programs written in MATLAB. Third, we present a distributed matrix paradigm based on underlying parallel numerical libraries and MATLAB 5.0's object oriented features. Each of these programming paradigms is facilitated by MEX routines that expose some functionality of the underlying communication layer to the MATLAB language.

Table 1: Selected commands in the MultiMATLAB system

\begin{tabular} {\vert l\vert l\vert } \hline \hline \multicolumn{2} {\vert c\ver... ...& Collects local data segments into a single global one.\\ \hline \end{tabular}

5.1 Master/Slave

The MultiMATLAB system can easily be thought of as a master/slave environment, where the interactive MATLAB process is the master and other MATLAB processes are slaves.

With the Eval command described in Section 2, the user already has the ability to send MATLAB commands to one or more slave MATLAB processes for execution. In addition, we provide two commands: Put, to copy any MATLAB data structure from the master MATLAB to one or more slaves, and Get, to retrieve a MATLAB data structure from a slave back to the master. As an example, an expensive eigenvalue computation can be performed on a slave process instead of the interactive process, freeing the user to perform other computations in the meantime. In the following code, we move the matrix *A* to the MATLAB process with ID 1 and instruct it to compute the eigenvalues.

```
>> Put(1,'A');
>> Eval(1,'w = eig(A)');
```

When the computation is finished, we may retrieve the eigenvalues back from the slave MATLAB process.

```
>> Get(1,'w');
```

At this point, the vector *w* is available on the master MATLAB process. Often, these simple functions are sufficient to develop many embarrassingly parallel applications or distributed applications. A master/slave paradigm is often adequate in situations where little or no communication is required. As in the above example, it is most appropriate when data can be sent and processed separately by slave processes and results can be retrieved at the end.

5.2 SPMD/Message Passing

In order to accommodate applications requiring a finer degree of communication, we provide a set of MATLAB message passing routines analogous to those in MPI. In particular, we provide point to point and collective communication MATLAB routines operating directly on MATLAB data structures. We list a subset of these routines in Table 1.

Although message passing is a relatively low-level concept, programming in MATLAB does simplify the process of developing message passing code. First, users may think in terms of matrices, matrix sections, or other MATLAB data structures instead of communication buffers. While the MATLAB communication routines are optimized for dense, real matrices that may be mapped almost directly to the underlying layer, they also work for arbitrary MATLAB data structures such as sparse matrices, cell arrays, and structures. The size and shape of communicated data structures are determined at runtime. Second, MATLAB communication routines do not expose any memory management issues to users. Routines do any allocation necessary for received data. Routines block until any data to be sent can safely be overwritten and any data to be received has arrived.

Figure 3 illustrates the overhead of message passing within the MATLAB language. We compare round-trip latencies on the IBM SP2 between MATLAB message passing and C with MPI. For the smallest data sizes, MATLAB message passing imposes roughly a factor of five overhead. However, for data of 32 kilobytes, e.g., a 64 by 64 real matrix, or greater, this overhead is only 30-40% over communication in C with MPI. As a result, the SPMD paradigm should be effective for applications exhibiting medium-grain communication. Further performance results are given in Section 6.

5.3 Distributed Matrices

The low-level details of the SPMD paradigm may make it undesirable to certain users and for certain applications. In particular, users must explicitly determine the necessary message passing and data distribution.

A possible alternative is to use parallel MEX routines that provide similar functionality to high-level MATLAB operations. These parallel MEX routines would perform matrix multiplications, solve for eigenvalues, compute Fourier transforms, and so on. In fact, these MEX routines could provide an interface identical to corresponding sequential MATLAB functions. In particular, they would distribute data from the interactive MATLAB process among all processes, perform the parallel computation, and collect results back to the interactive process.

While such MEX routines would be useful in many cases, they admit one serious flaw: the constant distribution and collection can severely limit performance in a sequence of parallel computations. As a simple example of this, consider developing an iterative algorithm such as conjugate

gradients. The primary computation in this algorithm consists of a matrix-vector multiplication at every iteration. However, the matrix is unmodified over all iterations. In an efficient parallel implementation, the matrix would remain distributed over the course of the algorithm.

We are currently investigating a paradigm based on distributed matrices. A distributed matrix is stored across all MATLAB processes. Each process holds a different portion of the matrix within a structure containing the local data and a descriptor denoting the data distribution of the entire matrix. Users may align MATLAB processes in a multi-dimensional grid and then distribute data across this grid. Currently, only block distributions are considered, but, in the future, it should be straightforward to include arbitrary block/cyclic distributions and irregular distributions to optimize performance. Parallel MEX routines can then perform high-level MATLAB operations on these distributed matrix structures. In fact, computations could actually be performed by a library such ScaLAPACK [9] or PLAPACK [10].

Distributed matrix structures permit the MultiMATLAB system to provide users an interactive interface to such parallel numerical libraries. In addition, implementing these structures with MATLAB 5.0's object-oriented features permits overloading of standard MATLAB operations for distributed matrices. The result is an intuitive and familiar interface to distributed matrices and corresponding parallel routines. As MATLAB was originally developed as an interface to EISPACK [11] and LINPACK [12], linear algebra packages for uniprocessors, this paradigm provides a natural extension of functionality to multiprocessors.

6 An Example: Conjugate Gradients

Figure 4: Conjugate Gradients

\begin{figure} \begin{tabular} {ll} \begin{minipage} {3in} {\scriptsize \begin{v...
...equential {\sc Matlab}& b. Message Passing {\sc Matlab}\end{tabular}\end{figure}

In this section, we describe different MultiMATLAB implementations of the conjugate gradients algorithm on the IBM SP2 at the Cornell Theory Center. The conjugate gradients algorithm is iterative method to solve a linear system of the form $Ax = b$, where A is a symmetric positive definite matrix and x and b are vectors. The computational core of this method is a single matrix-vector multiplication at each iteration. The sequential MATLAB code, adapted from [13], is shown in Figure 4a.

Figure 5: Sequential Conjugate Gradients on the IBM SP-2

\begin{figure*} \begin{center} \begin{minipage} {3in} \psfig {file=cg0.ps,width=4.5in}\end{minipage}\end{center}\end{figure*}

First, it is worthwhile to examine the performance of the sequential algorithm in MATLAB and C. Figure 5 compares the performance of both versions on different matrix sizes. The MATLAB version ranges from 5 to 20 times slower that the C version over the matrix sizes that are shown. Moreover, this gap steadily increases along with the matrix size. This degradation in conjugate gradients is due to the poor cache performance of MATLAB's matrix-vector multiplication for large matrices. It should be noted that, in the future, MATLAB's multiplication routine will be redesigned to use LAPACK [14], a high-performance numerical library for uniprocessors. In the meantime, however, MATLAB imposes a significant overhead compared to C code.

Figure 6: Matrix *A* and vector *x* distributed by row over *P* processors

\begin{figure*} \begin{center} \begin{minipage} {5in} \psfig {file=mvm.ps,width=5.0in} \end{minipage} \end{center}\end{figure*}

Figure 4b illustrates a parallel MATLAB implementation of conjugate gradients that uses the SPMD message passing described in Section 5.2. In this case, the matrix *A* and the different vectors are each distributed by row over all processes as in Figure 6. In each iteration, a process only does the computation required for its local data. However, communication is required for two different operations: the dot-products needed to compute *rho* and *alpha* and the matrix-vector multiply needed to compute $w\ local. The dot products only require communication of a single value over all processes. On the other hand, the matrix-vector multiplication needed for $w\ local requires the global vector *p*, and, thus, more expensive communication.

Figure 7: Parallel Conjugate Gradients on the IBM SP-2: $ matrix

\begin{figure*} \begin{center} \begin{minipage} {3in} \psfig {file=cg256.ps,width=4.5in}\end{minipage}\end{center}\end{figure*}

Figure 8: Parallel Conjugate Gradients on the IBM SP-2: $ 1024 \times 1024$ matrix

\begin{figure*} \begin{center} \begin{minipage} {3in} \psfig {file=cg1024.ps,width=4.5in}\end{minipage}\end{center}\end{figure*}

Figures 7 and 8 compare the performance of three different MultiMATLAB implementations with a stand-alone C implementation using MPI-F on up to 32 processors of the IBM SP2. Each implementation uses the same algorithm and data distribution described above. As predicted in Section 4, the performance of the MEX routine, written in C with MPI and run from MATLAB, is virtually indistinguishable from that of the stand-alone C code.

On the other hand, the MATLAB version using SPMD message passing, shown in Figure 4b, demonstrates similar overhead with respect to C versions as in the sequential case. Note, however, that as the number of processors is increased, the memory requirement of each processor is decreased, and the overhead between MATLAB and C decreases significantly. This behavior is consistent with that of the sequential case. In particular, for a 1024 by 1024 matrix, this MATLAB code exhibits almost linear speedup up to 32 processors, while the C versions achieve a slower rate of speedup, and none beyond 16 processors.

Another MATLAB implementation, based on the distributed matrix paradigm described in Section 5.3, removes the overhead due to MATLAB's matrix-vector multiplication. With MATLAB 5.0's object oriented features, this code is actually identically to that shown in Figure 4a. However, the matrix A and all vectors are, in fact, distributed objects, and the code must be run on all processors in a SPMD fashion. Moreover, the multiplication operation is actually implemented as a MEX routine handwritten in C with MPI. As a result, the communication and computation required by the parallel matrix-vector multiplication is folded into a single, more efficient routine. When compared to the MATLAB version with message passing, the result is a

dramatic increase in performance. In fact, for the 1024 by 1024 matrix, the performance of this version is virtually indistinguishable from the C versions.

These results demonstrate that the MultiMatlab system can be an effective platform for parallel computing. Message passing in Matlab offers a flexible means to implement different parallel algorithms with the Matlab language and to achieve substantial performance gains over sequential Matlab. For certain applications, distributed matrix objects may offer even better performance with less code change. Finally, MEX routines written with parallel C code can offer the performance of stand-alone parallel applications to the Matlab environment.

7 Previous and Related Work

The MultiMatlab project was established in 1995 with the goal of extending Matlab to run over multiple processers. From the beginning, the primary focus was programmability within Matlab with only coarse-grain performance. These efforts culminated in the first MultiMatlab architecture [1].

The first MultiMatlab architecture is different from the newer one described in this paper. It is more tightly coupled to MPICH [5] and its underlying P4 [7] layer resulting in some performance limitations. It cannot take advantage of vendor specific implementations of MPI, such as MPI-F [6], which can offer much better performance. Furthermore, it does not permit extensions via parallel MEX routines as described in Section 4. It does, however, already provide similar functionality to that shown in Table 1. It has been in use at the Cornell Theory Center since 1996 and is now available to the general public at http://www.tc.cornell.edu/~anne/projects/MM.html.

We are aware of two other projects [15,16,17] that also attempt to extend the Matlab environment to multiprocessors. Both provide systems built upon PVM, and both expose PVM functionality at the Matlab level. These systems are also designed for coarse-grain distributed computing. As far as we know, the architectures presented by these alternative parallel Matlab systems do not provide support for high-performance parallel routines within Matlab.

Another related area has been the development of Matlab language compilers. Given that the Matlab language is used for development by many scientists and engineers, it is not surprising that there have been several attempts to compile Matlab programs into production code. In particular, a number of projects have studied and implemented the compilation of Matlab programs into C, Fortran, and C++ [18,19,20,21,22]. In addition, a couple of projects relate to compilation of Matlab programs into high performance parallel code. The Falcon project [18] at Illinois is currently studying compilation of Matlab into High Performance Fortran [23]. RTExpress [24], from ISI, provides a system that facilitates the compilation of Matlab into C with MPI, ScaLAPACK, and other library calls. These projects differ from MultiMatlab in that they do not allow parallel programs to be run directly within the Matlab environment.

8 Future Directions

The MultiMatlab system is a step towards an effective parallel computing environment. There are number of promising directions to take to further improve this environment.

One direction is to provide a full-scale integration of a parallel numerical library, such as ScaLAPACK, with the MultiMATLAB system, as discussed in Section 5.3. A number of challenges remain in accomplishing this. In particular, efficient parallel numerical libraries generally expect data to obey certain distributions and to be stored in certain formats. As a result, a distributed matrix may need to be redistributed and stored in a manner compatable with the parallel routine. In addition, there are memory management issues. High performance parallel routines often destroy input data in order to save memory, contrary to normal MATLAB behavior. Parallel MEX routines would need to copy to avoid overwriting input data. In general, a balance must be found between the performance offered by parallel numerical libraries and the ease of use intrinsic to MATLAB.

Another promising direction is to develop or integrate a MATLAB language compiler with the MultiMATLAB system. As discussed in the previous section, compilation of the MATLAB language into C or Fortran has been studied and implemented. This technology can potentially be used to compile parallel programs written within the MATLAB language into much more efficient parallel MEX routines.

A further step in this direction would be the automatic parallelization of programs written in the MATLAB language. The same technology available in High Performance Fortran compilers should be applicable to MATLAB programs with similar annotations. In fact, related work [18] suggests that a MATLAB compiler may be able use the higher level of information in MATLAB programs to generate even better parallel code. This technology would significantly simplify the process of generating efficient parallel MEX routines in MATLAB.

One final direction is to study the MultiMATLAB architecture on shared memory multiprocessors. The architecture should be portable to shared memory platforms so long as MATLAB can be run on each processor. It has long been argued that shared memory programs are easier to develop than message passing programs. On such a platform, it would be useful to support shared memory programming within the MATLAB language.

Conclusion

We present the MultiMATLAB architecture as a promising design for a parallel numerical computing environment. As an extension to MATLAB, MultiMATLAB is able to provide users simple but effective access to parallel computing resources in a familiar and widely popular environment. This architecture enables MATLAB applications to take advantage of high performance distributed memory multiprocessors. Finally, it enables users to rapidly develop parallel programs and computations within the MATLAB environment.

Acknowledgements

The authors would like the acknowledge the contributions to this project made by Lloyd N. Trefethen, Nikola Valerjev, Chi-Chao Chang, Grzegorz Czajkowski, and Jim Steed.

This research was supported in part by The MathWorks, Inc. It was conducted in part using the resources of the Cornell Theory Center, which received major funding from the National Science Foundation (NSF) and New York State, with additional support from the Defense 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.

References

1

A. E. Trefethen, V. S. Menon, C. C. Chang, G. J. Czajkowski, C. Myers, and L. N. Trefethen.
MultiMatlab: Matlab on multiple processors.
Technical Report 96-239, Cornell Theory Center, 1996.
http://www.cs.cornell.edu/Info/People/lnt/multimatlab.html .

2

Message Passing Interface Forum.
A Message-Passing Interface Standard, May 1994.
http://www.mcs.anl.gov/mpi .

3

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam.
PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.
MIT Press, Cambridge, MA, 1994.

4

J. Dongarra and R. C. Whaley.
A user's guide to the BLACS.
Technical Report CS-95-281, Department of Computer Science, University of Tennessee, Knoxville, TN, 1995.
Also LAPACK Working Note No.94.

5

W. Gropp, E. Lusk, N. Doss, and A. Skjellum.
A high-performance, portable implementation of the MPI message passing interface standard.
Parallel Computing, July 1996.

6

H. Franke.
MPI programming environment for IBM SP1/SP2.
In *ICDCS '95*, Vancouver, 1995.

7

R. Butler and E. Lusk.
Monitors, messages, and clusters: The P4 parallel programming system.
Parallel Computing, April 1994.

8

IBM Corporation.
IBM Parallel Engineering and Scientific Subroutine Library. Release 2. Guide and Reference, 1996.

9

L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley.
ScaLAPACK: A portable linear algebra library for distributed memory computers - Design issues and performance.
In *Supercomputing '96*. ACM SIGARCH and IEEE Computer Society, 1996.
http://www.supercomp.org/sc96/proceedings/SC96PROC/DONGARRA/INDEX.HTM .

10
R. van de Geijn.
Using PLAPACK: Parallel Linear Algebra Package.
The MIT Press, 1997.

11
B.T. Smith, J.M. Boyle, Y. Ikebe, V.C. Klema, and C.B. Moler.
Matrix Eigensystem Routines: EISPACK Guide.
Springer-Verlag, New York, NY, second edition, 1970.

12
J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart.
LINPACK Users Guide.
Philadelphia, PA, 1978.

13
G. Golub and C. Van Loan.
Matrix Computations.
The Johns Hopkins University Press, 1996.

14
E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, editors.
LAPACK Users' Guide. Second Edition.
SIAM, Philadelphia, 1995.

15
J. Hollingsworth, K. Liu, and P. Pauca.
Parallel Toolbox for MATLAB.
Wake Forest University, 1996.
http://www.mthcsc.wfu.edu/pt/pt.html .

16
S. Pawletta, T. Pawletta, and W. Drewelow.
Distributed and parallel simulation in an interactive environment.

Technical report, University of Rostock, Germany, 1995.
Preprint.

17

S. Pawletta, T. Pawletta, and W. Drewelow.
Comparison of parallel simulation techniques - MATLAB/psi.
Simulation News Europe, 13:38-39, 1995.

18

L. De Rose and D. Padua.
A MATLAB to Fortran 90 translator and its effectiveness.
In *10th ACM International Conference on Supercomputing*, May 1996.

19

L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua.
FALCON: A MATLAB interactive restructuring compiler.
In *Languages and Compilers for Parallel Computing*, pages 269-288. Springer-Verlag, August 1995.

20

Y. Keren.
MATCOM: A MATLAB to C++ translator and support libraries.
Technical report, Israel Institute of Technology, 1995.

21

The MathWorks, Inc.
MATLAB Compiler, 1995.

22

P. Jacobson, B. Kågström, and M. Rämnnar.
Algorithm development for distributed memory multicomputers using CONLAB.
Scientific Programming, 1:185-203, 1992.

23

High Performance Fortran Forum.
High performance fortran language specification version 1.0, 1993.

24

Integrated Sensors Inc.
RTExpress.
http://www.sensors.com/Conf97 .

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

(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
```

```
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)' )
```

```
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.

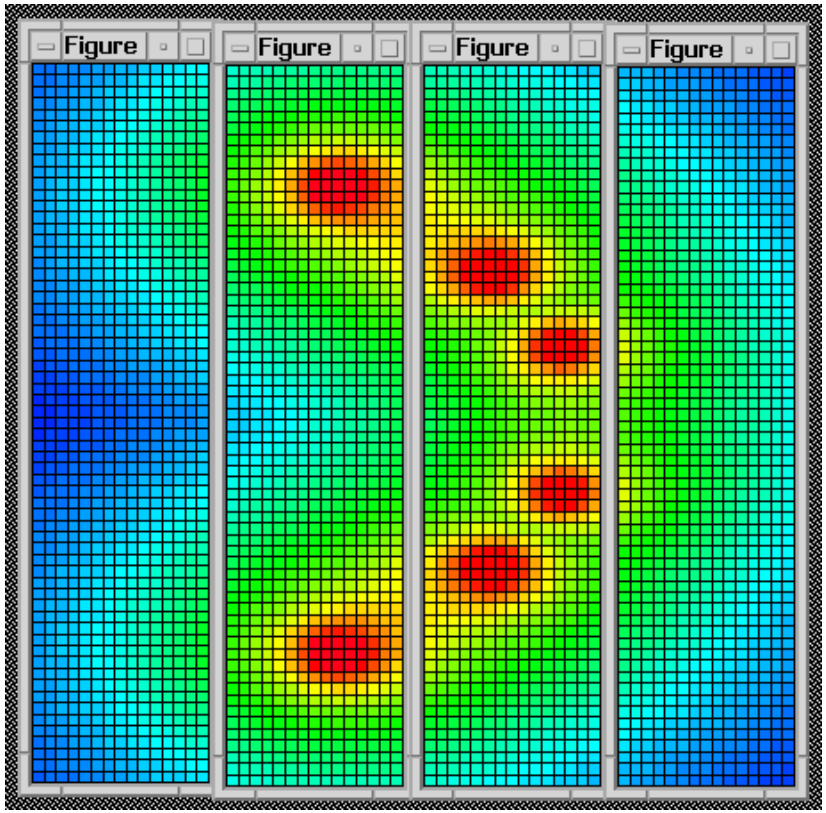
3. Multiprocessor Graphics

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.

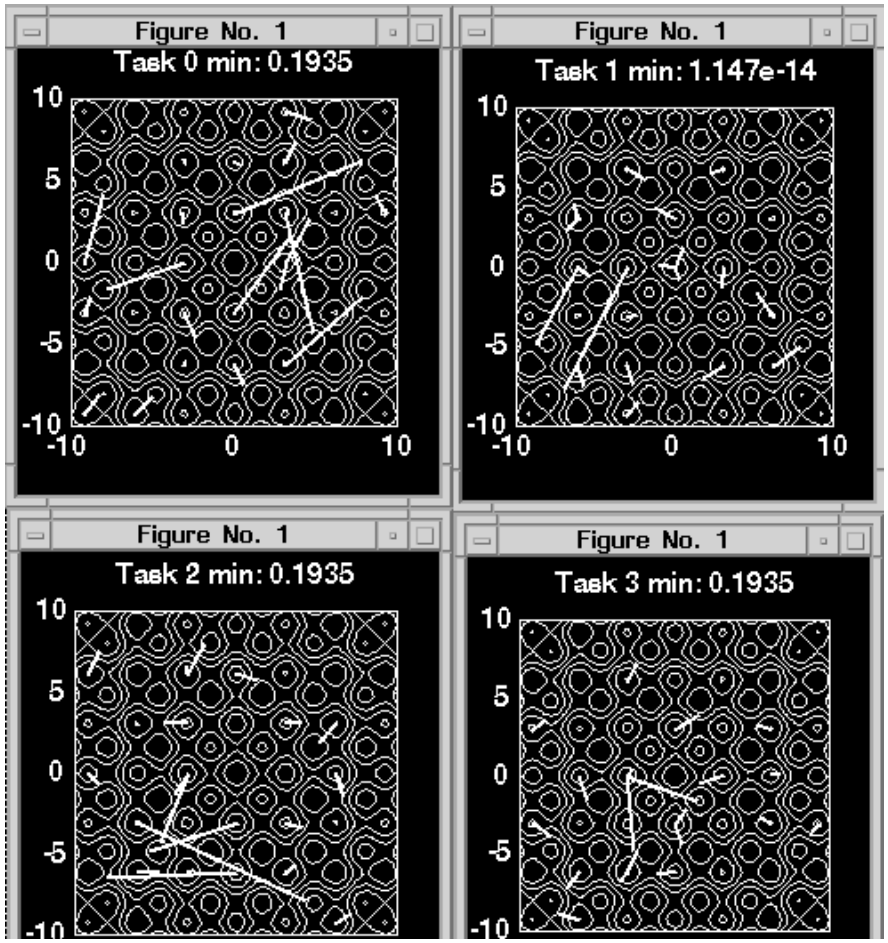
The figure below shows an example of this kind of computing; in this case we have a 4 by 1 grid of windows. In this particular example, what has been computed are the pseudospectra of a 64 by 64 matrix known as the "Grcar matrix" [17]. This is an easy application for MultiMATLAB since the computation 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

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).

The Wayback Machine - https://web.archive.org/web/20010128115400/http://www.cs.cornell.edu:80/Info/People/vsm/publications.htm

Recent and upcoming publications

Nikolay Mateev, Vijay Menon and Keshav Pingali. Left to Right and vice versa: Applying Fractal Symbolic Analysis to Restructuring Linear Algebra Codes. Submitted for publication.

Nikolay Mateev, Vijay Menon and Keshav Pingali. Fractal Symbolic Analysis for Program Transformations. Submitted for publication.

Vijay Menon and Keshav Pingali. A Case for Source-Level Transformations in MATLAB. In *The 2nd Conference on Domain-Specific Languages* The USENIX Association. Austin, Texas. October, 1999.

Vijay Menon and Keshav Pingali. High-Level Semantic Optimization of Numerical Codes. In *The 1999 ACM Conference on Supercomputing*. ACM SIGARCH. Rhodes, Greece. June, 1999.

VIjay Menon and Anne Trefethen. MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing. In *Supercomputing 1997*. IEEE Computing and ACM SIGARCH. San Jose, California. November, 1997.

Anne E. Trefethen, Vijay S. Menon, Chi-Chao Chang, Grzegorz Czajkowski, Chris Myers and Lloyd N. Trefethen. MultiMATLAB: MATLAB on Multiple Processors. Technical Report TR96-1586, Cornell University, Computer Science. May, 1996.

Chi-Chao Chang, Grzegorz Czajkowski, Xiaoming Liu, Vijay Menon, Chris Myers, Anne Trefethen and Lloyd N. Trefethen. The Cornell MultiMATLAB Project. In the *Proceedings of The 1996 Parallel Object-Oriented Methods and Applications Conference*. Santa Fe, New Mexico. February, 1996.

Talks

Symbolic Program Transformation for Numerical Codes *Motorola Labs*, Schaumburg, Illinois, 1/21/2000.

A Case for Source-Level Transformations in MATLAB *Usenix Conference on Domain Specific Languages*, Austin, Texas, 10/3/1999.

High-Level Semantic Optimization of Numerical Codes *ACM International Conference on Supercomputing* , Rhodes, Greece, 6/24/1999.

A Framework for Profile-driven Compilation in the MIPSPro Compiler. *Silicon Graphics, Inc.* , Mountain View, California, 9/4/1998.

MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing *IEEE/ACM Conference on Supercomputing* , San Jose, California, 11/19/1997.

Static Code Size Reduction for IA64. *Intel MRL* , Santa Clara, California, 8/23/1996.

Vijay Menon
Last modified: Tue Feb 15 22:38:24 EST 2000