MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing

Vijay Menon
Department of Computer Science
Cornell University
Ithaca, NY 14853
vsm@cs.cornell.edu

Anne E. Trefethen
Cornell Theory Center
Cornell University
Ithaca, NY 14853
anne@tc.cornell.edu

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.

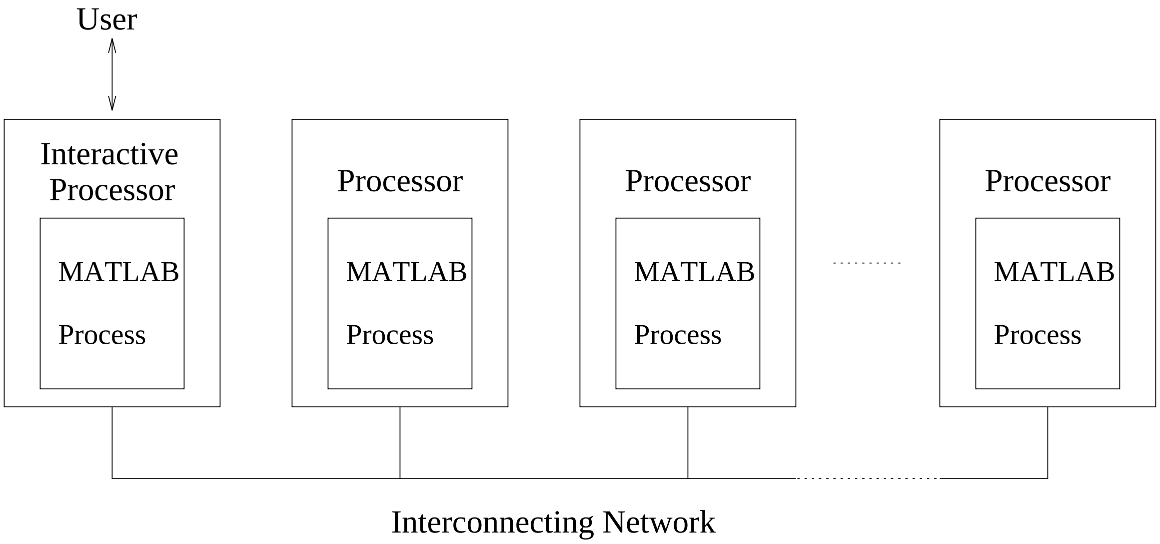

Figure 1: MultiMATLAB Architecture

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

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.

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 sys-

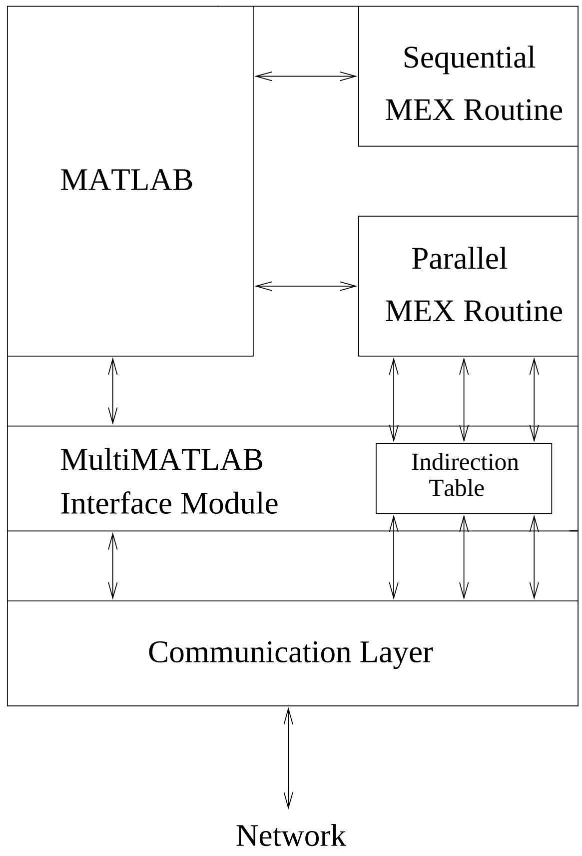

Figure 2: MATLAB Process Address Space

tem 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 0 and $p-1$. The process numbered 0 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 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

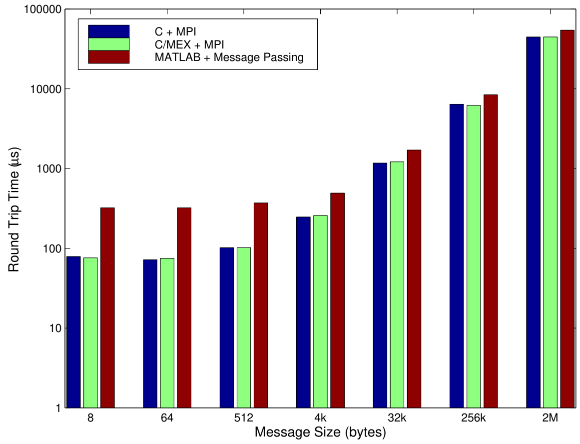

Figure 3: MultiMatlab Round Trip Latencies on the IBM SP-2

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.

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');
```

MultiMATLAB commands	
Starting and stopping MultiMATLAB (MPICH only)	
Start(n)	Initializes n remote MATLAB processes.
Quit	Closes remote MATLAB processes.
Running commands on multiple processors	
Nproc	Returns number of MATLAB processes.
ID	Returns ID of calling MATLAB process.
Eval(pid,'command')	Evaluates a command on one or more MATLAB process.
Master/Slave Communication	
Put(pid,dataname)	Put data from the master process onto one or more remote processes.
Get(pid,dataname)	Get data from a remote process onto the master process.
SPMD Communication	
Send(pid,data)	Send data from one process to another.
Recv(pid)	Receive data sent from another process.
Barrier	Synchronize processes.
Bcast(pid,data)	Broadcasts data from processor pid to all processes.
Sum(data)	Adds data across all processors to form a global sum.
Collect(data)	Collects local data segments into a single global one.

Table 1: Selected commands in the MultiMATLAB system

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

```
r = b;
rho = r'*r;
k = 0;
while((k < kmax))
  k = k+1;
  if (k == 1)
    p = r;
  else
    beta = rho/oldrho;
    p = r + beta*p;
  end

  w = A * p;

  alpha = rho/(p'*w);
  x = x + alpha * p;
  r = r - alpha * w;
  oldrho = rho;
  rho = r'*r;
end
```

a. Sequential MATLAB

```
r_local = b_local;
rho = Sum(r_local'*r_local);
k = 0;
while((k < kmax))
  k = k+1;
  if (k == 1)
    p_local = r_local;
  else
    beta = rho/oldrho;
    p_local = r_local + beta*p_local;
  end

  p = Gather(p_local);
  w_local = A_local * p;

  alpha = rho/(Sum(p_local'*w_local));
  x_local = x_local + alpha * p_local;
  r_local = r_local - alpha * w_local;
  oldrho = rho;
  rho = Sum(r_local'*r_local);
end
```

b. Message Passing MATLAB

Figure 4: Conjugate Gradients

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.

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 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

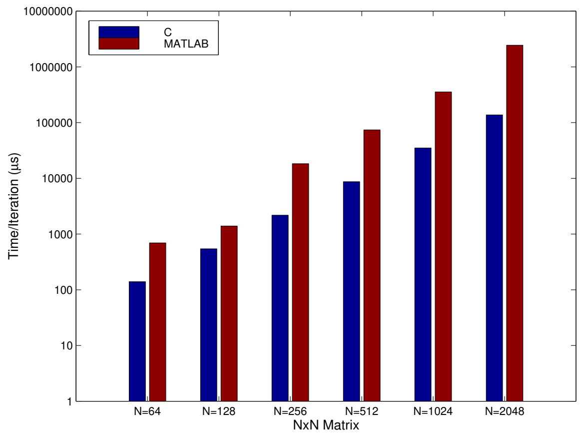

Figure 5: Sequential Conjugate Gradients on the IBM SP-2

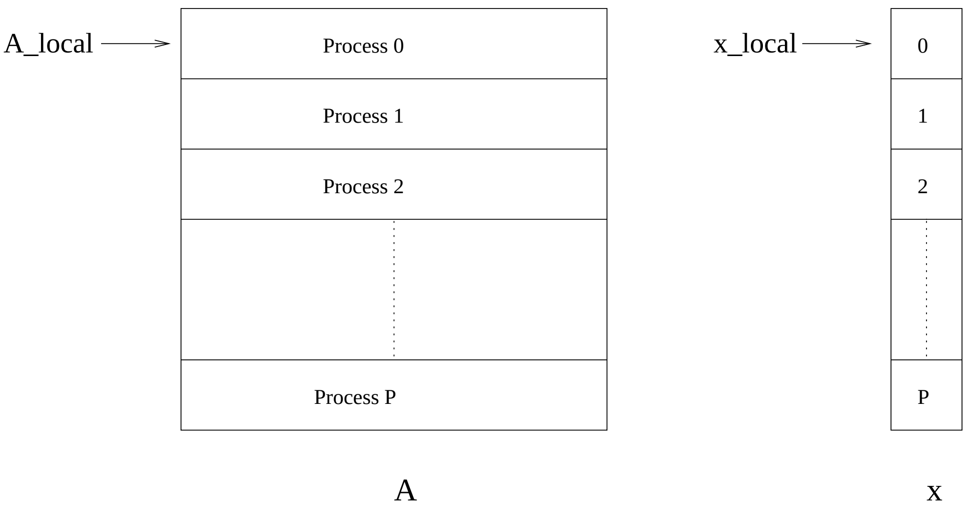

Figure 6: Matrix A and vector x distributed by row over P processors

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.

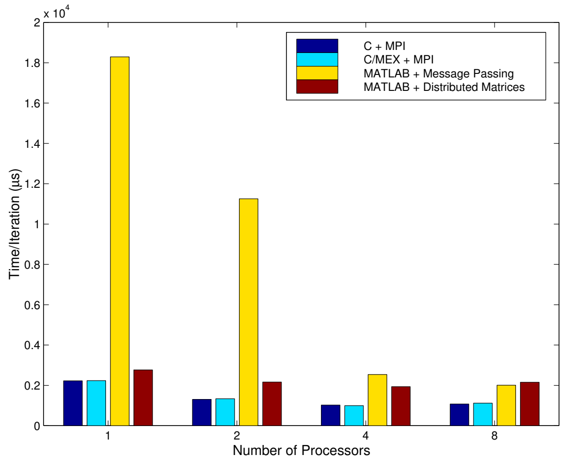

Figure 7: Parallel Conjugate Gradients on the IBM SP-2: 256 × 256 matrix

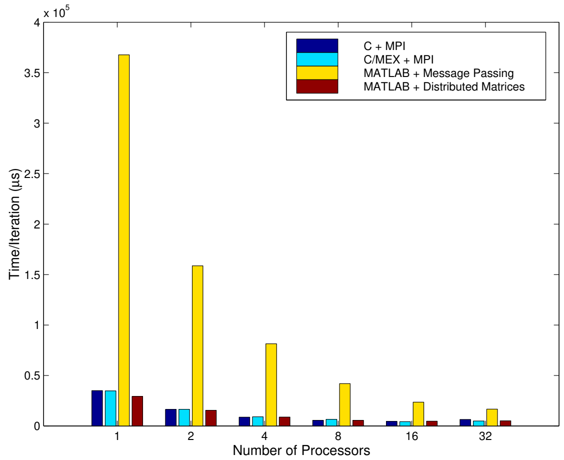

Figure 8: Parallel Conjugate Gradients on the IBM SP-2: 1024 × 1024 matrix

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.