Interactive Supercomputing with MITMatlab

Parry Husbands
Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square Room 218
Cambridge MA 02139 USA

Charles L. Isbell Jr.
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square Room 719
Cambridge MA 02139 USA

Alan Edelman
Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square Room 257
Cambridge MA 02139 USA

Abstract

This paper describes MITMatlab, a system that enables users of supercomputers to transparently work on large data sets within Matlab. MITMatlab communicates with an external server that is responsible for storing and operating on the data. Through the use of Matlab's object oriented features, we can handle this data as though it were "in" Matlab. For example, we can type [u,s,v] = svds(a,5) in Matlab and get results regardless of whether the matrix a has fifty or fifty million non-zero elements. We present the structure and details of our implementation along with some examples showing MITMatlab in action.

1 Introduction

This paper describes MITMatlab, a system that enables users of supercomputers to work in parallel transparently on large data sets within Matlab. MITMatlab is based on the Parallel Problems Server (PPServer)[8], a standalone linear algebra server that provides a mechanism for executing distributed memory algorithms on large data sets. This work is motivated by the desire to bring the many benefits of interactive environments to supercomputers while maintaining the efficiency and power of highly optimized computational libraries. Currently, when developing scientific applications on parallel machines, programmers use traditional languanges such as C and Fortran and either program in an explicitly parallel way or rely on their compiler to achieve good performance. Programming this way is typically difficult to debug and tune. This is in sharp contrast to the workstation world where, if the problem size is small, the application is quickly written in an interactive system such as Matlab.

MITMatlab provides a transparent way for Matlab users to interact with the PPServer. Through the use of Matlab classes and operator overloading, we can type `[u,s,v] = svds(a,5)` in MITMatlab and expect the singular triplets regardless of whether the sparse matrix `a` is a Matlab matrix with fifty non-zero elements or contains fifty million non-zero elements and is actually distributed among a number of machines working in tandem.

This document describes the structure and organization of our system as well as important details of our implementation. In Section 2, we discuss the Parallel Problems Server, the computational centerpiece of our system. Section 3 introduces the Matlab classes that are used. Section 4 contains sample MITMatlab sessions along with a discussion of the performance that we obtain. Our approach is then contrasted with other "parallel Matlabs" in Section 5. A glimpse of the future of MITMatlab s provided in Section 6 and we conclude with a discussion of the implications of our system in Section 7.

2 The Parallel Problems Server

The Parallel Problems Server forms the foundation of our work. It runs on any Unix-like platform supporting the MPI message passing library [4]. Simply, it is a compute server for large matrices. It contains functions for creating and removing distributed dense and sparse matrices, performing elementary matrix operations, and loading and storing matrices from/to disk using a portable format. Because matrices are created on the PPServer itself functions are also provided for transferring matrix sections to and from a client.

PPServer Matrices are two-dimensional and single precision. Dense matrices can be distributed by row or by column. Sparse matrices are distributed by column. Replicated dense matrices are also provided, though very few operations use them.

The PPServer communicates with clients using a simple request-response protocol. A client requests that an action be performed by issuing a command with the appropriate arguments, the server executes that command, and then notifies the client that the action is complete.

The PPServer is directly extensible via compiled libraries called *packages*. The PPServer implements a robust protocol for communicating with packages. Clients (and other packages) can load and remove packages on-the-fly, as well as execute commands within packages.

The PPServer provides a library of calls that enables package programmers access to direct information about the PPServer and its matrices. Programmers can thus write MPI code that operates directly on the PPServer matrices. Each package represents its own namespace, defining a set of functions and visible function names. This not only supports data encapulation, but also allows users to hide a subset of functions in one package by loading another that defines the same function names. Finally, packages support common parallel idioms (like applying a function to every element of a matrix), making it easier to add common functionality.

All but a few PPServer commands are implemented as packages, including basic matrix operations. Many highly-optimized public libraries have been realized as packages using appropriate wrapper functions. These packages include ScaLAPACK [2], S3L (Sun's optimized version of ScaLAPACK), PARPACK [9], and Petsc [5].

For a more complete description of the Parallel Problems Server, please see [8].

3 MITMatlab

By directly using the PPServer's client communication interface it is possible to access all of the PPServer's functionality from Matlab. Calls made directly to the PPServer from within Matlab are of the form:

`[error, errorstr,` out_1`, ...,` out_n`] = ppclient('cmd',` arg_1`, ...,` arg_n`);`

where `ppclient` is a small MEX function that implements the client-server communication protocol, arg_i is the list of arguments, out_i is the list or return arguments and `error` and `errorstr` contain information about any errors encountered.

We endeavor to make interaction with the PPServer as transparent as possible for the user. In principle, a typical Matlab user should never have to make a call to `ppclient` herself. Further, current Matlab programs should not have to be rewritten to take advantage of the PPServer. To these ends, we make use of Matlab 5's object-oriented features.

By defining Matlab *classes* that represent local stand-ins for objects that really exist on the PPServer, we can use Matlab's operator overloading features to ensure a transparent user experience. Although these new objects act like normal matrices to a user, they actually trigger operations on the PPServer instead of within Matlab.

3.1 New Classes

Matlab classes are defined for the dense and sparse PPServer matrix types, referred to as ddense and dsparse objects, respectively. The local objects contain the size and the name, or ID, of a particular PPServer matrix. It is the ID that is passed along to the PPServer.

3.1.1 Constructors

A *constructor* is a function that creates an object of some class or type. We have defined many constructors in Matlab that create the ddense and dsparse matrices that correspond to PPerver matrices. These are described in Table 1 below.

`a=dsparse('file')`	Load a sparse matrix from file
`a=dsparse(m,n)`	Create an empty mxn sparse matrix
`a=ddense('file'[,dist])`	Load a dense matrix with optional distribution `dist`
`a=ddense(m,n[,dist])`	Create an empty dense matrix
`a=drand(m,n,dist)`	Create a [0,1]-uniformly distributed dense matrix
`a=drandn(m,n,dist)`	Create a normally distributed random matrix
`a=dones(m,n,dist)`	Create a matrix full of 1s

Table 1: **Distributed Matrix Constructors.** There are several functions provided for creating PPServer objects. In Section 3.3 with describe transparent ways of invoking these constructors.

3.2 Operator Overloading

With the classes defined, we can overload common Matlab operations so that they work with the distributed objects. Overloading a function for a particular class requires only writing an m-file with the same name as the Matlab function to be overloaded that defines a function that takes and returns the same number of arguments. It is within this file that calls to `ppclient` are made in order to perform the appropriate computation. Table 2 lists functions we have overloaded for dsparse and ddense matrices.

ddense	`+,-,*,.*,./,\,inv,svds,svd,eig,hess,schur,` `qr,sum,cumsum,sort,exp,fft,imagesc,log`
dsparse	`*,svds,sum,nnz`

Table 2: **Overloaded Distributed Matrix Operations.**

Setting and retrieving array sections (using Matlab's syntax) also work transparently for ddense matrices. For dsparse matrices, individual elements can be retrieved and set.

3.3 p: Towards transparent constructors

One problem with the constructors described in Table 2 is that they do not directly correspond to Matlab's matrix constructors such as `zeros` or `rand`. Users must learn a new set of functions for creating distributed objects. To maintain transparency in the constructors, we introduce a new class: the *layout* object.

Layout objects behave exactly like integers, except that they enable us to overload Matlab's constructors (such as `rand`). If `rand(100,layout(100))` is entered, our overloaded constructor is called and a column distributed matrix is created in the server. If `rand(layout(100),100)` is entered, then the matrix is row distributed.

Although this seems cumbersome, it is possible to use the layout class using the global function `p`. Its value is `layout(1)` and so `100*p` is identical to `layout(100)`. Therefore `rand(100,100*p)` and `rand(100*p,100)` are identical to the examples above. In this way, `rand, randn, zeros, ones, eye, sprand`, and `sprandn` can all be called with `p`.

The use of `p` is not limited to just constructors. When matrix inquiry functions such as `size` return layout objects, these objects continue to propagate. Thus previously written Matlab functions that call constructors will automatically execute "in parallel" without modification.

For example, Figure 1 shows the code for Matlab's built in function `hilb`. The call `hilb(n)` produces the $n \times n$ Hilbert matrix ($h_{ij} = \frac{1}{i+j-1}$). When `n` is a layout object, a parallel array results:

- `J=1:n` in line 1 creates a PPServer object with $1, 2, \cdots, n$ and places it in `J`. Note that this behavior does not interfere with the semantics of for loops (`for i=1:n`) as Matlab assigns to `i` the value of each column of `1:n`: the numbers $1, 2, \ldots, n$.

```
1   function H=hilb(n)
2   J = 1:n;
3   J = J(ones(n,1),:);
4   I = J';
5   E = ones(n,n);
6   H = E./(I+J-1);
```

Figure 1: **Matlab code for producing Hilbert matrices.** When `n` is a layout object, each of the constructors creates a PPServer object instead of a Matlab object.

- `ones(n,1)` in line 2 produces a PPServer matrix.
- Emulation of Matlab's indexing functions results in the correct execution of line 3.
- In line 5, `E` is generated on the PPServer because of the overloading of `ones`.
- Finally `H` is also a PPServer matrix (line 6) because of proper overloading of elementary matrix operations.

We have also been able to execute much of Nicholas Higham's Matrix Test Toolbox [6] without any modification. Much of the work of this task involved supporting the multitude of Matlab's indexing capabilities. Successes include cauchy, circul, clement, cycol, dingdong, frank, kahan, lehmer, parter, pei, and triw. Some routines (such as hadamard and wilk) that return explicit Matlab matrices cannot be overloaded in this way. Others (kms, orthog, seqm, signm, and smoke) need support for complex numbers, an important extension not currently in the PPServer.

In examining the routines in the toolbox, it is clear that even with the capabilities of the PPServer it is not feasible to create very large instances of some of the matrices. For example, in double precision, `pascal(800)` contains `Inf`. Further, functions that make extensive use of element-wise operations (such as `pascal`), will not make full use of the parallelism of the PPServer.

4 MITMatlab in Action

MITMatlab currently runs on clusters of Symmetric Multiprocessors from Sun Microsystems and Digital Equipment Corporation residing at MIT's Laboratory for Computer Science, as well as clusters of Intel PCs at MIT's Artifical Intelligence Laboratory. In this section, we show screen dumps of MITMatlab to demonstrate its functionality. Most of these examples used four processors of an eight processor, 512 MB Sun Ultra Enterprise 5000 Server.

Sparse Linear Algebra The major sparse matrix operation that we provide is the sparse singular value decomposition, `svds` from PARPACK. This routine and others are shown in Figure 2.

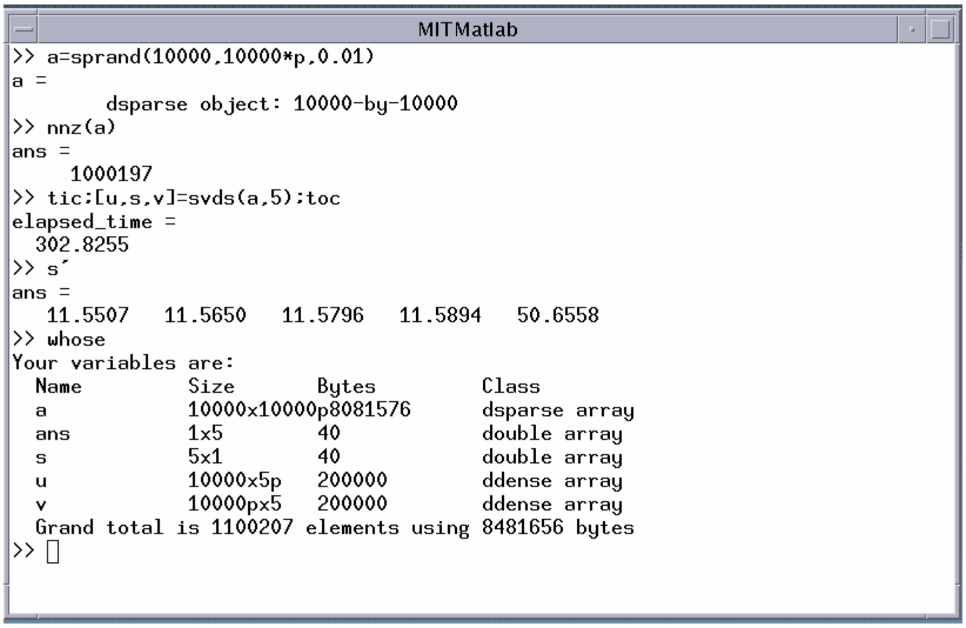

Figure 2: **MITMatlab Sparse Functionality.**

Dense Linear Algebra Most of the dense matrix functionality comes from ScaLAPACK. A few functions, most notably `inv`, use S3L. These routines are demonstrated in Figure 3.

Miscellaneous Matlab contains a host of utility routines that make programming easier. We have tried to incorporate some of the most common ones in the PPServer. Figure 4 shows some of these in action.

4.1 Performance

While the efficiency of PPServer naturally depends on the algorithms used for the operations, there are really two main factors that define the performance of the MITMatlab system. First, Matlab has to communicate with the PPServer: a message with a function to be called and its arguments must be sent, and the return message must be translated into Matlab 5. In our experiments, this incurs a round trip time of approximately 2 milliseconds. This is insignificant for most operations on large matrices (such as singular value decompositions and matrix multiplications) but greatly affects the retrieving and setting of small pieces of matrices.

Second, matrices may have to be re-distributed prior to computations. For example, if a row distributed matrix is added to a column distributed matrix, elements have to be sent to the correct processors so that the addition can be local.

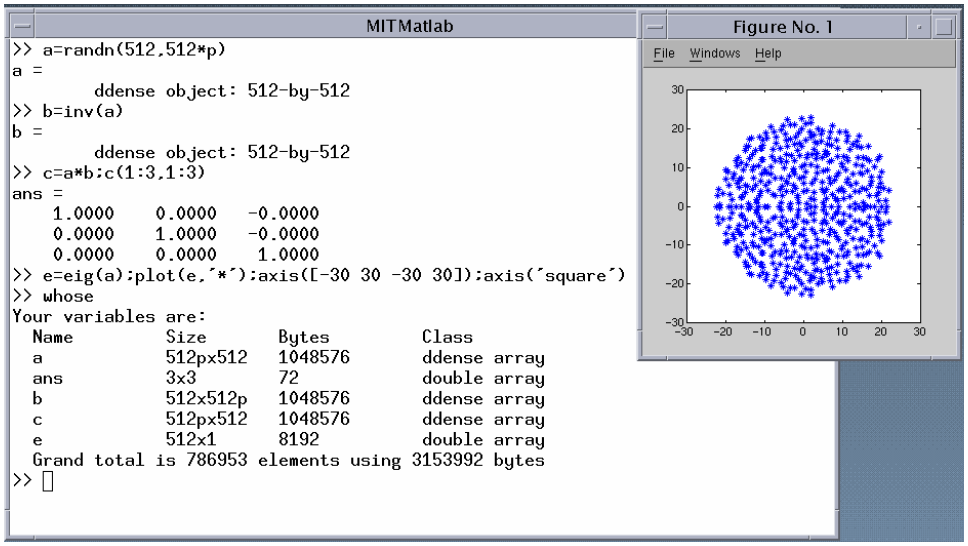

Figure 3: **Examples of MITMatlab Dense Matrix Functionality**

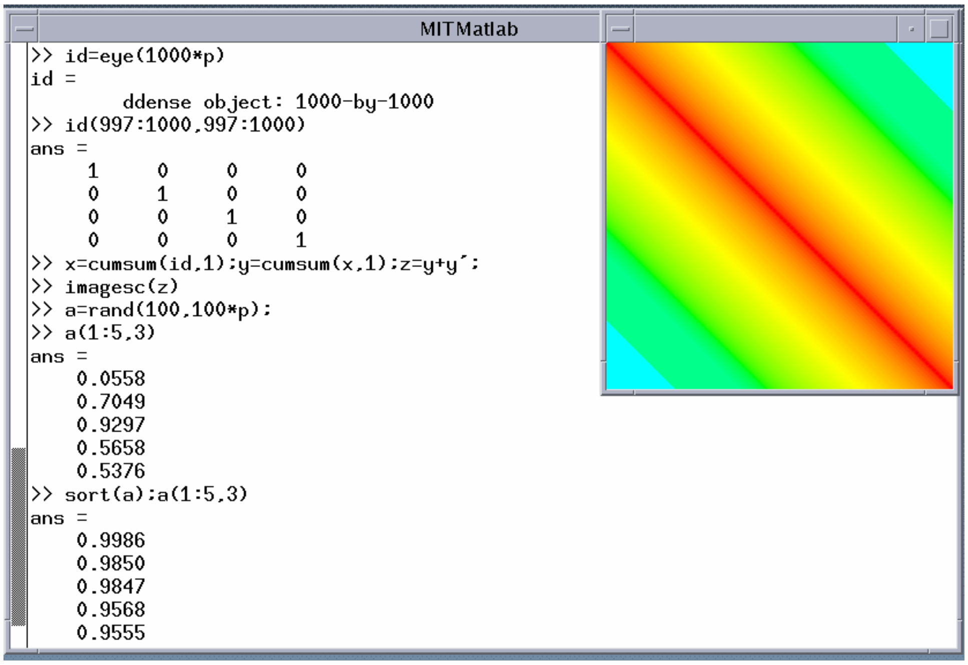

Figure 4: **Miscellaneous MITMatlab functions.**

5 Related Work

Both RCS [1] and NetSolve [3] provide facilities that enable interactive clients to access remote compute servers. The clients send over the function name and the data to be worked on, the server computes, and the results are sent back to the client. We believe that our system is different in two important respects. First, our client (Matlab) is not responsible for storing the data. In an environment such as a cluster of SMPs, this is an advantage: we can easily work with data sets that are much larger that the memory of a single machine. Such data sets are distributed in the PPServer across the machines of the cluster. Second, we provide transparent access from Matlab to the server's operations. NetSolve, for example, requires that users explicitly call the server to execute operations.

Other systems have been proposed that add message passing features to Matlab. MultiMatlab [10] and the Parallel Toolbox for Matlab [7] make it possible to manage a group of Matlab processes on a supercomputer. Here Matlab is extended to include send. receive, and some global operations. However, applications must still be developed using a message-passing style and computation makes use of Matlab's provided algorithms.

Other "parallel Matlab" systems include the DP-Toolbox, Paramat, and Matpar.

6 Further Work

It is our hope that MITMatlab can be a useful environment for a wide variety of users. To this end, we are porting as many packages as we can so as to provide as much functionality as possible. Other improvements, however, will need changes to Matlab and the server.

6.1 Garbage Collection

By far, the most difficult feature to implement with MITMatlab is automatic *garbage collection.* Garbage collection is the process of reclaiming memory space that is no longer accessible by a program. Although Languages such as Lisp and Java have sophisticated garbage collection mechanisms, even languages such as C++ and Fortran do garbage collection of a kind when local variable space is freed after it has gone out of lexical scope.

While Matlab 5 provides support for object-oriented programming, it does not yet implement true user-defined *destructors*. As a result, there is no way for an object to be notified automatically when it is about to be deleted, either explicting via `clear` or implicitly by going out of scope. To make matters worse, there is no way to obtain a list of all currently defined variables within a function. These combine to make it impossible implement an automatic garbage collector.

In the worst case, PPMatlab users are thus burdened with having to explicitly clear variables that are not wanted (using the `ppclear` function). Complicating things even further is the need to break up complex expressions like `e=a+b+c+d` to avoid creating garbage when subexpressions (such as `a+b`) are evaluated within Matlab, creating "temporary" objects.

To reduce some of this programming effort, we have implemented a semi-automatic mechanism for garbage collection. We provide a function called `ppscope` that returns a time stamp. When ppgc is called with a time stamp created by ppscope, all variables created since that time that are not in ppgc's argument list are deleted. For example,

```
SCOPE=ppscope;
g=(a+b+c+d)*e + f;
ppgc(SCOPE,g);
```

deletes all of the temporary variables that were created in the complex assignment to g.

6.2 PPServer Improvements

Currently the PPServer supports single precision, two-dimensional matrices distributed either by column or row. Sparse matrices are further restricted to be column distributed. While these choices have simplified initial implementation and served the purposes of our group effectively, it seems to wise to expand our choices.

Generalized block cyclic distributions (à la ScaLAPACK) often lead to better algorithm performance and so are planned for future versions of the PPServer. Matlab 5 implements higher rank objects (n-dimensional matrices) and we hope provide the same. Support for double precision and complex numbers is also planned.

Finally, we intend to provide a mechanism for adding user-defined data types to the PPServer as a way of supplementing our ability to add user code via the package system. For example, it would useful to allow packages to define symmetric and banded matrix types, or "matrix-free" linear operators.

7 Conclusion

We have argued that MITMatlab enables portable, high-performance interactive supercomputing through the use of the Parallel Problems Server and the new Matlab 5 classes. The PPServer provides a powerful, uniform mechanism for writing and accessing optimized algorithms, and via it client communication protocal makes it possible to implement transparent integration with sufficiently powerful clients, such as Matlab 5. Further, the overhead of using the PPServer as a backend is insignificant for the operations for which it was designed; namely, operations on large matrices.

With such a tool, researchers and students can now use Matlab as something more than just a way for prototyping algorithms and working on small problems. MITMatlab makes it possible to interactively operate on and visualise large data sets.

This style of computing represents a radical departure from traditional supercomputing where users submit jobs to batch queues and their results get saved for later analysis. While MITMatlab can certainly operate in such an environment (through the use of Matlab scripts) much of its power comes from its interactivity. We therefore hope to explore how tools such as MITMatlab can operate effectively in batch installations.

Acknowledgements

Parry Husbands is supported by a fellowship from Sun Microsystems. Charles Isbell is supported by a fellowship from AT&T Labs/Research. Most of this research was performed on clusters of SMPs provided by Sun Microsystems and Digital Corp.

References

[1] P. Arbenz, W. Gander, and M. Oettli. The Remote Computation System. Technical Report 245, ETH Zurich, 1996.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhilon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. ScaLAPACK Users' Guide. http://www.netlib.org/scalapack/slug/scalapack_slug.html, May 1997.

[3] Henri Casanova and Jack Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. In *Proceedings of SuperComputing 1996*, 1996.

[4] William Gropp, Ewing Lusk, and Anthong Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, 1994.

[5] PETSc Group. PETSc - The Portable, Extensible Toolkit for Scientific Computation. http://www.mcs.anl.gov/home/gropp/petsc.html.

[6] Nicholas Higham. The Test Matrix Toolbox, 3.0. http://www.ma.man.ac.uk/~higham/testmat.html.

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

[8] Parry Husbands and Charles Isbell. The Parallel Problems Server: A Client-Server Model for Interactive Large Scale Scientific Computation. In *Proceedings of VECPAR98*, June 1998.

[9] K. J. Maschhoff and D. C. Sorensen. A Portable Implementation of ARPACK for Distributed Memory Parallel Computers. In *Preliminary Proceedings of the Copper Mountain Conference on Iterative Methods*, 1996.

[10] Anne E. Trefethen, Vijay S. Menon, Chi-Chao Chang, Gregorz J. Czajkowski, Chris Myers, and Lloyd N. Trefethen. MultiMATLAB: MATLAB on Multiple Processors. http://www.cs.cornell.edu/Info/People/lnt/multimatlab.html, 1996.