Interactive Supercomputing

by

Parry Jones Reginald Husbands

B.Sc., University of Toronto (1992)
S.M., Massachusetts Institute of Technology (1994)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1999

Author .... [illegible] ....
Department of Electrical Engineering and Computer Science
January 29, 1999

Certified by .... [illegible] ....
Alan Edelman
Associate Professor of Applied Mathematics
Thesis Supervisor

Accepted by .... [illegible] ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Interactive Supercomputing

by

Parry Jones Reginald Husbands

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

MITMatlab is a system that provides users of high performance computers with an interactive, easy-to-use environment for solving their scientific and engineering problems. It is an effort to bridge the gap between scientific computing in the desktop and supercomputer worlds by providing users of parallel machines with a tool for manipulating and visualising large datasets. We believe that the many benefits of interactive tools (such as MATLAB) can be enjoyed in supercomputer installations without any appreciable loss in performance.

A key concept in our work is the development of a methodology that we denote "Parallelism through Polymorphism". This is an alternative way of obtaining parallel execution of existing code that falls between automatic parallelisation and complete rewriting. In contrast to traditional compiler-based approaches such as preprocessor directives and automatic parallelisation, our technique has the advantage of not requiring any internal changes to MATLAB while delivering parallelism to both interactive sessions and programs. Careful implementation choices provide us with a nearly transparent, seamless interface.

Our implementation is based on a simple client-server model. A server, the Parallel Problems Server, is responsible for manipulating application data and is controlled using a seamless MATLAB user interface. This thesis primarily details the design choices that lead to the creation of MITMatlab. In addition, we present its performance and describe some of its uses.

Thesis Supervisor: Alan Edelman
Title: Associate Professor of Applied Mathematics

Acknowledgments

I would like to thank my advisor Alan Edelman for his guidance and support over the last four and a half years. I am particularly grateful for all of the time and energy he devoted to this project. Many thanks to my thesis committee, David Karger, Charles Leiserson, and Paul Viola for their insightful comments and suggestions. Graduate counsellors Albert Meyer, Randy Davis, and Silvio Micali also helped me immensely as I attempted to navigate my way through the graduate program in Computer Science.

Thanks also to my fellow group members Ross Lippert, Peter McCorquodale, Yanyuan Ma, and Tony Wen for their support. Several people contributed to the development and implementation of MITMatlab including Susumo Kubo, Misha Chechelnitsky, and Charles Isbell. Charles deserves special mention for his many contributions to the project as collaborator, thesis reader, and friend. Sara Billey also gave valuable comments on an earlier draft of this thesis. William Ang, Scott Blomquist, and Fred Donovan were extremely helpful in making sure that the computing resources were up and running and accommodating our peculiar requests. Sun Microsystems provided funding during the last two years of the program and I am grateful to Don Dudley, Brian Hammond, Greg Papadopoulos, and Georgi Johnson for their interest in the project.

I am fortunate to have met a number of people during my time at the Lab. for Computer Science who helped me grow as a researcher, teacher, and student. These include Jakov Kučan, James Hoe, Mark Smith, Chris Hill, Mike Sipser, Mauricio Karchmer, Ron Rivest, and Be Blackburn (a.k.a. Mom) who seldom ran out of chocolate. The volleyball teams I've been part of provided a welcome relief from the stresses of life at the lab. I owe a great deal to the Vile Servers, Slimers, IVC, GVC, and `chickvb@mit.edu`.

Finally, many thanks to my parents Parry and Genetha Husbands and my sister Mary-Gene for their unfailing love, support, and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis details the design and implementation of a system that provides users of high performance computers with an interactive, easy-to-use environment for solving their scientific and engineering problems. In the desktop world, programs such as MATLAB, Maple, and Mathematica have been extremely successful, primarily by offering low development and maintenance costs through intuitive abstractions for expressing ideas and powerful built-in visualisation for understanding complex systems. With these products, scientific applications can be easily designed, debugged, and maintained with much less programming effort than with a traditional programming language such as FORTRAN. The main disadvantage of using these tools, of course, is speed. They were not written for use with very large datasets and do not take full advantage of today's high performance computers. As a consequence of this, interactive tools are typically used only for prototyping applications or small datasets.

In the supercomputer world, most programmers write their applications using subroutine libraries[1]. These libraries provide the speed and functionality required, but none of the user-friendliness, ease of use, and interactivity found in a package such as MATLAB.

Our work is an effort to bridge the gap between scientific computing in the desktop and supercomputer worlds by providing users of parallel machines with an interactive

[1]Readers unfamiliar with parallel programming are invited to read the short summary provided in Appendix A.

environment where they can manipulate and visualise large datasets. It is our contention that the benefits of interactive environments can be enjoyed in supercomputer installations without any appreciable loss in performance.

Using the technique of "Parallelism Through Polymorphism" developed in Chapter 2, we add fast parallel execution to MATLAB, a common interactive tool. In contrast to traditional compiler-based approaches such as preprocessor directives and automatic parallelisation, our method has the advantage of not requiring any internal changes to MATLAB while delivering parallelism to both interactive sessions and programs.

1.1 Design Philosophy

We believe that for an interactive supercomputing tool to be successful, it should provide users with:

- The advantages of interactive tools: Mechanisms for fast prototyping and implementation of different techniques for solving scientific problems in an interactive setting. Easy visualisation of results.
- Support for very large dense and sparse matrices.
- Complex operations on these matrices such as matrix multiplication and singular value decomposition.
- Implementation on a wide variety of contemporary parallel platforms, including clusters of Symmetric Multiprocessors (SMPs).

There are, of course, many different ways of implementing a system satisfying these requirements. We therefore formulated a set of principles that guide the development of our system:

Leverage Existing Technologies

There are many reasons for basing our system on "off the shelf" components. High quality parallel libraries for both dense and sparse linear algebra abound and it would be fruitless (and time consuming) to attempt to re-implement their functionality. In addition the use of these libraries leads to portability across a wide range of platforms.

It is also important to present a familiar interface to users so that they are not burdened with the task of learning a new programming language. MATLAB is already very popular in science and engineering (for example, only 4 of 30 students in a graduate Scientific Computing class at MIT confessed to never using it) and so we decided that it should be used for our user interface.

These considerations lead naturally to the choice of a client-server architecture for our system where we use the best tools possible both for numerical routines and the user interface. The alternative, a monolithic system, would either involve adding a user interface to parallel libraries or rewriting an interactive tool to work on a wide range of parallel computers. These options have serious drawbacks: the first would not present a familiar interface and the second would be tied to a particular version of specific tool.

We use a common interactive tool (MATLAB, but with support for others) as our client and user interface and a server that encapsulates the functionality of the libraries. The use of an existing environment adds many constraints to our effort. For example, we are not free to make major changes to the MATLAB programming language. However we believe that the many benefits of using MATLAB (its familiar user interface and visualisation features) far outweigh the difficulties that we may encounter while implementing our system (see Chapter 4 for a further discussion).

Minimise Client-Server Communication

This principle follows directly from the choice of a client-server architecture and the desire to achieve high performance. Sending large quantities of data to the server (or client) should be avoided whenever possible so that we don't pay the performance penalty. As a consequence of this, we decided to keep all of the data on the server and only send small pieces of it to the client when explicitly requested.

Expose as Many Details of the System as Possible

In any system new functionality is always required. By providing an API (Application Programming Interface) to the inner workings of our system, we have a mechanism for experts to add new routines to the client and server whenever needed. This also provides a way for the optimisation of existing routines if special purpose functions are desired.

Present Client Functionality in an Intuitive Way

All of the familiarity of using an existing client would be lost if separate function calls were needed to access server-side routines. As such, we attempt to provide this access in as transparent a way as possible. This also lead to the choice of MATLAB as a client. With its object oriented features we can implement our "Parallelism through Polymorphism" technique and have MATLAB commands work on server objects in the same way as MATLAB's native data types. It is through this interface that our system derives much of its power; existing MATLAB code can be re-used with little or no modification thus preserving investments made both in code and learning time.

1.2 System Overview

Our system is based on a simple client-server model. We first built a server that is responsible for manipulating application data. We then constructed a user interface using MATLAB, a popular desktop scientific computing tool.

1.2.1 The Parallel Problems Server

The Parallel Problems Server (PPServer) [23] forms the computing foundation of our work. It runs on any Unix-like platform supporting the MPI message passing library [19]. Simply, it is a (single-user) compute server for large matrices. It contains functions for creating and removing dense and sparse matrices, performing elementary matrix operations, and loading and storing matrices from/to disk using a portable format. Because matrices are created within the PPServer's address space, functions are also provided for transferring matrix sections to and from general clients, such as MATLAB.

Currently PPServer matrices are (at most) two-dimensional. At compile-time users have a choice of using either single or double precision numbers for server matrices. In our implementation, dense matrices can be distributed across the MPI processes by row or by column and sparse matrices are distributed by column. Support for more flexible distributions is planned as a future enhancement. Replicated dense matrices (copies of which are stored on every processor) are also provided, though very few operations use them.

The PPServer communicates with clients using a simple request-response protocol (see Figure 1-1). 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. At present no security measures are implemented on the client communication channel. In keeping with our principle of minimising client-server communication most results of PPServer operations are stored on the server. Large matrices are only transferred to MATLAB when explicitly requested by the user.

The PPServer is extensible via compiled libraries called *packages.* These *packages* are written in C++ and access the server's data (and algorithms) using a provided API. Clients (and other packages) can load and remove packages on-the-fly, as well as execute commands within these packages. It is worth noting (see Figure 1-1) that non-trivial operations do not always have to be implemented with packages. Client-side scripts (such as MATLAB "m-files") can often be employed for this purpose, using existing server operations as building blocks for more complex functions.

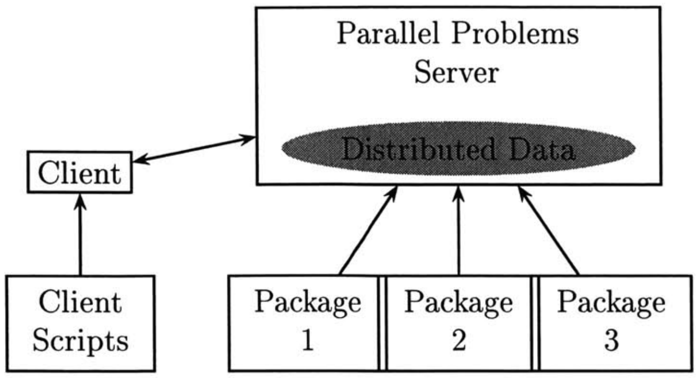


Figure 1-1: Organisation of the Parallel Problems Server with client

1.2.2 MITMatlab

MITMatlab provides MATLAB with a transparent way of accessing server data and functions. It consists of a collection of MATLAB 5 classes and methods that make it possible for users to interact with server data by using traditional MATLAB commands. Figure 1-2 shows some sample input to an ordinary (serial) MATLAB session. Figure 1-3 shows how the same commands can be executed in parallel in MITMatlab.

The only differences between the two sessions are the inclusion of "`p`" on lines 2 and 4 and the use of "`whose`" on line 6. The "p" serves as our "hook" into the PPServer. Its presence in any MATLAB matrix constructor[2] (such as `randn`) signals the creation of a PPServer matrix. This matrix is then operated on by methods such as "`inv`" (line 13) that emulate the corresponding MATLAB functions. The "`whose`" function is similar to MATLAB's "`whos`", but gives detailed information about PPServer matrices.

In addition to the emulation of MATLAB functionality we have designed the classes and methods so that ordinary MATLAB code can be executed with little modification (in some cases no modification is needed). In fact, in order to get "up to speed" with our system, MATLAB users only need to know how to use `p`, how

[2]A *constructor* is a function that creates an object of a specified type.

```
>> % Create a random (normally distributed) 512x512 matrix
>> a=randn(512,512);
>> % Create a 512x512 matrix full of ones
>> a2=ones(512,512);
>> % Display current variables
>> whos
Your variables are:
 Name     Size         Bytes          Class
 a    512 x 512      2097152         double array
 a2   512 x 512      2097152         double array
Grand total is 524288 elements using 4194304 bytes
>> % Find the inverse of a
>> b=inv(a);
>> % Multiply a by b
>> c=a*b;
>> % View the upper left part of c
>> c(1:3,1:3)
ans =
  1 0 0
  0 1 0
  0 0 1
>>
```

Figure 1-2: Sample MATLAB session

```
>> % Create a random (normally distributed) 512x512 server matrix
>> a=randn(512,512*p);
>> % Create a 512x512 matrix full of ones
>> a2=ones(512*p,512);
>> % Display current variables
>> whose
Your variables are:
 Name     Size        Bytes          Class
 a    512 x 512p    1048576          ddense array
 a2   512px 512     1048576          ddense array
Grand total is 524288 elements using 2097152 bytes
>> % Find the inverse of a in parallel using the PPServer
>> b=inv(a);
>> % Multiply a by b
>> c=a*b;
>> % View the upper left part of c
>> c(1:3,1:3)
ans =
  1 0 0
  0 1 0
  0 0 1
>>
```

Figure 1-3: MITMatlab input for the operations in Figure 1-2

to load and save PPServer matrices (which currently cannot be overloaded), and an appreciation of the issue of garbage collection. Chapter 4 further discusses the design choices that make this all possible.

1.3 Related Work

Most approaches to Interactive Supercomputing have also focused on using MATLAB. In this section, we summarise these attempts and compare them with our PPServer approach.

1.3.1 Parallel MATLAB Systems

Both MultiMATLAB from Cornell University [43] and the Parallel Toolbox for MATLAB from Wake Forest University [21] make it possible to run and manage MATLAB processes on different machines. In these systems MATLAB is extended to include send, receive and global operations so that the MATLAB processes can communicate to solve a computational task. This is much like MPI with MATLAB as the implementation language.

Our approach to "parallel MATLAB" is different in many respects. Instead of endowing MATLAB with communication primitives, we make the communication implicit. The user is not responsible for moving the data around in our system. He simply inputs standard MATLAB commands and they execute on a parallel machine. Our approach can be viewed as bringing data parallel computing to MATLAB where other efforts have focused on message passing. The main advantage of our path is that complex operations on large matrices can be specified with only a single command instead of code that explicitly moves data around.

Secondly, we do not use MATLAB for our computational engine. While functions that are not implemented on the server can be emulated by transferring the data to MATLAB and then executing the function there, this would incur a substantial performance penalty. It would therefore be best to re-implement these functions. This has the short-term disadvantage of limited functionality, but we eventually reap the advantage of high performance. We are free to use the fastest available distributed memory implementations of the algorithms that we need. We are also not limited to double precision (MATLAB's default). Sometimes single precision computations suffice, particularly when space is at a premium.

1.3.2 Client-Server Systems

There are also systems that implement a similar client-server model with MATLAB as the front-end. RCS [2], Netsolve [12], PSI [34], and MatPar act as fast back-ends for slower clients. In their model, clients issue requests with special function calls, data is communicated to the remote machine and results sent back. Clients have been developed for Netsolve using both MATLAB and Java. MATLAB and Mathematica clients exist for PSI and it uses PLAPACK [1] for computation.

Our clients, however, are not responsible for storing the data to be computed on. Generally, data is created and stored on the server itself; clients receive only a "handle" to this data. This means that there is no cost for sending and receiving large datasets to and from the computational server. Further, this approach allows computation on data sets too large for the client itself to even store (for example, when working on a workstation cluster).

We also support transparent access to server data from clients. PPServer variables can be created remotely but still be treated like local variables. It is worth noting that our approach to the client-server model very clearly separates the large-scale computation from the user interface. By doing so we can use the best possible interface for the task at hand without considering its computational (or storage) facilities. We can also use the fastest possible means to solve the computational problem. This delegation of responsibilities thus results in extremely powerful, yet easy to use applications.

1.3.3 Compilers

Compilers for MATLAB and similar languages have also been an active area of research. The CONLAB system from the University of Umeå [14] is a parallel MATLAB-like simulator. CONLAB scripts may be compiled into C for parallel execution. The FALCON environment from the University of Illinois at Urbana-Champaign [39, 40] compiles MATLAB to Fortran 90 and pC++ (a parallel dialect of C++). Otter [38, 37] converts MATLAB scripts into C code which calls a special run-time library (based on ScaLAPACK [6]) that provides dense linear algebra functionality. These conversions are accompanied by sophisticated analyses of the MATLAB source so that the most efficient target code can be generated.

By contrast our system takes the view that we can obtain high performance by simply providing users with the fastest available implementations of basic operations in an interpreted environment. Because we work in an interactive setting, we can reduce development time without severely affecting performance.

1.4 Thesis Roadmap

Chapter 2 introduces the concept of "Parallelism through Polymorphism". Chapters 3 and 4 show how it is implemented in our system with the PPServer and MITMatlab. The performance of our implementation is then discussed in Chapter 5. In this chapter we also attempt to characterise applications that are well-suited to our model and implementation. Chapter 6 describes some applications of our system, including IRLAB, an interactive environment for Information Retrieval research developed with MITMatlab. Finally, Chapter 7 points to some further directions for our work and concludes with a discussion of the applicability of our system for scientific research.

Chapter 2

Parallelism Through Polymorphism

One of the main contributions of this work is the novel use of polymorphism to deliver parallel execution of existing MATLAB code.

2.1 Influencing Properties of Program Execution

Currently there are a few options available to implementors who wish to add parallel execution to routines in an existing programming language without changing the text of the routines or the programming language itself:

- Use Compiler Directives embedded in comments. In this style, a compiler is built that understands these "smart comments" and generates the appropriate code. For example, in a "C"-like language the following comment could be used to denote a parallel `for` loop:

  ```
  /* $(C)$ PARALLEL FOR */
  ```

- Write a new compiler for the language that automatically parallelises code. These tools are currently the "holy grail" of supercomputing research and attempt to discover from the program text opportunities for parallelism, mainly in `for` loops. While this technology is very sophisticated it is often not sufficient to fully parallelise applications as some loops to not fall into the set of recognised "patterns".

- Translate the routine into another programming language that is implicitly parallel. An example of this is the FALCON environment. MATLAB source is translated to Fortran 90/pC++ where parallelism is an integral part of the programming model.

If all else fails, the program may have to be rewritten to support the property. However, if the language supports polymorphism, the ability of functions to take arguments of different types, there is another way:

- Define new classes and overload operators in the language to provide for parallel execution of these operators. In this case, to run routines in parallel, they are called with arguments of the appropriate class.

Note that these techniques are all solutions to the general problem of influencing aspects of program execution while preserving the syntax and semantics of the underlying programming language. Parallel execution can be replaced with other run-time "properties" such as "has efficient garbage collection", "uses a certain algorithm for operation x", "uses sparse instead of dense matrices", or "takes advantage of specialised hardware".

In terms of programmer effort (measured by changes to the original routine [1]), the techniques can be ordered according to Figure 2-1.

2.2 Advantages of Polymorphism

In addition to not requiring code changes, the use of polymorphism enjoys several other advantages:

[1]Depending on the programming language, the use of polymorphism may necessitate a change in the way the routine is *called*.

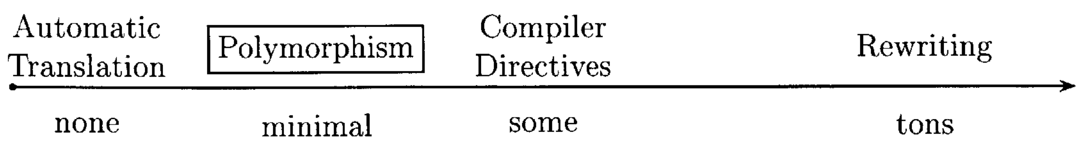


Figure 2-1: The programming cost of adding parallelism

- Support for interpreted/interactive environments. Compiler directives are useless in an interactive environment (such as MATLAB) as there is no compiler. For example, explicit statements must be added to users' sessions to perform the required tasks at specified times. Automatic tools (translation, etc.) somehow have to be integrated into the environment. However, this would entail rewriting of the tool or environment. This has the disadvantage of tying the implementation to a specific version of the tool.

- Modifiability/Extensibility, a.k.a Plug 'n Play. This refers to the ability of "educated users" to tailor or extend the technique to their needs. For example, if a new parallel method for a certain task is invented, a compiler would need to be updated. Extensive modifications may be necessary depending in its implementation. By contrast, in our approach, only the classes (and methods) need to be changed. In addition it can be performed using the programming language itself and requires no source code for the compiler/runtime system.

Figure 2-2 summarises all of these issues.

	Automatic	Polymorphism	Directives	Rewriting
Programmer Effort	none	minimal	some	tons
Interactive?	no	yes	no	yes
Plug 'n Play Cost	high	low	high	low

Figure 2-2: Other costs of adding parallelism

2.2.1 The Importance of Libraries

Libraries play a crucial role in all of the techniques described above. Compilers (Otter, for example) may translate language constructs into library calls, and the overloaded operators (as in our system) may use libraries to deliver functionality. In some cases it may also be possible to simply recompile the routine using the library. For example, if the routine uses MPI calls, a simple recompilation is needed when moving from a shared memory to a distributed memory machine. However, if language keywords are used (`+,*,-` in MATLAB, for example), libraries must be used in conjunction with operator overloading.

2.3 The Polymorphism Recipe

The formula for using polymorphism is outlined below:

1. Define classes and overload the operations for the types of interest. In the MATLAB case, we needed new dense and sparse sub-classes. Matrix operations were overloaded to provide for parallel execution.

2. Analyse other types that influence the construction/behaviour of the types in 1) and overload/extend appropriately. Ordinary MATLAB `doubles` are passed into functions that create MATLAB matrices. Therefore in order to complete the task (making MATLAB routines execute in parallel on the PPServer), a way of making these constructors work in parallel is required, ergo `p`.

This process may, of course, be iterative. As with automatic parallelisation, the goal is to add the property to as much code as possible. Chapter 4 discusses the design and implementation of the class system that we devised to enable MATLAB code to execute in parallel.

2.4 Language Considerations

The success of the procedure outlined above depends, of course, on the language and property under consideration. MATLAB, for example, supports (and encourages) the data-parallel style of programming where complex operations are provided using simple keywords and operators (`A+B` adds two matrices, for instance). In a language such as C, such operations would have to be performed using either loops or function calls. We can therefore deliver parallel execution to much more code in MATLAB (by overloading `+,*,-` etc.) than in a language such as C.

Another key point is that we provide parallelism directly through operators and do not infer it from other language constructs (loops, for example). Further, we are not able to perform optimisations based on program text. This has the disadvantage of not being able to extract all of the parallelism available in routines. However, the use of the data-parallel style in MATLAB makes it possible to execute many complex operations on large matrices, which is sufficient for an important class of problems (as seen in Chapter 6). In addition, we can provide parallel versions of operators in an interactive environment.

Finally, the overhead involved in using overloaded methods also has to be taken into account. This has direct bearing on the performance of the modified system. In the context of our system, its impact on performance is discussed in Chapter 5.

Chapter 3

The Parallel Problems Server

The Parallel Problems Server (PPServer) provides an easy way for clients to perform large scale linear algebra computations on parallel machines. Server functionality is delivered through routines that assume a distributed memory model of parallelism and use the MPI message passing library for communication. Clients then call these routines without having to take care of parallel programming details.

The choice of MPI as our parallel environment follows our principle of leveraging existing technologies and has two key advantages:

Richness of available software Many scientific software libraries are written for MPI and can be incorporated into the server. These include ScaLAPACK [6], PLAPACK [1], and S3L for dense linear algebra, PETSc [3] for the solution of equations related to partial differential equations, KeLP [15] for domain decomposition methods, and PARPACK [31] for Arnoldi methods for solving linear equations and eigenproblems. In fact, most parallel scientific software today is written for MPI.

Portability MPI runs on a wide variety of platforms, from clusters of workstations to high end multiprocessors. Shared-memory multithreading packages such as PThreads [35] were considered as alternatives. However, these packages are only supported on a limited range of architectures (not, for example, on many clusters of SMPs) and their use would adversely affect the portability of our system.

In addition to actually operating on data, the PPServer has three main responsibilities:

1. Matrix Management

2. Client Communication

3. Function Management and Dispatch: The Package System

3.1 Matrix Management

The PPServer operates on two main types of matrices – dense and sparse. Sparse matrices have a very small number of non-zero elements and space can often be saved by only storing these elements (in addition to their row and column indices). Users decide on the type of matrix that is best for their data, and functions are provided for conversion between the two. Because the server runs in a distributed memory environment (MPI), the matrix data has to be spread across the processes that form the server program. For dense matrices, the server supports three types of distribution: by row, by column, and replicated. In our current system, sparse matrices are only distributed by column (with more distributions planned). See Figure 3-1 for examples of the row and column distributions used in the PPServer.

The local pieces of dense server matrices (residing on a single processor) are always stored in column major (FORTRAN) order as opposed to row major (C) order. In column major order, each column of the matrix is contiguous in memory while in row major order each row is contiguous. Figure 3-2 shows an example of the two orders.

Each PPServer matrix has an associated identifier (a natural number). These identifiers are allocated at matrix creation time and are reclaimed when the matrix is deleted.

At compile time, a user can decide between single or double precision storage in the PPServer. For some applications, the full accuracy of double precision may not

Column distributed			
Processor 0		Processor 1	
(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

Row distributed				
Processor 0	(1,1)	(1,2)	(1,3)	(1,4)
	(2,1)	(2,2)	(2,3)	(2,4)
Processor 1	(3,1)	(3,2)	(3,3)	(3,4)
	(4,1)	(4,2)	(4,3)	(4,4)

Figure 3-1: Different ways of distributing a 4 by 4 matrix between two processors

Location 1	a(1,1)	a(1,1)
2	a(2,1)	a(1,2)
3	a(3,1)	a(2,1)
4	a(1,2)	a(2,2)
5	a(2,2)	a(3,1)
6	a(3,2)	a(3,2)

Figure 3-2: Column and row major orders for a 3 by 2 matrix

be required and so the space requirements of the server can be cut in half by using single precision.

PPServer matrices can also be stored to and read from disk in either precision. To ensure portability of the saved matrices across different machines we have to address the issue of byte order. Floating point numbers occupy many bytes (4 for single precision, for example) and these bytes can be read either from "left to right" or from "right to left". Therefore, if a matrix is saved with its floating point numbers in one order, these numbers have to be "reversed" when loaded on a machine assuming a different order.

PPServer are always stored in "little endian" format (Sun's native order) and automatic conversions are performed when running on "big endian" (e.g. DEC) machines.

3.2 Client Communication

Clients communicate with the PPServer using a pair of Unix sockets that are typically opened and initialised at program startup. The client can either reside on the same machine as the server or on any networked host. The server implements a simple request/response protocol with clients (independent of the MPI layer that the server uses internally). Client requests consist of a command followed by the required arguments. Server responses first contain an error code and error string (in the event of a problem) and then the particular return arguments of the command.

Figure 3-3 gives an example of an exchange between a client and the server. The client would like the value of element (10,15) from the matrix with id 1. The server function for this is "`pp_view_element`". In the figure, the client packs up the request and sends it over through the upstream socket. The server replies through the downstream socket with an error code, an error string, and the requested value. Note that each datatype (matrix id, integer, double, vector of integers) that can be communicated has an associated tag.

Client Request:

15	Command string length
"pp_view_element"	Command string
3	Number of arguments
PPMATRIXID	First argument type
1	First argument value
DOUBLE	Second argument type
10	Second argument value
DOUBLE	Third argument type
15	Third argument value

Server Response:

Error code	0
Error string length	0
Error string	""
Number of arguments	1
First return arg. type	DOUBLE
First return arg. value	3.14159

Figure 3-3: Sample PPServer/client communication.

3.3 The Package System

The PPServer's package system makes it possible for routines to be added to the system at run-time allowing both for the optimisation of existing functions and the addition of new functionality. Several of our principles come into play here. Through the package system we provide an interface to the inner workings of our system and we also make it possible to include third-party libraries.

The package system is basically a mechanism that allows users to write functions in C++ that can be dynamically linked into the server as a shared library. In fact, all basic server functionality is implemented with the package system. Using shared libraries for functionality allows us to reap several benefits. First, the server does not need to be recompiled whenever new functions are needed. In addition, debugging is expedited as packages can be loaded, modified, and re-loaded without having to restart the server. Moreover, if needed, the server can be customised on a per-user basis with only one copy of the PPServer executable. In this case users simply load the packages they need.

Collections of related functions implementing a package are compiled and linked (see Figure 3-4) with any additional libraries to form binary packages (shared libraries). These packages can be loaded into the server at run-time (through the Unix shared library facility) making their routines available to clients. Packages only need to be loaded once and their functions can then be called over and over again.

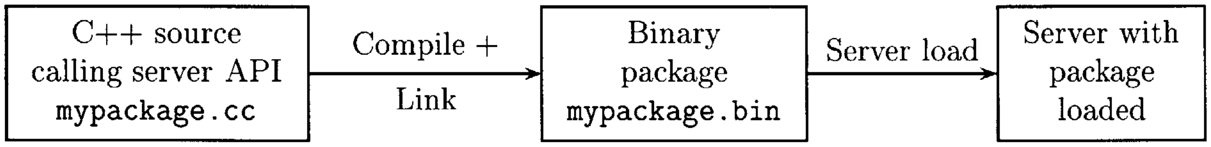

C++ source calling server API mypackage.cc
Compile + Link
Binary package mypackage.bin
Server load
Server with package loaded

Figure 3-4: Steps to package creation and use

3.3.1 Package Management

At server startup, the "base" package gets loaded automatically. It contains most of the basic matrix functionality that the server needs (functions for creating, saving, adding, and multiplying matrices, for example). Clients request that other packages be loaded through the `ppusepackage` server function. Packages can also be "unloaded" with `ppreleasepackage`. This removes their functions from use by the server. When functions in two different packages have the same name, the server always uses the one that was loaded last.

3.3.2 Provided Packages

To date we have implemented a number of PPServer packages, the most important being the base package, `ppbase.pp`. A partial list of this package's routines is given in Table 3.1. A ScaLAPACK package has also been written that makes it possible to call ScaLAPACK functions on PPServer matrices. Its functions are detailed in Table 3.2. The reader is encouraged to consult one of the many books ([18], for example) on numerical linear algebra for descriptions of the algorithms used in these routines.

3.3.3 The Package API

The package API is implemented using two classes, `PPServer` and `PPArg` (see section 3.3.4). These classes give access to the server's data structures and algorithms. More complex functions (such as matrix multiplication, for example) are then built using these methods. Because the PPServer is linked with the MPI library, package creators have direct access to all MPI functions and variables.

The C++ source for a package is usually made up of three main components (see Figure 3-5 for an example):

- Header files and global variable definitions. For example, all packages need to include `PPServer.h` and `mpi.h` so that the API functions and communication services are available.

- Package function definitions. See 3.3.4 for more details.

- The package initialisation and registration routine. When a package is loaded, the server calls the `ppinitialize` routine that must be present. Its main purpose is to register (through `addPPFunction`) with the server the routines that

pp_make_dense	Create a new dense matrix
pp_make_sparse	Create an empty (no element) sparse matrix
pp_load_dense	Load a dense matrix from a file
pp_load_sparse	Load a sparse matrix from disk
pp_view_element	Get the value of a specific matrix element
pp_set_element	Insert a value into the matrix
pp_set_random	Fill a dense matrix with random (uniform [0,1]) values
pp_getcol	Retrieve an entire row or column of a dense matrix
pp_setcol	Set a row or column of a dense matrix
pp_getdist	Return the distribution of a matrix
pp_distribute_by_rows	Make the matrix row distributed
pp_distribute_by_cols	Make the matrix column distributed
pp_gc	Perform garbage collection - delete a subset of matrices
pp pp_transpose	Transpose a matrix
pp_delete	Delete the specified matrix
pp_sum	Sum the rows or columns of a matrix
pp_sumsquares	Find the sum of the squares of the rows or columns of a matrix
pp_cumsum	Perform the prefix operation on the rows or columns of a matrix
pp_sort	Sort the rows or columns of a matrix
pp_nnz	Find the number of nonzeros in a matrix
_mm	Matrix multiplication
pp_add	Matrix addition
pp_sub	Matrix subtraction
pp_getdiag	Get the diagonal elements of a dense matrix
pp_setdiag	Set the diagonal elements of a dense matrix
pp_triu	Extract the upper triangular part of a dense matrix
pp_tril	Extract the lower triangular part of a dense matrix
pp_svds	Perform the sparse SVD (from PARPACK)

Table 3.1: Base package functions

pp_inv	Find the inverse of a matrix
pp_solve	Solve systems of linear equations
pp_schur	Compute the Schur decomposition of a matrix
pp_qr	Compute the QR decomposition of a matrix
pp_svd	Find the complete SVD
pp_chol	Compute the Cholesky decomposition of a symmetric positive definite matrix

Table 3.2: ScaLAPACK package functions

make up the package. Additional initialisation (such as the seeding of random number generators) can also occur here.

3.3.4 Defining Package Functions

Package functions come in two main types: map functions and general functions. Map functions provide an easy way for programmers to apply operations to every element, row, or column of a matrix. General functions give the user full access to the server environment.

Map functions

As an example of a map function consider applying the prefix operation (or Fourier transform) to all of the rows or columns of a matrix. With the PPServer, the only code that needs to be specified is the function that maps a single vector to another vector as in the following example that computes the sine of every row or column of a dense matrix:

```
void sine(int numElements, PPELTYPE *in, PPELTYPE *out, int step) {
  int ptr=0;
  for (int i=0; i<numElements; i++,ptr+=step) {
    out[ptr]=sin(in[ptr]);
  }
}
```

```
#include "PPServer.h"
#include <mpi.h>

/* Two functions are defined here: sumall (find the sum of all matrix
   elements) and sine (compute the sine of all matrix elements) */
.
.
<definitions for sumall and sine functions>
void sumall(...) {}
void sine(...) {}
.
.

/* The initialisation and registration routine */

extern "C" ppinitialize(PPServer &theServer);
PPError ppinitialize(PPServer &theServer)
{
  theServer.addPPFunction("sumall",sumall); // Register sumall
  theServer.addPPFunction("sine",sine); // Register sine
  return(NOERR);
}
```

Figure 3-5: Package layout

Here the vectors are of type PPELTYPE. This type denotes the precision used for server matrices. It is either `float` or `double` depending on the precision chosen for the server. The integer `numElements` specifies the number of elements in the vector and `step` gives the stride. This way we can operate either on the columns (`step`=1) or rows (`step`=local number of rows) of matrices. The two different values of `step` reflect the use of column major order for storing the local pieces of matrices. The columns are contiguous when working down columns, but when working across rows, successive row elements are separated by a distance equal to the number of (local) rows in the matrix.

When the function `mapfunction` is called, the server first ensures that it can be executed locally. For example, if the user wishes to apply the function to the rows of a matrix, the matrix must be distributed by rows. Similarly, the distribution of the output matrix must match that of the input matrix. Redistributions of the argument matrices are performed if necessary. After this, the function is then executed (in parallel) on the input matrix.

Map functions also exist for scalar functions of columns (or rows) and for applying functions to all non-zero elements of sparse matrices.

General functions

The situation is a little more complicated for general functions (see Figure 3-6 for an example). The prototype for such a function is:

```
void genfunction(PPServer &theServer,PPArgList &inArgs,
                 PPArgList &outArgs);
```

The server environment is passed in with `theServer`, the input argument list with `inArgs`, and the output arguments are communicated with the client using `outArgs`.

The input argument list, `inArgs`, is an array of pointers to arguments (much as `argv` is in C programs). To determine the total number of arguments passed, `inArgs.length()` can be called. The primary way of extracting arguments from these pointers is through casting. For example, if an integer is expected in argument

```
void sumall(PPServer &theServer,PPArgList &inArgs,
            PPArgList &outArgs) {
  PPArgType types[] = {PPMATRIXID};
  int mtypes[] = {ANYDENSEBY};

  // Validation: Make sure that the input is a dense matrix id
  if (!(inArgs.validArgTypes(types, 1)) ||
      !(theServer.validIDs(inArgs)) ||
      !(theServer.validMatrixTypes(inArgs,mtypes))) {
    outArgs.addError(BADINPUTARGS,"Expected a dense matrix");
    outArgs.add((PPELTYPE)0.0);
    return;
  }

  // Get the id from the argument list
  PPMatrixID srcID = *(inArgs[0]);
  PPELTYPE mysum=0;
  PPELTYPE theAnswer;

  // Get the local part of the matrix
  PPDenseMatrix *src = (PPDenseMatrix *)theServer.getData(srcID);
  PPELTYPE *srcData = src->data();

  const int tot = src->numRows() * src->numCols();

  // Sum up the local part
  for (int i=0; i<tot; i++) {
    mysum += srcData[i];
  }

  // Sum up over all processors using an MPI function for
  // communication.  Package routines have access to any MPI routine
  MPI_Allreduce(&mysum,&theAnswer,1,PPMPITYPE,MPI_SUM,MPI_COMM_WORLD);

  // Return the answer to the client
  outArgs.addNoError();
  outArgs.add(theAnswer);
}
```

Figure 3-6: Sample function definitions

2, the line:

```
int i= *(inArgs[2]);
```

will place the argument in variable `i`. The scalar types supported in input arguments include `float`, `double`, `char`, `int`, and `PPMatrixID` (for passing matrix identifiers). Vector arguments can also be passed. For example, if the client wants to pass an array of `floats` to the server, the following code will do the trick:

```
int length = inArgs[3]->length();
float *vector = (float *) *(inArgs[3]);
// Now use vector[0..length-1]
```

In addition to extracting arguments, there are validation functions that check the types of arguments passed and return an error if the input is not in order (see Figure 3-6 for an example). At present input error checking needs to be done (if desired) by package programmers. In the future we may look into a more automated mechanism where the types of input arguments expected are registered along with the function in `ppinitialize`. In this scheme arguments are checked before the function is called and this checking could even be turned off for optimised performance.

Output arguments are returned to the client through `outArgs`. Every function must first return an error code and error string in addition to other, more specific arguments. The methods `addNoError` and `addError` automate the return of errors to clients. The rest of the return values are added to `outArgs` through the `add` method. Both scalars and vectors can be passed to clients as the following example shows:

```
// Return a scalar
outArgs.add((double) 3.1415926535);
// Return a vector
outArgs.add(vector,length);
```

With the exception of argument handling, all access to the PPServer's environment goes through the PPServer class. An instantiated object (called `theServer` in all examples) of this class is passed in as the first argument to all general PPServer functions. With this object, package programmers have access to:

- Information about the MPI environment.
- Information about server matrices.
- Other PPServer functions.

Table 3.3 describes the available methods in more detail.

getType	Return the type (dense/sparse) of the matrix argument
isDense	Is the matrix dense?
isSparse	Is the matrix sparse?
getRows	How many rows (globally) does this matrix have?
getCols	The number of columns of the matrix argument
getData	A pointer to local data of the matrix
global2local	Convert global matrix indices to local matrix indices
getBlockSize	How many rows/columns are assigned to each processor
getStartingRow	What's the global row number of local row 0?
getStartingCol	What's the global column number of column 0?
getCommand	Get a function pointer to the specified command
dispatchCommand	Execute another PPServer function
isRoot	Am I processor 0?
processorID	What's my processor number?
numProcessors	How many processors are there?
log	Send a string to the log file
rootlog	Send a string to the log file only if I'm the root

Table 3.3: PPServer methods

Access to the local pieces of PPServer matrices is provided through the `PPMatrix` class and its two subclasses `PPDenseMatrix` and `PPSparseMatrix`. Pointers to these classes are returned by the PPServer's `getData` method. Matrix elements are retrieved and set using the `get` and `set` methods. The sizes of the matrices are obtained through `numrows` and `numcols`. Pointers to the actual data of the matrices are also provided through the `data` (for both dense and sparse matrices) and `indices` (for sparse matrices) methods.

Chapter 4

MITMatlab

In order to present a familiar and transparent user interface to the Parallel Problems Server we developed MITMatlab. MITMatlab offers users:

- Large matrix support through the PPServer.
- Transparency through MATLAB's object oriented features. Very little additional information is needed for MATLAB users to access the power of the PPServer and no special function calls are needed for most PPServer matrix manipulations.
- Ease of application development with MATLAB's scripting language.
- Use of MATLAB's environment for analysis and visualisation of results.

4.1 New MATLAB Classes

MATLAB 5's classes and objects are the mechanism by which transparent access to the PPServer is achieved. Two new classes are defined in MATLAB, `ddense` and `dsparse`, corresponding to the two main types of matrices that are available on the PPServer (distributed dense and distributed sparse). These classes can be viewed as the parallel analogues of MATLAB's own `double` and `sparse` classes. An object belonging to the `ddense` or `dsparse` class holds the id of the associated PPServer matrix and its size (rows and columns). Distributions (row or column) are not kept in MATLAB as these are likely to change on the server.

The MATLAB constructors for these classes have different forms matching the different ways matrices can be created on the server:

`a=ddense(m,n[,dist])` This creates an empty dense matrix on the server using `pp_make_dense` of size `m` by `n`. The optional argument `dist` specifies the distribution of this matrix (`1` distributes by row, 2 by column, and 3 replicates the matrix).

`a=ddense('filename'[,dist])` Here the matrix is loaded in with optional distribution `dist` from file `filename`.

Equivalent `dsparse` constructors (without the distribution argument) also exist.

4.2 Calling the PPServer from MATLAB

As detailed in Chapter 3, any program implementing the client communication protocol can control the PPServer. In MATLAB's case we have provided a function called `ppclient`. This function, written in C++ and linked into MATLAB using its MEX API [25], allows users to send commands to the PPServer and receive the return values. Its syntax is as follows:

```
[code,string,out1,out2,...,outn]=ppclient('functionname',in1,in2,
                                                   ...,inm);
```

The `ppclient` function takes the function name and input arguments, converts them into the format required by the protocol, and sends them through the upstream socket. It then listens on the downstream socket for the server's response. Finally, it converts this byte stream into the corresponding MATLAB variables. When it encounters a `ddense` or `dsparse` object in the input arguments, `ppclient` only forwards the matrix identifier to the server. All other arguments (such as character strings) are sent without modification. For example, to retrieve element (10,15) of the distributed matrix associated with variable `A`, a user would enter:

```
[code,string,value]=ppclient('pp_view_element',A,10,15);
```

The required element will then be placed in variable `value` (if no errors occured). Figure 4-1 shows how `ppclient` is used in the `ddense` constructor.

4.3 Operator Overloading

Operator overloading enables MATLAB users to treat `ddense` and `dsparse` objects in the same way as ordinary MATLAB matrices. They can, for example, type `c=a*b` with `a` and `b` PPServer matrices and have the result `c` be the matrix product of `a` and `b`.

As another example, consider MATLAB's `sum` function. In MATLAB, `s=sum(x,1)` sums up each column of `x` and places these sums in the row vector `s`. Similarly, `s=sum(x,2)` sums up each row. On the PPServer, the `pp_sum` function accomplishes the same task for parallel matrices. We can therefore overload the `sum` function in MATLAB so that it parses the input arguments, creates the output matrix and calls the PPServer using `ppclient`. This function is listed in Figure 4-2.

It is also possible to emulate MATLAB's complex subscripting. MATLAB provides two functions, `subsref` and `subsasgn` that handle expressions such as `B = A(1:3,1:3)` and `A(1:3,1:3)=C`. Here, simple expressions such as `v=A(3,4)` return values to MATLAB while more complicated sections such as `B=A([1 4 5],:)` return PPServer matrices. The decision to keep sections on the PPServer is motivated by the desire not transfer large quantities of data to and from MATLAB. As discussed in Chapter 5, sending huge chunks of data to and from the server severely impacts performance.

We have also overloaded the `display` method for parallel arrays. This function comes into play when commands are entered that do not end in semicolons. For example, `c=a*b` prints out variable `c`. Because inadvertently printing out large matrices is a frequent source of frustration for MATLAB users, we decided that only small matrices (under 50 elements) be displayed. This also minimises client-server communication. For other matrices, only information about their sizes is displayed.

```
function m=ddense(varargin)
% DDENSE Distributed dense matrix class constructor
%     m=ddense(a,dist) creates a distributed dense matrix from the
%     char array a with optional distribution dist.
%     m=ddense(m,m,dist) creates a zeroed matrix of size mxn
numargs = length(varargin);
.
.
if isa(varargin{1},'char')
  % Load the matrix from a file
  m.name = varargin{1};
  if numargs > 1, disttype = varargin{2}; else disttype = 1; end
  % Make the server cd to MATLAB's current working directory
  [errorcode,errorstr] = ppclient('pp_do_cd',pwd);
  % Load the file
  [errorcode,errorstr,m.id,m.rows,m.cols] = ppclient('pp_load_dense',
                                            m.name,disttype-1);
  if errorcode ~= 0, error(errorstr); end
  m=class(m,'ddense');
  return;
elseif isa(varargin{1},'double')
  m.name = 'Internally Created';
  rows = varargin{1};
  .
  .
  cols = varargin{2};
  if numargs > 2, disttype = varargin{3}; else disttype = 1; end
  % Make an empty dense matrix
  [errorcode,errorstr,m.id,m.rows,m.cols] = ppclient('pp_make_dense',
                                            rows,cols,disttype-1);
  if errorcode ~= 0, error(errorstr); end
  m=class(m,'ddense');
elseif isa(varargin{1},'dsparse')
  % Convert from sparse to dense
.
.
else
  error('Input is of wrong type');
end
```

Figure 4-1: The use of `ppclient` in constructors

```
function s = sum(x,d)
% S = SUM(X,D)
% This is the same as the MATLAB sum

if nargin == 1
  if x.rows == 1
    d = 2;
  elseif x.cols == 1
    d = 1;
  else
    d = 1;
  end
end

if d == 1
  s = ddense(1,x.cols,getdist(x));
elseif d == 2
  s = ddense(x.rows,1,getdist(x));
else
  error('Distributed objects only have two dimensions');
end

[errorcode,errorstr] = ppclient('pp_sum',x,d,s);
if errorcode ~= 0
  ppclear(s);   % Delete s
  error(errorstr);
end
```

Figure 4-2: Overloading sum in MITMatlab

Figure 4-3 shows MITMatlab's operator overloading in action.

The technique of invoking procedures on remote data used here is an active area of research. More complete treatments of the issues involved can be found in the Network Objects [5] and CORBA [4] projects. These efforts are primarily concerned with providing support for writing general purpose distributed applications. Our prototype is considerably simpler (we only have matrix objects, for example) and complex issues such as object migration and security are not considered.

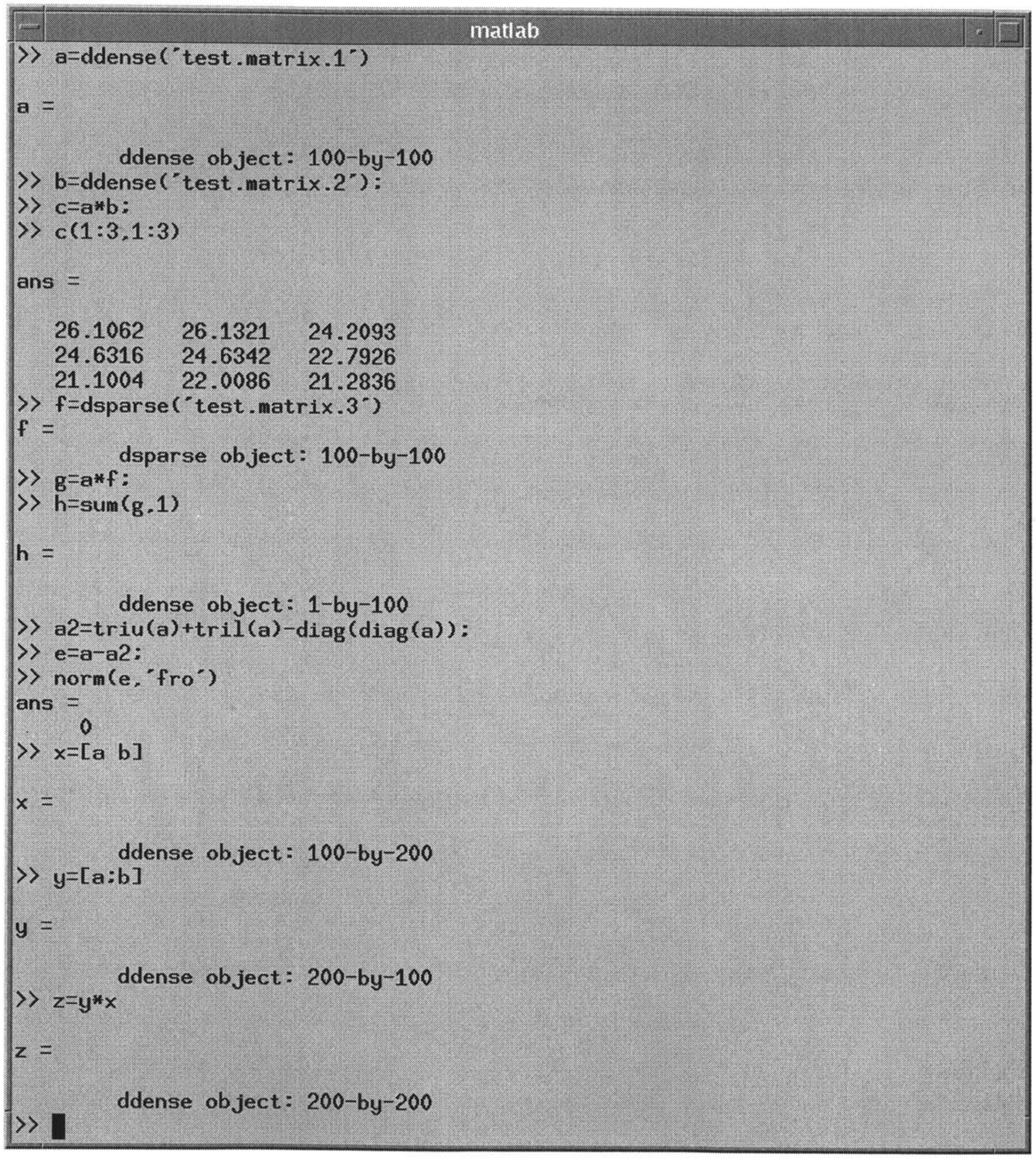
4.4 Code Reuse

The operator overloading discussed in the previous section completes the first step of our "Parallelism through Polymorphism" recipe. It is now possible in most cases for users to write ordinary MATLAB code and apply it to PPServer matrices. Code that calls MATLAB operators on MATLAB matrices does not have to rewritten to use PPServer matrices as long as all of the appropriate functions are properly overloaded. Therefore investments in MATLAB code are not wasted.

However, MATLAB code that creates matrices with its own constructors such as `zeros`, `ones`, `rand`, `randn`, and `sprand` cannot be reused in this way as MATLAB matrices would result. For example, the following code cannot be reused with operator overloading alone:

```
function m=randomeig(n)
% Find the mean of the eigenvalues of a random
% symmetric matrix
a=rand(n,n);
a=a+a';
e=eig(a);
m=mean(e);
```

It would thus be advantageous if the matrices created in this way were parallel and so we proceed with the second step of our recipe.



```
>> a=ddense('test.matrix.1')

a =

        ddense object: 100-by-100
>> b=ddense('test.matrix.2');
>> c=a*b;
>> c(1:3,1:3)

ans =

   26.1062   26.1321   24.2093
   24.6316   24.6342   22.7926
   21.1004   22.0086   21.2836
>> f=dsparse('test.matrix.3')
f =
        dsparse object: 100-by-100
>> g=a*f;
>> h=sum(g,1)

h =

        ddense object: 1-by-100
>> a2=triu(a)+tril(a)-diag(diag(a));
>> e=a-a2;
>> norm(e,'fro')
ans =
     0
>> x=[a b]

x =

        ddense object: 100-by-200
>> y=[a;b]

y =

        ddense object: 200-by-100
>> z=y*x

z =

        ddense object: 200-by-200
>> 
```

Figure 4-3: MITMatlab operator overloading

This is solved by creating a new class (`layout`) for use as the arguments to MATLAB constructors. The members of this class behave in the same way as MATLAB's `double`, but when they appear as arguments to matrix constructors, overloaded functions get called:

zeros(10,10) calls MATLAB's internal routine for creating an empty 10 by 10 matrix

zeros(layout(10),10) calls the overloaded function `zeros` for the `layout` class.

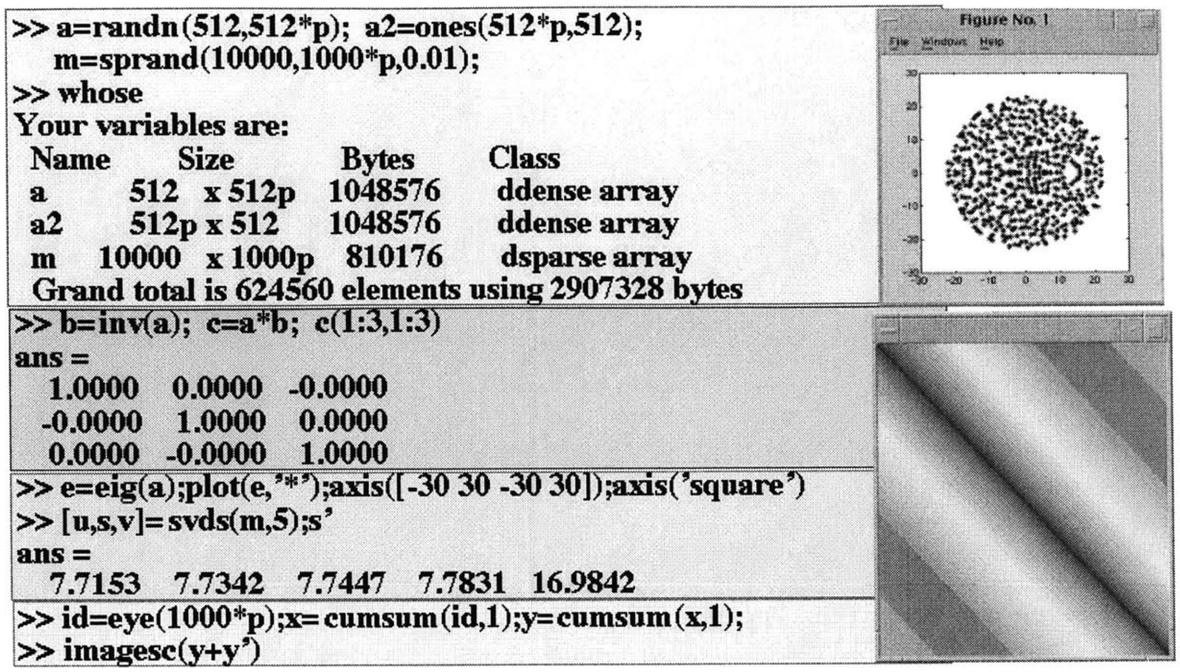
Easy creation of `layout` variables is provided by the `p` function. This function always returns `layout(1)`. Because the class has been designed so that arithmetic on layout variables yields layout variables, any layout variable can be easily constructed as a multiple of p: `layout(100)` $\equiv$ `100*p`.

In constructors the position of layout variables plays a role in the distribution of the created matrix; `zeros(100*p,100)` results in a row-distributed matrix while `zeros(100,100*p)` gives a column-distributed matrix. General block distributions are not supported on the server and so `zeros(100*p,100*p)` is row distributed. Figure 4-4 illustrates the use of `p` in matrix constructors and further operator overloading on the resulting matrices.

The use of `p` is not just limited to constructors. Matrix information functions such as `size` return layout variables. For example, `size(zeros(100*p,100))` returns `[100*p 100*p]`. Layout variables are returned in both places so that any matrix derived from a parallel matrix is also parallel (and so maximise the possibilities for introducing parallelism). For example, both `b` and `c` are parallel matrices in the following code:

```
a=zeros(100*p,100);
[m,n]=size(a);
b=zeros(1,2*m);
c=zeros(n,n);
```

In addition, constructions such as `x=1:n` (the vector `[1 2 ... n]`) result in equivalent distributed objects if `n` is a layout object. This makes it possible to support



```
>> a=randn(512,512*p); a2=ones(512*p,512);
   m=sprand(10000,1000*p,0.01);
>> whose
Your variables are:
 Name       Size        Bytes      Class
 a       512  x 512p   1048576     ddense array
 a2      512p x 512    1048576     ddense array
 m     10000  x 1000p   810176     dsparse array
 Grand total is 624560 elements using 2907328 bytes
>> b=inv(a); c=a*b; c(1:3,1:3)
ans =
   1.0000   0.0000  -0.0000
  -0.0000   1.0000   0.0000
   0.0000  -0.0000   1.0000
>> e=eig(a);plot(e,'*');axis([-30 30 -30 30]);axis('square')
>> [u,s,v]=svds(m,5);s'
ans =
   7.7153   7.7342   7.7447   7.7831   16.9842
>> id=eye(1000*p);x= cumsum(id,1);y=cumsum(x,1);
>> imagesc(y+y')
```

Figure 4-4: p in MITMatlab

a common style of MATLAB programming; code that creates matrices often starts with a : construction:

```
x=1:n;
y=x';
z=sqrt(x*y);
```

Of the many alternatives considered, this behaviour is ideal. The result of `1:10*p` can either be stored in MATLAB or on the server. The consequences of both choices are discussed below:

- If 1:10*p returns an ordinary MATLAB matrix, it cannot be subscripted with a distributed array due to a restriction in MATLAB:

  ```
  A=1:10*p;
  I=ones(10*p,1);
  C=A(I,:)
  ```

- If 1:10*p returns a layout array we risk losing performance due to extra client-server communication if this possibly large array is used in another computation:

  ```
  A=1:1000000*p;
  B=ones(1000000*p,1);
  C=A*B;
  ```

- Returning a distributed array does not adversely affect for loops:

  ```
  1 for i=1:n
  2   a(i,i)=2;
  3 end
  ```

  When MATLAB parses line 1 above, if `n` is a layout object, a distributed matrix is created; however, the semantics of MATLAB's `for` loops dictate that `i` is assigned to each *column* this matrix in turn. Retrieving a column of a distributed object returns a MATLAB variable, and so `i` gets assigned the *numbers* 1 through n.

As an example of the use of layout variables, consider the function `hilb` (Figure 4-5 shows the code). This function is part of MATLAB and 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 2 creates a PPServer object with $1, 2, \cdots, n$ and places it in `J`.
- `ones(n,1)` in line 3 produces a PPServer matrix.
- Emulation of Matlab's indexing functions results in the correct execution of line 3.
- Overloading of `'` (transpose) in line 4 makes `I` a parallel array.
- In line 5, `E` is generated on the PPServer because of the overloading of `ones`.

```
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 4-5: MATLAB code for producing Hilbert matrices.

- 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 Test Matrix Toolbox [20] without any modification. Much of the work of this task involves supporting the multitude of Matlab's indexing capabilities and built-in functions. In addition MATLAB's iterative solvers `pcg` and `gmres` also work "out of the box". Successes from Higham's Toolbox include `cauchy`, `circul`, `clement`, `cycol`, `dingdong`, `frank`, `kahan`, `lehmer`, `moler`, `parter`, `pei`, and `triw`. Some routines (`kms`, `orthog`, `seqm`, `signm`, and `smoke`) need support for complex numbers, an important extension not currently in the PPServer. Others (such as `hadamard` and `wilk`) that return explicit MATLAB matrices cannot be overloaded in this way. Part of the code for `hadamard.m` is summarised in Figure 4-6. It shows that even though `k` is a layout variable, `H` will never be distributed because `k` (and by extension `n`) has no direct influence on the construction of any part of `H`.

Users of our system should therefore be aware of the way parallelism gets propagated through our system so that their code benefits from the use of the PPServer. The rules are simple:

- Operations on parallel matrices give parallel matrices
- Arithmetic on layout variables yields layout variables
- `1:n*p` produces a parallel matrix
- Use of layout variables in constructors results in parallel matrices

```
function H=hadamard(n)
% Construct a Hadamard Matrix H, such that:
% 1) H(i,j) = +1 or -1
% 2) H*H'=n*eye(n).
% Assumes n is a power of 2.
k=log2(n);
H=[1];
for i=1:k
  % Double the size by making a block matrix
  H=[H  H
     H -H];
end
```

Figure 4-6: MATLAB code for Hadamard matrices

```
function H=hilb(n)
H=zeros(n,n);
for i=1:n
  for j=1:n
    H(i,j)=1/(i+j-1);
  end
end
```

Figure 4-7: Alternative `hilb.m`

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 matrices. For example, even in double precision, `pascal(800)` (the numbers of Pascal's triangle), contains `Inf`. Further, functions that make extensive use of element-wise operations will not make full use of the parallelism of the PPServer. For example, the alternative `hilb` implemented with loops in Figure 4-7 is much less efficient than the one in Figure 4-5 as the server cannot exploit any parallelism. It is instructive to note that the data-parallel version of `hilb.m` (Figure 4-5) performs better even in traditional MATLAB and this style of programming is encouraged where possible, fitting well with our model.

4.4.1 A bottom-up view of p

In the preceding discussion, we started with the PPServer and then introduced `p` as a way of getting parallel execution of a larger subset of MATLAB code. In this section we present another view of the process, starting MATLAB, adding different versions of `p` and ending with the full MITMatlab environment. Figure 4-8 explains the steps taken in "reverse" order.

MATLAB with:	Effect on code reuse
`p=1`	None Code runs in MATLAB environment
`p=layout(1)`	Arithmetic and functions using `layout` need to be overloaded for code reuse. Routines still run in MATLAB environment
`p=layout(1)` + PPServer	Parallel execution of MATLAB routines

Figure 4-8: The steps to MITMatlab

4.5 Language Features

In this section we discuss some new features of the MITMatlab "language".

4.5.1 An Interpreted HPF?

The MATLAB language shares many similarities with HPF (High Performance Fortran) [27]. These include the use of array syntax (`c=a+b` can be used to add two matrices without using loops), triplet notation (`1:2:n`), and array sections (`a(1:10,1:5)`). MITMatlab also introduces parallelism in a similar way. In HPF parallel arrays are created using `DISTRIBUTE` compiler directives that specify how the arrays are partitioned among processors. For example, a 1000 by 1000 row-distributed single precision matrix is declared using:

```
      REAL, DIMENSION(1000,1000) :: A
!HPF$ DISTRIBUTE A(BLOCK,*)
```

In MITMatlab, a similar matrix is created with `A=zeros(1000*p,1000)`.

With HPF's directives other distributions (such as cyclic) can also be specified. As opposed to the *block* distributions currently implemented in the PPServer where row (or column) i is assigned to processor i div n/p (with n the number of rows (columns) and p the number of processors), *cyclic* distributions assign row (column) i to processor i mod n/p. *Block cyclic* distributions also exist, where the rows (or columns) of a matrix are first partitioned into blocks of a certain size and these blocks are then distributed to the processors in a cyclic fashion. For example, the ordinary cyclic distribution can be viewed as a block cyclic distribution with a block size of 1.

These additional distributions are not currently implemented in our prototype, but can easily be accommodated using our framework. Cyclic distributions can be described with `c` (or another suitable letter), instead of `p` as Table 4.1 demonstrates.

MITMatlab	HPF
`A=zeros(1000*p,1000*p)`	`A(BLOCK,BLOCK)`
`A=zeros(1000*c,1000)`	`A(CYCLIC,*)`
`A=zeros(1000*c,1000*p)`	`A(CYCLIC,BLOCK)`
`A=zeros(1000*c(10),1000)`	`A(CYCLIC(10),*)`

Table 4.1: Specifying distributions in HPF and MITMatlab

4.5.2 An Important Constraint: Dealing with Pointers

While MATLAB supports object-oriented programming, it does not yet implement true user-defined *destructors* (as in C++). Consequently, there is no way for an object to be notified automatically when it is about to be deleted, either explictly via MATLAB's `clear` or implicitly by going out of scope. To make matters worse, there is no way in the curent version of MATLAB to obtain a list of all active variables using a function. These combine to make it impossible to have automatic garbage collection in MITMatlab. The inclusion of destructors in a future version of MATLAB would eliminate this problem.

MITMatlab 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.

Another issue relating to object oriented support in MATLAB is the inability to define a *copy constructor*. In MATLAB, when `a=b` is encountered, the bytes for variable `b` are copied to form the bytes for variable `a`. However, in the case of `ddense` and `dsparse` variables, copying the bytes means that both `a` and `b` point to the same PPServer matrix. We would like to be able to (as in C++) define a *copy constructor* that allows us to create a copy of `b` on the server, but it is impossible with the current version of MATLAB. As a result the following code does not execute as expected:

```
b=ones(10*p,10);
a=b;
a(1,1)=2;
c=a(1,1)-b(1,1);
```

The variable `c` should contain the value 1, but because `a` and `b` are the same matrix, `c` equals 0. As with the garbage collection issue, this results from the fact that we use MATLAB variables to hold "pointers" to data stored elsewhere. The problem of garbage collection is one of "dangling references" and the copy constructor problem deals with "aliasing". In MATLAB, storage for the class variables (and other native types) is automatically reclaimed, pointers are not followed.

Chapter 5

System Performance

Due to the architecture of MITMatlab, overall system performance depends on several factors in addition to the quality of the computational routines residing in the PPServer. These factors can be classified as:

- Client Communication
- Memory Management
- Additional Server Overhead

After discussing these issues in sections 5.1–5.3, the operational performance of MITMatlab is presented in section 5.4 and compared with other implementation strategies.

5.1 Client Communication

Because MITMatlab is a client-server system, the cost of operations includes the time it takes to communicate with the server. In experiments on a DEC Alpha system, a simple ping to and from the server averages 2 ms (round trip). Some of this time (about 0.4 ms) can be attributed to MATLAB overhead in calling the `ppclient` function. The rest of this time it taken up by socket communication and server execution.

These costs conspire to make accessing a single matrix element relatively expensive. Compared to MATLAB's 141 us, the statement `v=A(i,j)` takes 3 ms when `A` is a server matrix. This means that MITMatlab applications fare poorly when forced to transfer many small matrix pieces to or from the PPServer. In addition, functions that execute very quickly on the server will be dominated by the client communication involved.

5.2 Memory Management

In MATLAB code many statements involve assignment (for example, `c=a*b` creates `c`). If the assignment is to an entire parallel variable (as opposed to an element, `A(i,j)`, or section, `A(i,:)`), a new PPServer variable is created. Because we do not have access to MATLAB's parser, simple operations and assignments to current variables (as in the following code) also create new matrices:[1]

```
a=ones(10,10*p);
a=2*a;
```

Combined with the necessary garbage collection that this entails, we find that MITMatlab applications spend a non-trivial amount of time managing memory. By contrast, in most compiled code, programmers take great pains to ensure that new variables are created only when necessary. In Section 5.4.4 we perform some experiments that quantify the effects of memory management on our applications.

5.3 Server Overhead

The performance of server computational functions is primarily dictated by their implementation. For example, the time taken by `inv(A)` in MITMatlab depends mainly on the quality of the ScaLAPACK routines `pdgetrf` and `pdgetri`. In addition

[1]For this reason, we have implemented a separate "self" function (`scaleself`)that allows `a=2*a` to execute without wasting memory.

to this there is the overhead of finding the right function to call (negligible on the DEC Alpha), possible data redistributions, and of course, the performance of the underlying MPI library.

5.3.1 Data Distribution

In some cases, the server spends time redistributing matrix data to comply with external library constraints. Libraries such as ScaLAPACK (but not S3L) require their input to be distributed in a certain way. For example, in a call to `pdgehrd` (the routine that reduces matrices to Upper Hessenberg form), the input matrix must be 2-D block distributed with square blocks. Because PPServer dense matrices do not conform to this distribution, a copying and re-alignment phase must precede each call to this routine (and others).

Redistributions from row to column (and vice-versa) also take place on the server prior to certain operations. For these operations certain distributions are preferable to others but the cost of redistribution impacts their performance. In our current system there are a few operations that begin with a redistribution step. In the future we hope to rewrite these routines where necessary so that they are executed "in place". By minimising redistributions we achieve higher performance while giving users more control (if desired) on the placement of their matrices. However MATLAB is a declaration-free language and matrices are only created when assigned to (`C=A+B` creates `C`). While we have tried to make reasonable choices for the distribution of target matrices (such as `C`), there are some cases where these will not be optimal for performance.

5.3.2 Speed of MPI

The quality of the MPI implementation used with the server has a great impact on the performance of server routines. As an example, consider the server's matrix multiplication routine (`C=A*B`). In most cases it broadcasts each local piece of matrix `A` and performs a local multiplication and sum. As observed in [22], if great care is not taken with the MPI broadcast implementation on a network of SMPs, dismal performance may result. Most routines implement a tree-like broadcast algorithm and assume that the performance of each link between processors is identical. However, in a clustered environment this is not the case and collective operations that incorporate this knowledge into their communication patterns exhibit much better performance. In MPI-StarT [22], a two-stage broadcast algorithm is implemented where data is first broadcast to all machines and then broadcast within each machine.

Table 5.1 compares the performance of the original and optimised broadcasts in MPI-StarT on a network of Sun E5000s. In the benchmark, each process broadcasts a different 1 MB buffer in turn and the time for such a "round" is recorded. In the table (and in all subsequent experiments) 2+2 means 4 processes equally divided between two machines.

p	Optimised	Original
1+1	0.148	0.147
2+2	0.466	0.781
4+4	1.158	2.420
8+8	3.591	6.934

Table 5.1: MPI-StarT broadcast performance

This improvement translates directly into enhanced matrix multiplication performance.

5.4 Performance Summary

5.4.1 Evaluation Platforms

Hardware

The following sections detail the results of experiments performed on dedicated Sun Ultra Enterprise 5000 and Digital AlphaServer 4100 clusters. The Sun machines each contain 8 167MHz Sparc Ultra 1 processors, 512 MB of main memory and are connected either with 100 Mbit/s Ethernet or with MIT's Arctic Network. The Digital cluster is comprised of 4-way SMPs each with 466 MHz Alpha processors, 2GB of main memory and uses MemoryChannel as its interconnect.

Software

The majority of the results here were obtained via MATLAB scripts. For example, timing a single matrix multiply is as easy as:

```
a=rand(1000*p);b=rand(1000*p);
tic;c=a*b;t=toc;
```

The time for the operation can be extracted from the variable `t`. As a consequence of this, the times reported include client communication and reflect the performance that MITMatlab users see.

5.4.2 Distribution Costs

Table 5.2 summarises the cost of converting matrices of various sizes between row and column distributions on the DEC cluster. These matrices are all in single precision and distributed among 2, 4, 6, and 8 processors.

The time for conversion to ScaLAPACK's block form is also presented in Table 5.3.

	Processors							
	2		4		3+3		4+4	
n	r → c	c → r	r → c	c → r	r → c	c → r	r → c	c → r
1024	0.179	0.179	0.097	0.103	0.110	0.081	0.085	0.066
2048	1.521	1.334	0.578	0.577	0.418	0.361	0.376	0.312
4096	8.498	8.770	5.996	5.644	3.153	2.542	3.085	2.216

Table 5.2: Redistribution costs for $n \times n$ dense matrices

To weigh the impact of this operation we must consider what happens to the matrix after it has been redistributed. For complex operations that take a while (such

n	Processors			
	2	4	3+3	4+4
1024	0.108	0.050	0.051	0.047
2048	0.854	0.403	0.212	0.202
4096	1.947	1.998	1.175	0.932

Table 5.3: Costs of converting $n \times n$ column distributed dense matrices (single precision) to ScaLAPACK form

as those in Section 5.4.3) we see that redistribution adds only a little extra time (for example, on the Suns it takes 5.85s to redistribute a 2K x 2K single precision matrix, but 69.5s to multiply two 2K x 2K matrices). However, for simple operations it should be avoided.

5.4.3 Individual Server Operations

The performance of large, complex linear algebra operations is summarised in this section. As an example of this, we compare the costs of finding the Cholesky factorisation of a large matrix using MITMatlab (and ScaLAPACK) against a compiled code using the Sun Performance Library (Sun's optimised version of LAPACK) on the Sun E5000. Table 5.4 presents these times along with the time that MATLAB takes to perform the same operation on the same matrix. Here the matrices are all in double precision (so that the comparison with MATLAB is fair). Note that these times include the time to convert the PPServer matrix to ScaLAPACK form.

	Processors								
		2		4		6		8	
n	MATLAB	MIT	Sun	MIT	Sun	MIT	Sun	MIT	Sun
1024	22.78	5.45	3.19	3.03	2.46	2.30	2.17	2.19	2.04
2048	202.17	31.26	24.95	16.17	16.51	11.94	13.45	9.95	12.11

Table 5.4: Cholesky decomposition time of the $n \times n$ Moler matrix

The results show that the MITMatlab/PPServer approach is competitive with using the optimised library and clearly outperforms MATLAB on the same machine.

For such operations we basically get the performance of the underlying library.

The ubiquitous matrix multiplication operation also exhibits speedups compared with MATLAB. Table 5.5 shows the performance of our our single precision implementation on the Sun SMP cluster using MIT's Arctic network and MPI-StarT.

n	Processors							
	1+1	2	2+2	4	4+4	8	8+8	8+8+8
1024	9.7	9.6	4.8	4.7	2.6	3.0	2.0	3.0
2048	69.5	69.4	35.1	35.0	17.5	17.7	10.6	9.5
4098	NA	NA	403.5	402.9	204.2	207.8	102.9	79.1

Table 5.5: $n \times n$ Matrix multiplication performance

One advantage of our software is its ability of perform computations that would not be feasible with traditional MATLAB. To illustrate this, consider finding large sparse singular value decompositions. The algorithm incorporated into the PPServer uses the PARPACK library and deals quite easily with very large matrices. As an example, we created 10K by 10K random sparse matrices with 1, 2, and 4 million nonzero elements and then attempted to find their first five singular triplets. MATLAB failed to complete the computation but the MITMatlab approach found the required triplets and even exhibited speedups when more processors were added. The results are detailed in Table 5.6.

size	Processors			
	2	4	3+3	4+4
1M	116.8	60.6	43.6	37.7
2M	301.6	158.4	119.5	95.7
4M	425.5	316.7	205.0	167.0

Table 5.6: Sparse SVD performance

5.4.4 Combined Operations

All of the issues discussed in Sections 5.1-5.3 come into play when MATLAB scripts are run in MITMatlab. To better understand the effects of these factors, we implemented the same program in many different ways and analysed the execution time of each version. Figure 5-1 shows the MATLAB code for our test program. It basically performs many matrix/vector products and vector additions in a loop. The following versions of the code were created in:

1. MATLAB
2. MITMatlab – running the MATLAB code on PPServer arrays. In this version, `ppscope` and `ppgc` (see Chapter 4) calls were added for garbage collection.
3. MITMatlab, but implementing the entire program as a package (see Figure 5-2)
4. FORTRAN 77 using LAPACK.

```
A=rand(3000,3000);
x0=rand(3000,1);
Q=rand(3000,9);
n=10;

function X=testfun(A,x0,Q,n)
% Form columns of X in the following way:
% Column i+1 of X is the product of column i
% of X added to column i of Q

X(:,1)=x0;
for i=1:n-1
  X(:,i+1)=A*X(:,i)+Q(:,i);
end
```

Figure 5-1: MATLAB code for program experiment

Table 5.7 shows the results of the experiments. Not surprisingly the native Fortran version is the fastest; however, the PPServer package version does not incur a substantial performance penalty. The interpreted MITMatlab version, while still

```
void pp_mc(PPServer &theServer,PPArgList &inArgs,PPArgList &outArgs)
{
  // First get the arguments
  PPMatrixID A=*(inArgs[0]);PPMatrixID x0=*(inArgs[1]);
  PPMatrixID Q=*(inArgs[2]);int n=*(inArgs[3]);
  PPMatrixID X=*(inArgs[4]);
  // First make sure that x0,X, and Q row distributed.
  .....
  // First copy x0 to the first column of X, x(:,1) = x0
  PPDenseMatrix *matX = (PPDenseMatrix *)theServer.getData(X);
  PPDenseMatrix *vecx0 =(PPDenseMatrix *) theServer.getData(x0);
  PPDenseMatrix *matQ = (PPDenseMatrix *)theServer.getData(Q);
  int i;
  for(i=0;i < matX->numRows();i++) {
    matX->set(i,0,vecx0->get(i,0));
  }
  // Create temp vector vecx1 and set x1 to be x0
  .....
  // Now implement the loop
  PPArgList multArgs;
  for(int j=1;j < n;j++) {
    // We're setting X(:,j) in this iteration
    // First multiply A by X(:,j-1)
    multArgs.clearArgs();
    multArgs.add(A);  multArgs.add(x1); multArgs.add(x2); //x2=A*x1
    multArgs.add(0); // Don't transpose A
    answer=theServer.dispatchCommand("pp_mm",multArgs);
    delete answer;
    // Make sure x1 and x2 are row distributed
    .....
    // Because the local data may have changed because of the
    // redistribution get new pointers
    vecx2=(PPDenseMatrix *) theServer.getData(x2);
    vecx1=(PPDenseMatrix *) theServer.getData(x1);
    // Add Q(:,j-1) to x2 and set it to both x1 and X(:,j)
    for(i=0;i < vecx2->numRows();i++) {
      vecx1->set(i,0,vecx2->get(i,0)+matQ->get(i,j-1));
      matX->set(i,j,vecx1->get(i,0));
    }
  }
  outArgs.addNoError();
}
```

Figure 5-2: The package version of the MATLAB code

faster than the pure MATLAB version, was predictably slower than the two compiled versions. It had to manage the temporary variables that were created in the loop and incurred a little overhead for every server function called. We believe that this small cost is well worth the advantages we obtain in ease of implementation (a simple MATLAB script) and interactivity.

Processors Used	Time (sec)			
	Fortran	Server Package	MITMatlab	MATLAB
1	5.50			49.93
2	2.94	3.18	3.62	
4	1.60	1.78	2.25	
6	1.13	1.60	2.12	
8	0.97	1.76	2.73	

Table 5.7: Performance of different implementations of the combined operations test program

Table 5.8 shows the times for calling `moler(n)` from Higham's Toolbox. This routine creates an $n \times n$ matrix using mostly data parallel operations. Various values of `n` were tested on the DEC Alpha platform in double precision. The results show that for small sizes MATLAB is faster due to the overhead involved in the parallel operations but the PPServer shines for larger sizes. This behaviour is typical of most parallel computing systems and MITMatlab is no exception; the gains in parallelism are only seen when the problem sizes are large enough to justify the overhead involved in parallel execution.

5.4.5 The Big Picture

Based on the results presented in this chapter, we can draw a few conclusions on the performance of the MITMatlab environment. The main point is that code that performs complex operations on large PPServer matrices is very competitive. By contrast, code that uses many elementwise operations and accesses fares very poorly. It is interesting to note that this is also the case for sequential MATLAB code.

n	MATLAB	Processors			
		2	4	3+3	4+4
10	0.020	0.192	0.133	0.194	0.154
50	0.004	0.175	0.134	0.192	0.148
100	0.027	0.190	0.138	0.196	0.155
200	0.208	0.243	0.173	0.231	0.173
500	3.016	0.947	0.555	0.571	0.452
1000	42.95	4.78	2.84	2.45	1.95
2000	352.1	27.84	19.01	14.09	11.99

Table 5.8: Times (in seconds) for computing Moler matrices

The use of `for` loops and elementwise assignments is eschewed in favour of the data-parallel style of programming which is suitable for many scientific computing tasks (as evidenced by the proliferation of languages such HPF). Further, using the PPServer for operations on small matrices may not result in performance gains due to the overhead involved.

5.5 Accuracy

In addition to the usual accuracy and stability concerns of numerical routines discussed at length in numerical analysis books such as [18], the use of parallelism in the PPServer introduces issues of *coherence* and *repeatability.*

Loosely speaking, an algorithm is termed *incoherent* if "values that should be the same on all processors somehow aren't". A common way this can happen is when the value is the result of some global operation. For example, if all processors have a floating point number x and wish to find the global sum of all the local xs, it is possible that this sum could be different on each processor. This can occur if the values are added up in a different order on each processor as floating point addition is not associative ($a + (b + c)$ does not always equal $(a + b) + c$).

Repeatability is the property that "different runs of the algorithm obtain the same results". As an example, consider the algorithm of Figure 5-3 that computes a global sum. Because the values sent to processor 0 can arrive in any order, the sum can be

```
if processor number != 0
  send x to processor 0
  wait for sum to be broadcast from processor 0
else
  sum=my local x
  for i=1:number of processors-1
    receive a value v from some processor
    sum=sum+v
  end
  broadcast sum to all other processors
end
```

Figure 5-3: Computing a global sum

computed in any order. The non-associativity of floating point addition results in a nonrepeatable algorithm.

It is therefore up to library writers to ensure that their algorithms are coherent and repeatable. ScaLAPACK for example, ensures coherence and repeatability if the following three conditions are met:

- The processors are identical
- The message passing library used to compile ScaLAPACK does not convert messages while sending them
- The same binary program is executed on each processor

These conditions hint at the role the MPI implementation plays in guaranteeing reliable computation. For example, a bad implementation of the `MPI_Allreduce` function (that performs global operations such as sums) could be incoherent and/or nonrepeatable.

The situation is considerably more complicated in a heterogeneous computing environment. In such environments the processors may have different floating point formats and use non-IEEE arithmetic. These and other issues are discussed in [7].

Finally, the precision used in the PPServer is also an issue. Programmers (and users) must take care not to use single precision when accuracy is required. For example, in single precision `hilb(1000)*invhilb(1000)` does not return the identity matrix (`invhilb` computes the inverse of a Hilbert matrix using a formula).

Chapter 6

Applications

In a research setting MITMatlab combines MATLAB's ease of development with the ability to operate on very large data sets. This provides users with the ability to use MATLAB for applications instead of having to build special purpose tools written in a compiled language.

In this chapter we describe some demonstrations of this idea, presenting IRLAB, a research Information Retrieval system "written" mainly in MATLAB, and summarising other applications that use MITMatlab. The key point here is that only familiarity with MATLAB and domain-specific (Information Retrieval, for example) knowledge is necessary to build the applications described here. Familiarity with the intricacies of parallel programming is not a prerequisite.

6.1 IRLAB

Efficiently searching large collections of documents for relevant articles is one of the most challenging problems in Computer Science. Compounding the problems of scale (such collections typically reach several gigabytes in size) are those of semantics. Natural language words can have several meanings (polysemy) and many different words can refer to the same concept (synonymy). As a result, individual words are often insufficient to disambiguate meaning. Thus, a crucial issue in Information Retrieval (IR) involves deriving representations that capture as much meaning as possible while remaining space and time efficient (see [16] for a discussion of the many issues involved in building IR systems).

The Vector Space Model (VSM) [41] is one of the most popular representations in IR. In this model, each document is represented as a vector v where v_i is some number reflecting the existence of term i in the document. This number is usually some function of the number of times the term appears in the document and often takes into account global (across the entire document collection) term counts. Very frequent terms ("how" being an extreme example), often have little use in distinguishing relevant from non-relevant documents. To mitigate the effects of these terms various term-weighting schemes have been developed. In IRLAB we use the TF-IDF strategy. The entire document collection can then be represented as a term by document matrix.

This model is particularly appealing because it is easy to reason about concepts such as relevance and document similarity using well known (and understood) operations on vectors. For example document/query similarity is usually the inner product between the respective vectors. In a production environment (such as a Web search engine) inverted indices optimise the task of finding documents that contain a given set of terms (and hence relevant documents).

The work of IR systems is to find relevant documents. To evaluate their effectiveness they are usually run against sets of queries for which relevant documents have been tagged (usually by humans). Based on the documents returned and the documents judged relevant measures of the system's accuracy can be computed. One of the standard measures of retrieval performance is precision-recall. Retrieval systems typically rank the documents in order of relevance to a particular query. The *precision* at rank i is the fraction of documents at ranks $1, \ldots, i$ that are relevant to the query. The *recall* at this point is the fraction of total relevant documents retrieved. Precision-recall curves are usually plotted (precision on the y-axis and recall on the x-axis) to give graphical representations of the effectiveness of different retrieval methods (see Section 6.1.2). In general, "higher" curves indicate better performance. Figure 6-1 shows a "perfect" precision-recall curve where all relevant documents are returned first.

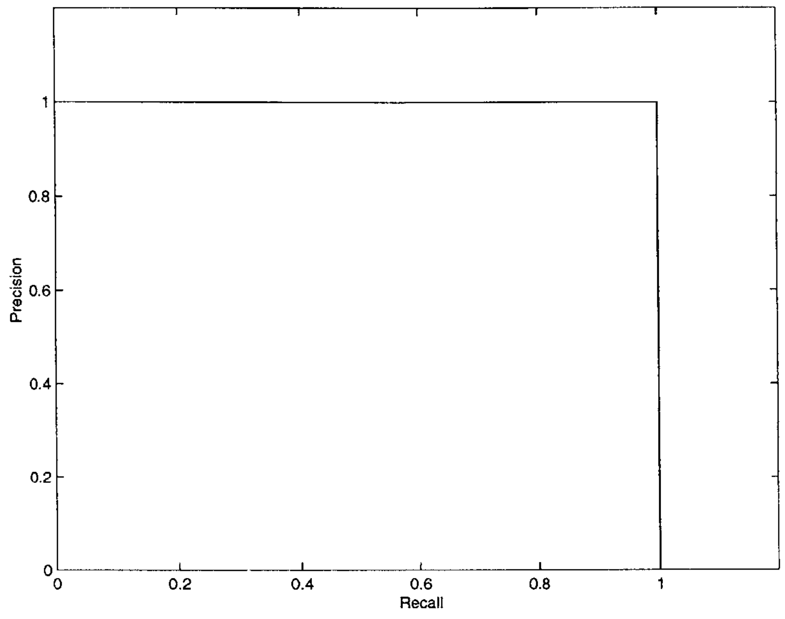


Figure 6-1: A perfect precision-recall curve

6.1.1 Dimensionality Reduction

Dimensionality reduction techniques attempt to represent the data in the term-document matrix A by a reduced matrix B which has fewer rows (the number of columns, i.e. documents, stays the same). If the documents were originally represented as d (the number of documents) points in an m-dimensional vector space (where m is usually the number of terms), they will now be points in a k-dimensional space ($k \ll m$). The hope of these techniques is that the important relationships among the documents will be more easily revealed in the smaller space. Dimensionality reduction has been used successfully in other fields. Factor Analysis has been used for most of this century to summarise multivariate data. Principal Components Analysis is widely used in Pattern Recognition applications such as face recognition [44].

Linear projection schemes are some of the more widely used dimensionality reduction techniques. Here a $k \times m$ matrix P (called a projection matrix) is found such that the reduced matrix $B = P \cdot A$.

One of the first linear projection schemes was introduced by Deerwester, Dumais, et. al. [13]. Called Latent Semantic Indexing, it used the Singular Value Decomposition (see [18] for properties and algorithms) of A to find P. If the SVD of A is written as $A = USV'$, then the LSI Projection Matrix is $S^{-1}U'$.

It is interesting to note that using LSI, the reduced matrix B may not necessarily occupy less space than A. Since most documents only contain a small fraction of all possible terms, A will be sparse. If A is encoded using one of the popular sparse matrix encoding schemes, it can use much less than the $m \times d$ floating point locations that a naive scheme would need. However, in LSI, the reduced matrix is dense and so needs all $k \times m$ floating point locations. Depending on the choice of k, more space than the original matrix may be required.

Most dimensionality reduction approaches, however, rely on matrix decompositions that are computationally intensive and so the sheer size of the data matrices used in IR has greatly hindered their usefulness.

6.1.2 Using MITMatlab

The large matrix support in MITMatlab enables IR researchers to focus on refinement of techniques instead of directly dealing with problems of scale. Term-document matrices are simply `dsparse` objects and the myriad of provided matrix operations form the building blocks for retrieval schemes.

IRLAB is basically a collection of MITMatlab scripts and programs that are used to create document retrieval applications. The functionality of IRLAB can be broken down into three main areas:

- Text conversion programs
- Matrix manipulation routines

- IR-specific functions

Text conversion

Text collections do not usually come in term-document form. As such, a number of external Unix programs are provided that take collections of text and produce the matrices that are used in MITMatlab.

We assume that all incoming documents and queries are in SMART [41, 11] format. The first step is to convert into another format (called HI) that delimits sentences and paragraphs (in case such information is needed by the researcher). This is accomplished by the `filter-stopwords` program. It also removes stopwords (commonly used words) and has a facility for part of speech tagging using the PRINCIPAR parser [30].

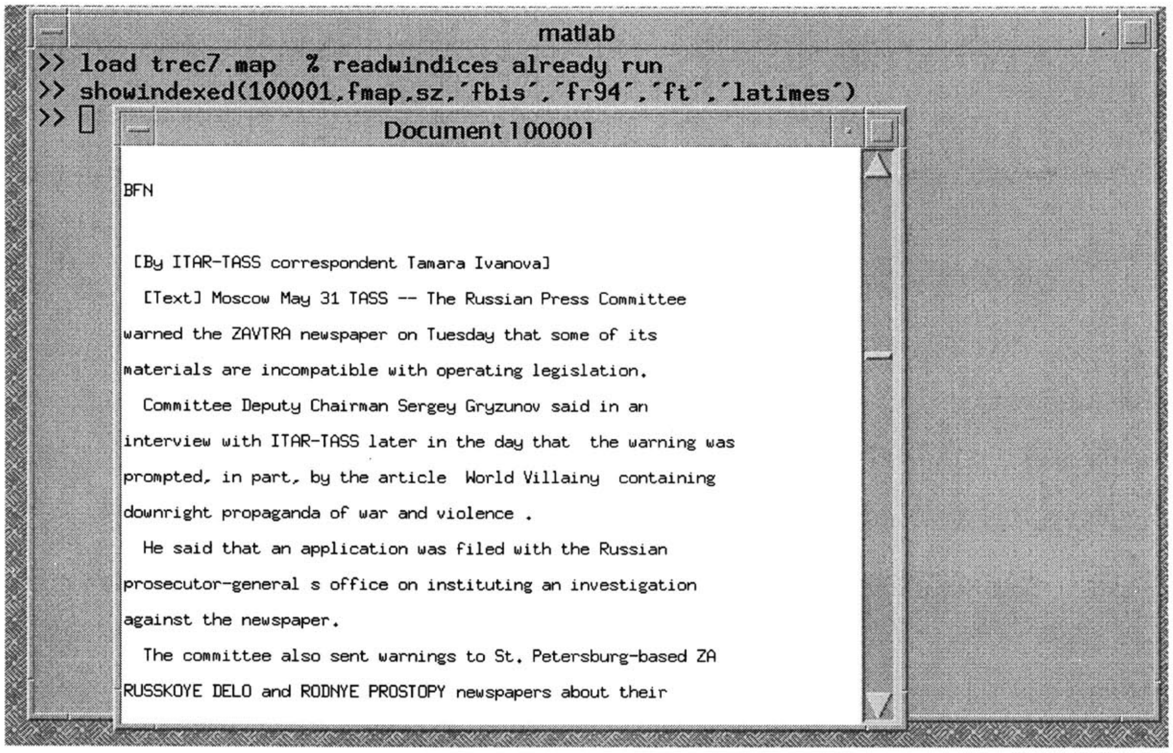
Next, `text2matrix` converts a document collection from HI format to a term-document matrix readable by the PPServer. `qtext2matrix` converts a set of queries in HI format to term-document matrices. For this program the mapping from words to indices is required so that term i in both the document and query matrices refer to the same word.

If pre-computed relevance judgements exist for the document/query pair, `qrel2-matrix` takes these judgements and produces a sparse document-query matrix R, where $R(i, j) = 1$ if (and only if) document i is relevant to query j.

Finally, `indexdocs` takes the SMART document collection and produces files that enable easy access to the text of any document in the collection. The MATLAB functions `readwindices` and `showindexed` use this information to display individual documents within MATLAB. Figure 6-2 shows these functions in action.

Matrix manipulation

Because `ddense` and `dsparse` objects are produced by the programs detailed in the above section, application designers have access to all PPServer operations. Some have been extended to deal more efficiently with the scale of the data under consideration.



matlab

```
>> load trec7.map  % readwindices already run
>> showindexed(100001,fmap,sz,'fbis','fr94','ft','latimes')
>> 
```

Document 100001

```
BFN


 [By ITAR-TASS correspondent Tamara Ivanova]
  [Text] Moscow May 31 TASS -- The Russian Press Committee
warned the ZAVTRA newspaper on Tuesday that some of its
materials are incompatible with operating legislation.
  Committee Deputy Chairman Sergey Gryzunov said in an
interview with ITAR-TASS later in the day that  the warning was
prompted, in part, by the article  World Villainy  containing
downright propaganda of war and violence .
  He said that an application was filed with the Russian
prosecutor-general s office on instituting an investigation
against the newspaper.
  The committee also sent warnings to St. Petersburg-based ZA
RUSSKOYE DELO and RODNYE PROSTOPY newspapers about their
```

Figure 6-2: Displaying documents in IRLAB

"Self" versions of some frequently used operations are provided so that an additional copy of the matrix arguments are not created. These operations transform a matrix without allocating new space on the server and include `scaleself` that scales the rows or columns, `normself` that normalises columns, `powerself` that raises all entries to a specific integer power, and `idfself` discussed in the next section.

The matrix decomposition functions `svds` (Sparse Singular Values) and `pcas` (Principal Components Analysis) have been broadened to work on matrices derived from products in the following way:

`[u,s,v]=svds(A,5)` returns the five largest singular triplets of A, but
`[u,s,v]=svds(A,B,C,D,5)` operates on the matrix $A \cdot B \cdot C \cdot D$ without ever forming the product (which may be prohibitively large).

The functions described above were programmed into the PPServer directly in an attempt to optimise performance. In fact, these optimisations are relevant to ordinary (serial) MATLAB code as well. It would be possible to implement IRLAB without them, but the routines would not be as time and space efficient.

IR-specific functions

A few commonly used IR operations have already been provided in MITMatlab. These are written entirely in MATLAB (no PPServer modifications were made) and include:

- `idf=idfself(D)` extracts the TF-IDF weights for server term-document matrix `D`, puts them in `idf`, and applies these weights to `D`. .
- `sc=getdqscores(D,Q)` performs simple term-matching retrieval on documents `D` and queries `Q` and places the document/query scores in `sc`
- `sc=getdddqscores(D,Q)` computes $sc = D'DD'Q$. The idea behind this technique is to use document/document similarity to assist document/query similarity. $D'D_{ij}$ gives the similarity of document i to document j and $D'Q_{mn}$ gives the similarity of document m to query n. If these two matrices are multiplied,

then

$$D'DD'Q_{kl} = \sum_{d=1}^{nd}(D'D)_{kd}(D'Q)_{dl}$$

In other words, if document k is similar to document d and document d is similar to query l then the score of document k and query l is increased.

- `sc=getlsiscores(U,S,V,Q)` computes the document/query scores based on LSI: $sc = V'S^{-1}U'Q$
- `[pr,re]=precisionrecall(sc,R)` evaluates the average precision and recall based on scores sc and relevance matrix `R`.

Examples

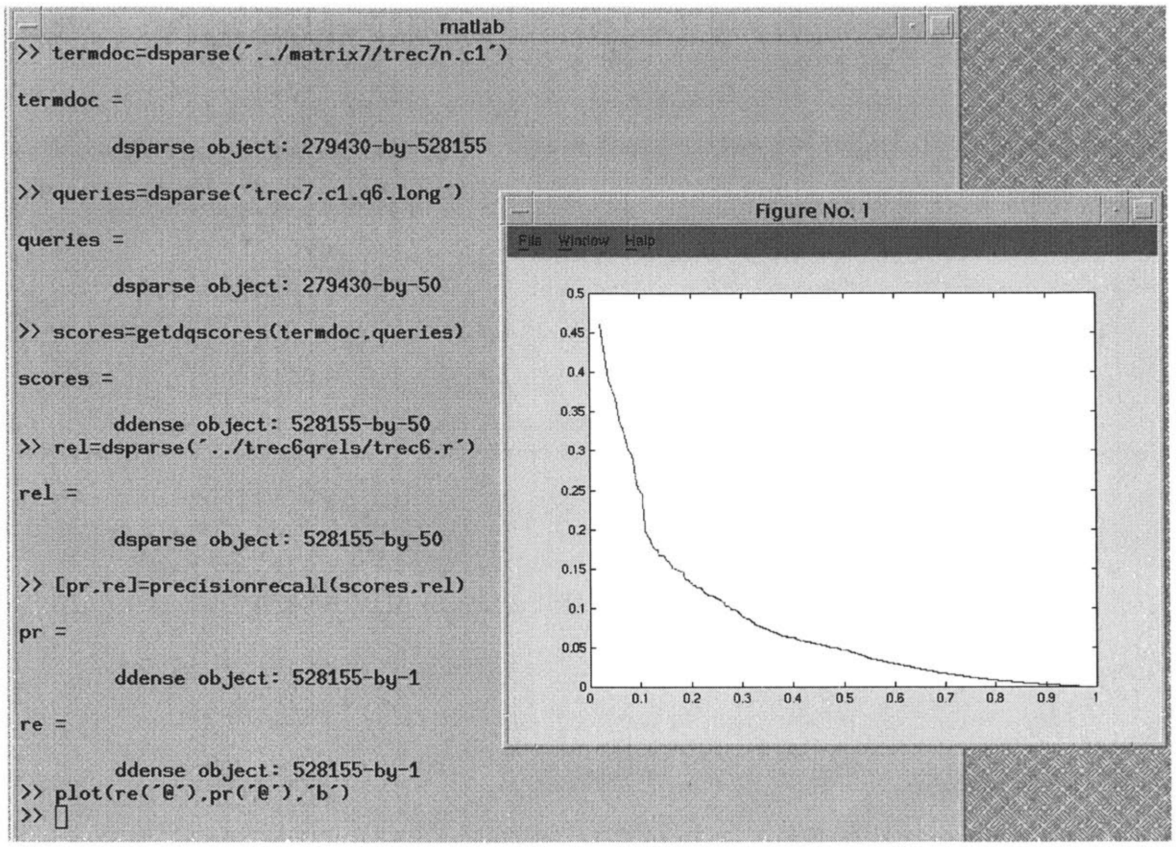
Term Matching

The easiest and most common IR technique used today is term matching. Here the document similarity to the query is measured in terms of the number of words in the document that are contained in the query. A number of variants of this idea (in addition to term weighting described above) exist and are easily implemented in IRLAB:

Boolean matching The MATLAB function `bool` takes a sparse matrix and replaces all nonzero elements with 1. Combined with dot products we get document/query scores that reflect the number of times document words match query words.

Normalising documents Long documents with terms that appear many times will seem more favourable to term matching algorithms. In order to "level the playing field", each document is typically normalised to have unit length (in the Euclidean sense). The `normself` function normalises each column of a term-document matrix in place.

Figure 6-3 shows how these ideas come together in IRLAB. Here we perform term matching on term weighted and normalised documents. The performance of the method can be seen by viewing the associated precision-recall curve. This method is usually used as the baseline for retrieval experiments.



```
>> termdoc=dsparse('../matrix7/trec7n.c1')

termdoc =

        dsparse object: 279430-by-528155

>> queries=dsparse('trec7.c1.q6.long')

queries =

        dsparse object: 279430-by-50

>> scores=getdqscores(termdoc,queries)

scores =

        ddense object: 528155-by-50
>> rel=dsparse('../trec6qrels/trec6.r')

rel =

        dsparse object: 528155-by-50

>> [pr,re]=precisionrecall(scores,rel)

pr =

        ddense object: 528155-by-1

re =

        ddense object: 528155-by-1
>> plot(re('@'),pr('@'),'b')
>>
```

Figure 6-3: Term matching in IRLAB

Using SVDs

Another retrieval technique involves the use of the Singular Value Decomposition of the term/document matrix. The hope of this technique is to project the documents into a lower dimensional space where similar documents would "move" closer together, thus improving retrieval performance. Using the `svds` function in MITMatlab we can easily implement this idea:

```
D=dsparse('term-doc');
Q=dsparse('queries');
```

```
[U,S,V]=svds(D,200);

sc=getlsiscores(U,S,V,Q);
```

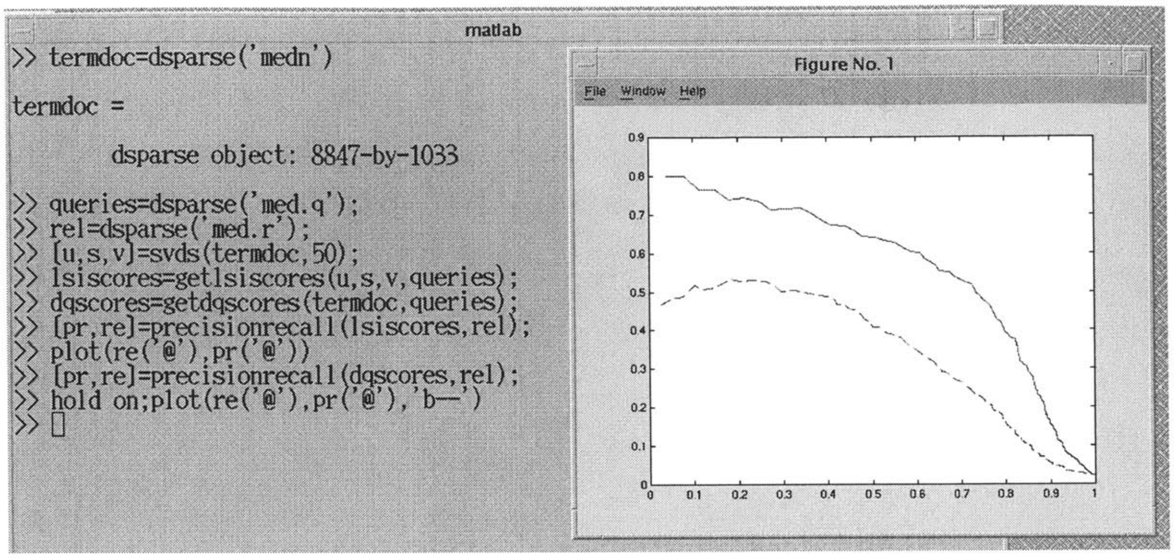
The `getlsiscores` function simply projects the queries into the lower dimensional space and computes their inner products with the projected documents (see Figure 6-4).

```
function c = getlsiscores(u,s,v,q)
% C = GETLSISCORES(U,S,V,Q) returns the dot products of the LSI
% documents (V)
% and the queries (Q).  It uses U and S to project the queries
% according to
% the rule S^{-1}U'Q

if isa(u,'double')
  temp1 = u'*q;
else
  temp1 = atb(u,q);  % a'*b in one operation
end
oneOvers = 1./diag(s);
if isa(temp1,'ddense')
  temp2 = oneOvers'*temp1;
  ppclear(temp1);
else
  temp2 = diag(oneOvers)*temp1;
end
c = v*temp2;
if isa(temp2,'ddense')
  ppclear(temp2);
end
```

Figure 6-4: `getlsiscores.m`

Computing SVDs for very large sparse matrices is an extremely computationally intensive process. As such, the example shown here is for a very small collection (only 1033 documents). The only difference between this example and larger collections is the time necessary to compute the decomposition. Figure 6-5 shows how LSI can be used in IRLAB and its precision-recall performance relative to term matching (LSI is the solid line).



```
>> termdoc=dsparse('medn')

termdoc =

        dsparse object: 8847-by-1033

>> queries=dsparse('med.q');
>> rel=dsparse('med.r');
>> [u,s,v]=svds(termdoc,50);
>> lsiscores=getlsiscores(u,s,v,queries);
>> dqscores=getdqscores(termdoc,queries);
>> [pr,re]=precisionrecall(lsiscores,rel);
>> plot(re('@'),pr('@'))
>> [pr,re]=precisionrecall(dqscores,rel);
>> hold on;plot(re('@'),pr('@'),'b--')
>> 
```

Figure 6-5: LSI in IRLAB

Other schemes

More complex retrieval schemes can also be easily implemented with our software. For example, due to the computational complexity of SVDs, we may only want to perform it on a subset of the documents and use the projection matrix on the entire collection. The MATLAB code of Figure 6-6 accomplishes this.

```
nd=size(D,2);
scores = getdqscores(D,q);
indices=sortself(scores);           % Sort document/query scores
topdocs=indices(nd-n+1:nd,1);       % Get the top documents
newdocs=D(:,topdocs);               % Select these documents
[U,S]=svds(newdocs,m);              % Find the new SVD
scaleself(U,1./S,2);
newscores=getdxxqscores(D,U,q);     % Find the new scores, (U'D)'(U'q)
```

Figure 6-6: Performing LSI on a subset of documents

6.2 Other Applications

As seen with IRLAB, applications needing large matrix support can be quickly developed without having to explicitly deal with problems of scale. MITMatlab has been used as a platform for teaching the concepts of parallel scientific computing. In a research setting, it has aided efforts in Ocean Modelling and Machine Learning.

In the Ocean Modelling example, the researchers attempted to validate their ocean model by comparing predicted sea surface heights with actual observations from satellite data. They already had a collection of MATLAB programs to do the task, but these did not run on the production data due to its size. These programs were modified slightly (as reported before, loading of matrices from disk uses a different syntax) and run successfully using MITMatlab. Their results and conclusions are discussed in [32].

Chapter 7

Conclusions

7.1 Future Directions

MITMatlab currently possesses the majority of the core linear algebra routines that large scale scientific applications need. Based on this infrastructure several extensions to the software are possible that expand the range of easily supported applications.

7.1.1 Zero Finding and Optimisation

A class of problems where reuse of MATLAB software seems particularly challenging is zero finding and optimisation. In these problems a function f is presented and its zeros (x s.t. $f(x) = 0$) or extreme points (x s.t. $f(x)$ is maximised or minimised) are requested. The difficulty is that users specify the function in an interpreted MATLAB script. For example, `fmin('myfun',0,1)` in MATLAB finds the minimum of `myfun` in the interval from 0 to 1. The obvious problem is that the PPServer does not contain a MATLAB interpreter and thus it seems impossible to interactively evaluate such functions. This problem does not only arise with minimisation. Any MATLAB function that takes another as an argument (thus using code as data) cannot be easily parallelised with the PPServer.

A solution that we believe is promising and wish to further explore is to quickly compile the function at runtime using a serial MATLAB compiler and then dynamically load the produced object code (a la packages). Multiple instances of the function with different arguments may then be evaluated in parallel and so the optimisation can be carried out on the server. A preliminary study of this approach using PGA-Pack [28], a parallel genetic algorithms package, was successfully carried out as a class project. In this package, the various parameters of the algorithm (such as the fitness and mutation functions) are specified as C functions. In order to fully integrate this package into MATLAB's environment these functions need to be implemented as ordinary MATLAB scripts. The project achieved this by first compiling the MATLAB scripts to C++, running an automated script to remove unnecessary statements and declarations from the C++ source, compiling the C++ to an object file, and then loading it into the server.

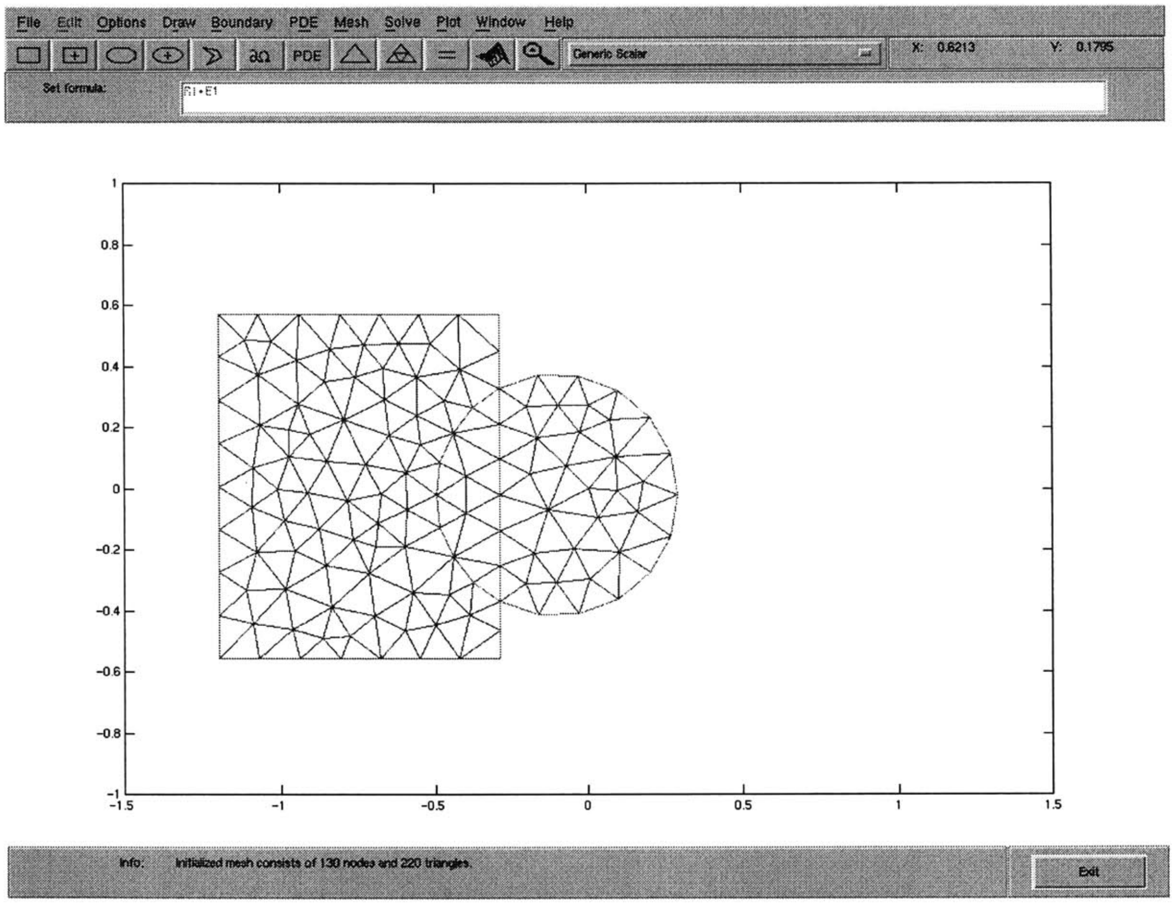
Another approach would be to use MATLAB's interpreter on each MPI process to evaluate any MATLAB code that gets passed on to the PPServer. The main disadvantage of this is that the MATLAB interpreter would not run in the same address space as the PPServer and so arguments would have to be unnecessarily copied. For this problem we believe that the perfect solution would be the inclusion of MATLAB's interpreter (in some way) within the PPServer's address space.

In the context of optimisation, further enhancements using MATLAB's symbolic toolbox are also planned. Using this feature, algebraic expressions may be manipulated symbolically. Jacobians and Hessians (first and second derivatives that are used in zero-finding and optimisation routines) can thus be computed automatically, freeing programmers from the tedium of hand computations.

7.1.2 Partial Differential Equations

MATLAB already provides a flexible interface for solving Partial Differential Equations in its PDE toolbox [24] (see Figure 7-1 for a screendump of the toolbox in action).

This software makes it easy to express complex domains, specify the equation to be solved, and provide boundary conditions. The solution is then performed in serial and results can be graphically displayed for further analysis. For example, Figure 7-2


File Edit Options Draw Boundary PDE Mesh Solve Plot Window Help
∂Ω
PDE
Generic Scalar
X: 0.6213
Y: 0.1795
Exit

Set formula: R1+E1

Info: Initialized mesh consists of 130 nodes and 220 triangles.

Figure 7-1: The PDE Toolbox

shows a representation of the solution to Poisson's equation ($u_{xx} + u_{yy} = -10$, $u = 0$ on boundary) that was generated with the toolbox and Figure 7-3 shows the first eigenmode of $u_{xx} + u_{yy} = -\lambda u$.

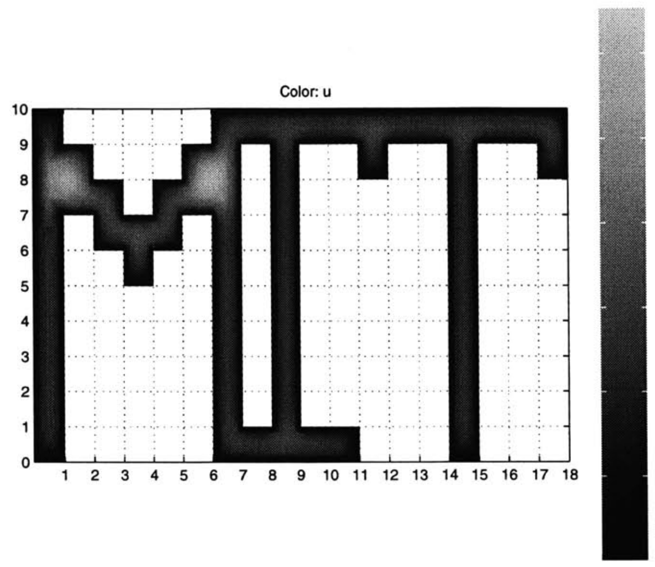


Figure 7-2: Solution of Poisson's Equation

We would like to extend the functionality of the PDE toolbox by making it possible to work with larger, more realistic problems in an intuitive way and of course, be able to reap the benefits of parallel computing. As always we attack the problem from both the computation and interface ends.

First we plan to include more support in the Parallel Problems Server for solving the large (often sparse) linear systems that arise in this application. We already have routines for solving dense linear systems, and we are working on integrating PETSc's sparse solvers (mainly Krylov Subspace Methods and preconditioners) into MITMatlab.

With these solvers in place it will be possible to specify the geometry and parameters with the toolbox, export the equations to the PPServer and find the solution in parallel. This method of introducing parallelism may not be the most efficient as the matrix is first formed in MATLAB and then shipped over to the server. The next step would be to only export the geometry and have the matrix assembly and

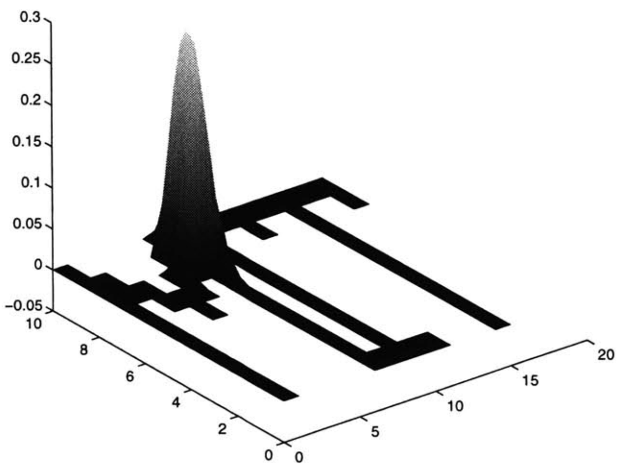


Figure 7-3: First eigenmode of $u_{xx} + u_{yy} = -\lambda u$

solution take place in parallel.

In our proposed system the user draws a figure made up of many regions. These regions are then distributed in parallel by the server, and the solution of the PDE obtained by domain decomposition [42]. The KeLP [15] library from UCSD contains routines for the management of many parallel domains and we are in the process of incorporating its functionality into the PPServer.

A further extension is to add grids (discretised areas of n-dimensional space) as first class data objects into MITMatlab itself. Even the serial MATLAB PDE Toolbox uses ordinary sparse matrix structures for specifying grids with no further abstractions. Our proposal is to introduce new classes and methods for grids so that, for example, common multigrid [10] operations such as refinement, coarsening, and operator restriction are easily expressed. The grids will be server objects and therefore distributed in parallel. We intend to create a simple stencil language for specifying operators (on these grids) and using the features of PETSc and KeLP we intend to allow ghost images from neighbouring grids to be communicated correctly. In multigrid, many coarsening and restricting operators may be implemented easily using the descriptions provided in the stencil language. The stencil language can be designed for both regular and irregular grids.

7.1.3 Queueing

The resources of supercomputers are traditionally managed by *queueing systems* that ensure controlled (or even exclusive) access to the machine. In this model *batch jobs* (user programs) are submitted and are run when the required resources become available.

This style of use presents many challenges for our system. At MIT our users have two options:

- Submit a (Unix) shell job and run MATLAB interactively when the shell starts running.
- Submit a MATLAB script and have it execute (in parallel) without user intervention.

The first approach suffers from unpredictability as users do not know exactly when their shell will be run. In the second approach all of the interactivity is lost.

At present, we do not have a solution that completely satisfies both batch and interactive users. Several proposals, with varying degrees of implementation complexity, have been suggested. These range from the preemption of long running background jobs by interactive sessions to the close interaction of the PPServer with the queueing system on a per-MATLAB operation basis.

7.2 Applicability for Scientific Research

There are many different criteria for determining the suitability of a programming language (or environment) to a particular task. In our case we believe that expressiveness, portability, extensibility and interactivity are paramount and we designed MITMatlab with these considerations in mind.

In [36], Pancake and Bergman discuss the properties of parallel languages that make them useful to scientific researchers. These properties were proposed in the context of compiled languages but it is instructive to see how MITMatlab measures up to their standards. The four "desirable characteristics" proposed are:

Convenience *Because of its popularity, a FORTRAN-like syntax would be beneficial. Few additional language constructs should be introduced, but the language should support fine-grained parallelism.*

It could be argued that today, MATLAB has joined FORTRAN as one of the dominant languages in scientific programming. Because we haven't changed any MATLAB internals, MITMatlab only introduces the concept of the distributed array (`p`), thus keeping learning time to a minimum. However, support for fine-grained parallelism is absent. The easiest way to provide this support would be to add a "parallel `for` loop" in MATLAB. This would necessitate a change in MATLAB's design and unfortunately there is some evidence [33] that this will not occur.

Reliability *Existing language constructs should work as before (with only a few changes). The well-known FORTRAN model of storing and operating on data should be supported. Critical sections and barriers should be included. Programmers should know where the parallelism lies in their applications.*

In our system, we have tried to exactly emulate MATLAB's functionality and so existing constructs (and code) work as before. The PPServer supports the FORTRAN model of data layout (column major). Because parallel execution only takes place on distributed arrays, programmers know exactly what sections of their code execute in parallel. As with support for parallel loops, critical sections and barriers would require changes to MATLAB itself.

Expressiveness *It must be easy for the language to express frequently used techniques in scientific programming. Programmers should be able to specify data distributions and operators need to be provided for working directly with these distributed arrays.*

The rich library of functions that MATLAB supports (with extension to MITMatlab via the package system) makes it possible for a wide variety of techniques to be easily implemented. In MITMatlab we have user-specified distributions and operators for processing parallel arrays (by using operator overloading).

Compatibility *The language should be portable across different architectures and be "reasonably efficient". It is also paramount that the language have access to visualisation routines and exploit available parallel libraries.*

By using MPI, MITMatlab gains portability and efficiency as well as a host of functionality through available software libraries. As demonstrated in Chapter 6, we can also use MATLAB's visualisation routines.

While the features listed above are certainly important, users will ultimately decide on the suitability of our work. As an interactive supercomputing tool, we believe MITMatlab offers the community many important features not found (in total) either in traditional supercomputing tools or desktop environments. To conclude we list these features and point out how they are provided in our tool:

- Interactive "Scientific" syntax. Our evidence: MATLAB
- Ability to interface to parallel libraries. Our evidence: The PPServer and its package system
- A return on users' investment in code. Our evidence: Over 70% of Higham's Toolbox runs without modification. In addition code such as `pcg` and `gmres` also work with our system. The fruits of the "Parallelism through Polymorphism" tree.
- Large scale support. Our evidence: The PPServer's ability to distribute matrices across multiple processors.

Appendix A

Parallel Computing Models

The promise of parallel computing is to deliver improved application performance through the use of multiple cooperating processors. This appendix summarises contemporary ways of characterising and programming parallel machines.

A.1 Main Models

The two major models of parallel computing differ in the way the processors access main memory. In the *shared-memory* model, each processor can read and write to any memory location. In the *distributed-memory* model, processors can only access their own local (private) memory. In this model processors determine the contents of other processors' memory only through the exchange of *messages*. Figure A-1 gives a graphical description of the two models.

A.2 Programming Shared Memory Machines

The main challenge in programming shared memory machines is ensuring controlled access to memory. For example, many programs maintain a queue of pending work and a common way of parallelising programs is to have processors add work to and remove work from this queue. The add (or remove) operation is seldom *atomic* as many separate memory operations are needed and interleaving these operations would

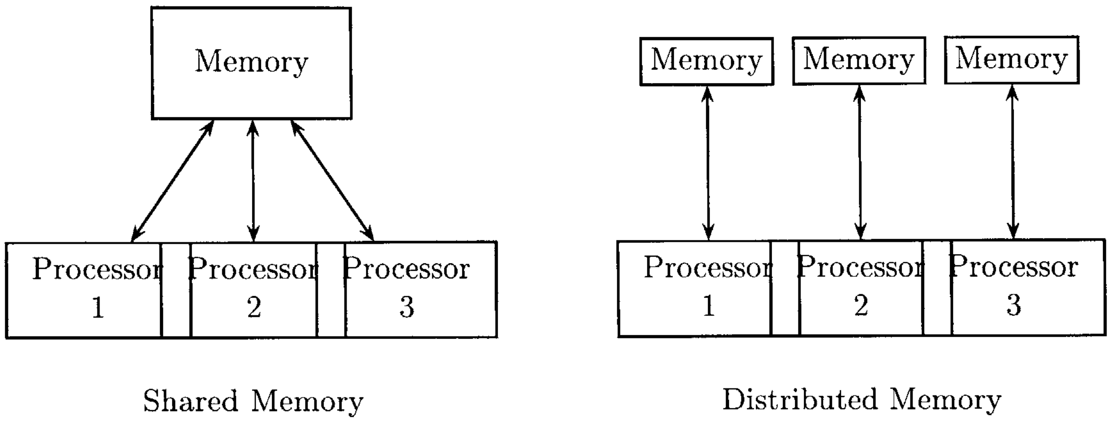


Figure A-1: Parallel computing models

result in an inconsistent queue.

Programming support for overcoming these difficulties is provided in the form of *synchronisation primitives*. The simplest of these is a *lock*. If the shared queue of the example above has an associated lock, a processor wishing to modify its contents can (atomically) *acquire* the lock, make changes, and then *release* the lock. If other processors adhere to the protocol of only modifying the queue when holding the lock, the queue will never be in an inconsistent state.

It is sometimes desirable for the processors to wait until all processors have completed a specific task. This is accomplished by calling a *barrier* function. The semantics of this function dictate that no processor returns until all processors have called the function.

Critical Sections can be viewed as locks for code. They ensure that only one processor is able to execute a particular piece of code (such as modifying the queue's contents). In the Unix world, libraries such as PThreads [35] give users access to functions for these and other synchronisation primitives.

Popular machines of this model are the so-called "Symmetric Multiprocessors"

(SMPs) such as the Sun Ultra Enterprise series and Digital Alphaserver series.

A.3 Programming Distributed Memory Machines

As each processor's memory cannot be accessed by other processors in distributed memory machines, synchronisation is not needed to manipulate memory. However, no processor has a global view of main memory. In order for the processors to cooperate *messages* need to be exchanged. For example, if the queue is contained in the local memory of one processor, other processors must send messages to it in order to add or delete elements. The data used by the algorithm is therefore *distributed* among the processors. The most common programming libraries used with this model are MPI [19] and PVM [17]. They contain routines for sending and receiving different datatypes between pairs of processors as well as *collective operations* such as *reductions* (finding the sum, product, or logical and of data in each processor's memory).

There have also been efforts (TreadMarks [26], for example) at building *Distributed Shared Memory* systems. These systems are distributed memory systems at the hardware level but have an extra layer of software that enable them to behave (from a programming point of view) like shared memory machines.

Machines such as Thinking Machines' CM-5, Cray's T3D are distributed memory machines. Clusters of workstations linked together with message passing software also fall into this category.

A.4 Automatic tools

In order to spare programmers the burden of writing their applications in an explicitly parallel way (using MPI or PThreads), parallel extensions have been made to contemporary programming languages. In these languages the compiler is then responsible for generating parallel code.

A widely used example of this style of programming is HPF (High Performance Fortran) [27]. The commonly used FORTRAN language is extended to include mechanisms for specifying the distribution of data (by the programmer) and parallel loops. In addition it supports data parallel programming where operations on the distributed data can be carried out with one statement (or function call). For example `C=A+B` performs the addition (in parallel) of matrices `A` and `B`. Many extensions to the C/C++ family such as pC++ [9] have also been developed. New languages such as ZPL [29] and NESL [8] where parallelism is an integral part of the programming model have also been proposed.

Compilers for sequential languages have also been enhanced. Any loop where iterations do not depend on each other can be easily parallelised by allocating different iterations to different processors. Even loops with some dependencies (such as those that compute some reduction operators) can be automatically parallelised by the compiler. Programmers can also give hints by using compiler directives, identifying pieces of code that can be parallelised.

Parallel subroutine libraries are also very common. Usually tuned to a particular model (shared or distributed memory) these libraries provide parallel implementations of commonly used routines. Programmers can then view their program as basically sequential with library calls that work on their data in parallel.

Bibliography

[1] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. PLAPACK: Parallel linear algebra package. In *Proceedings of the SIAM Parallel Processing Conference*, 1997.

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

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries.* Birkhauser Press, 1997.

[4] Ron Ben-Natan. *CORBA:A guide to Common Object Request Broker Architecture.* McGraw-Hill, 1995.

[5] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. Technical Report 115, DEC Systems Research Center, 1995.

[6] 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.* Society for Industrial and Applied Mathematics, 1997.

[7] L. S. Blackford, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, A. Petitet, H. Ren, K. Stanley, and R.C. Whaley. Practical experience in the

dangers of heterogeneous computing. Technical Report CS-98-330, Computer Science Department, University of Tennessee, Knoxville, July 1996.

[8] Guy E. Blelloch. NESL: A nested data-parallel language (3.1). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, 1995.

[9] François Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), 1993.

[10] William Briggs. *A Multigrid Tutorial.* Society for Industrial and Applied Mathematics, 1987.

[11] C. Buckley. Implementation of the SMART information retrieval system. Technical Report 85-686, Cornell University, 1985.

[12] Henri Casanova and Jack Dongarra. NetSolve: A network server for solving computational science problems. In *Proceedings of SuperComputing 1996*, 1996.

[13] S. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the Society for Information Science*, 41(6):391–407, 1990.

[14] Peter Drakenberg, Peter Jacobson, and Bo Kågström. A CONLAB compiler for a distributed memory multicomputer. In *Proceedings of the Sixth SIAM Conference on Parallel Processing from Scientific Computing*, volume 2, pages 814–821. Society for Industrial and Applied Mathematics, 1993.

[15] S. J. Fink, S. R. Kohn, and S. B. Baden. Efficient run-time support for irregular block-structured applications. *To appear in Journal of Parallel and Distributed Computing*, 1998.

[16] William B. Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.

[17] Al Geistand, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[18] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1993.

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

[20] Nicholas J. Higham. The Test Matrix Toolbox for MATLAB (version 3.0). Numerical Analysis Report No. 276, Manchester Centre for Computational Mathematics, September 1995.

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

[22] P. Husbands and J. C. Hoe. MPI-StarT: Delivering network performance to numerical applications. In *Proceedings of SC98*, 1998.

[23] 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.

[24] The MathWorks Inc. Partial differential equation toolbox users' guide, 1995.

[25] The MathWorks Inc. Application program interface guide, 1996.

[26] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 94 Usenix Conference*, 1994.

[27] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Jr. Guy L. Steele, and Mary E. Zosel. *The High Performance FORTRAN Handbook.* MIT Press, 1994.

[28] David Levine. Users' guide to the PGAPack parallel genetic algorithm library. Argonne National Laboratory, Mathematics and Computer Science Division, 1996.

[29] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In *Proceedings of the 6th International Workshop of Languages and Compilers for Parallel Computing*, 1993.

[30] Dekang Lin. PRINCIPAR – an efficient, broad-coverage, principle-based parser. In *Proceedings of COLING–94*, 1994.

[31] 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.

[32] D. Menemenlis and M. Chechelnitsky. Error estimates for an ocean general circulation model from altimeter and acoustic tomography data. *To appear in Monthly Weather Review*, 1999.

[33] Cleve Moler. Cleve's Corner - Why there isn't a parallel MATLAB. MATLAB News and Notes, September 1995.

[34] Greg Morrow and Robert van de Geijn. A parallel linear algebra server for matlab-like environments. In *Proceedings of SC98*, 1998.

[35] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming.* O'Reilly and Associates, 1996.

[36] Cherri M. Pancake and Donna Bergmark. Do parallel languages respond to the needs of scientific programmers? *IEEE Computer*, 23(12):13–23, December 1990.

[37] Michael J. Quinn, Alexey Malishevsky, and Nagajagadeswar Seelam. Otter: Bridging the gap between MATLAB and ScaLAPACK. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, August 1998.

[38] Michael J. Quinn, Alexey Malishevsky, Nagajagadeswar Seelam, and Yan Zhao. Preliminary results from a parallel matlab compiler. In *Proceedings of the 12th International Parallel Processing Symposium*, March 1998.

[39] L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. FALCON: An environment for the development of scientific libraries and applications. In *Proceedings of KBUP'95 - First International Workshop on Knowledge-Based Systems for the (re)Use of Program Libraries*, November 1995.

[40] Luiz De Rose and David Padua. A MATLAB to FORTRAN 90 translator and its effectiveness. In *Proceedings of the 10th ACM International Conference on Supercomputing - ICS'96*, May 1996.

[41] Gerald Salton, editor. *The SMART Retrieval System: Experiments in Automatic Document Processing.* Prentice-Hall, 1971.

[42] Barry Smith, Petter Bjørstad, and William Gropp. *Domain Decompisition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 1996.

[43] Anne E. Trefethen, Vijay S. Menon, Chi-Chao Chang, Gregorz J. Czajkowski, Chris Myers, and Lloyd N. Trefethen. MultiMATLAB: MATLAB on Multiple Processors. Technical Report CTC96TR293, Cornell Theory Center, 1996.

[44] M. A. Turk and A. P. Pentland. Face recognition using eigenfaces. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 586–591, 1991.