The Parallel Problems Server: A Client-Server Model for Interactive Large Scale Scientific Computation

Parry Husbands[1] and Charles Isbell[2]

[1] Laboratory for Computer Science, MIT, Cambridge MA 02139 USA
parry@supertech.lcs.mit.edu
[2] Artificial Intelligence Laboratory, MIT, Cambridge MA 02139 USA
isbell@ai.mit.edu

Abstract. Applying fast scientific computing algorithms to large problems presents a difficult engineering problem. We describe a novel architecture for addressing this problem that uses a robust client-server model for interactive large-scale linear algebra computation.
We discuss competing approaches and demonstrate the relative strengths of our approach. By way of example, we describe MITMatlab, a powerful transparent client interface to the linear algebra server. With MITMatlab, it is now straightforward to implement full-blown algorithms intended to work on very large problems while still using the powerful interactive and visualization tools that Matlab provides. We also examine the efficiency of our model by timing selected operations and comparing them to commonly used approaches.

1 Introduction

We describe a novel architecture for a "linear algebra server" that operates on very large matrices. Matrices are created by the server and distributed across many machines or processors. Operations take place automatically in parallel. The server includes a general communication interface to clients and is extensible via a robust package system.

We are motivated by three observations. First, many widely-used algorithms in machine learning, differential equations, simulation, etc. can be realized as operations on matrices. Second, it is vital to be able to test new ideas quickly in an interactive setting. Finally, algorithms that appear promising on small data sets can fail on large problems and it would be helpful to have a tool that easily enables experimentation on large problems.

Common approaches suffer from several difficulties. Interactive prototyping environments such as Mathematica, Maple, Octave, and Matlab exist; however, they often fail to work well on large problems. Linear algebra libraries designed to work on large problems abound; however, they involve steep learning curves. Further they are typically not interactive, requiring that applications be written in a compiled language, such as C++ or Fortran. This is a burden for users who simply want a library's functionality and for programmers who wish to extend it.

We address these problems directly. Like standard libraries, our system encapsulates basic functionality; however, by modeling the system as a server, we allow for on-the-fly interaction with arbitrary user interfaces. Further, the server is a self-contained application, so we are able to extend it at run-time.

In this paper, we show that our model opens several possibilities. We briefly describe standard approaches in Section 2 before describing the Parallel Problems Server itself in Section 3. We detail its architecture, focusing on its extensibility. Section 4 describes MITMatlab, a system that enables users to compute interactively with very large data sets directly from within Matlab. We then report on the results of some performance experiments in Section 5. Finally, we conclude, discussing further extensions to the system.

2 Standard Approaches

2.1 Linear Algebra Libraries

For many compute-intensive tasks, the best way to maximize performance is to use a library. For example, optimized versions of LAPACK [1] exist that outperform similar code written in a high-level programming language (thanks primarily to native implementations of the BLAS). For distributed memory architectures, vendor-optimized libraries (e.g. Sun's S3L and IBM's ESSL) coexist with public domain offerings such as ScaLAPACK [5], PARPACK [11] and Petsc [4][9].

Each of these libraries has its own idiosyncratic interface and assumptions about the types and distributions of data allowed. It is often a major programming effort to incorporate library routines into an application.

2.2 Interactive Systems

The power of prototyping systems like Maple, Matlab, Mathematica and Octave is that they are interactive. It is straightforward for both seasoned programmers and relatively naive users to develop algorithms and to visualize results from such algorithms. Unfortunately, while these tools work well for small problems, they are often inadequate for production-level data.

There have been many attempts to extend prototyping tools in order to make them work in parallel with large data sets. Here, we focus on systems that add parallel features to Matlab, a widely-used scientific computing tool.

Both MultiMatlab from Cornell University [13] and the Parallel Toolbox for Matlab from Wake Forest University [10], make it possible to manage Matlab processes on different machines. Matlab is extended to include *send*, *receive* and *collective* operations so that separate Matlab processes can communicate. In short, these approaches implement traditional message passing with Matlab as the implementation language.

Compilers for Matlab are also an active area. Both the CONLAB system from the University of Umeå [7] and the FALCON environment from the University of Illinois at Urbana-Champaign [3][12] translate Matlab-like languages into intermediate languages for which high performance compilers exist. For example, FALCON compiles Matlab to Fortran 90 and pC++. Sophisticated analyses of the Matlab source are performed so that efficient target code is generated.

Both of these approaches have merits; however, it is our claim that they do not adequately address the issues we have raised. The former approach is too involved for the naive user and the latter approach sacrifices direct interaction with the computation and includes an edit-compile-run cycle that increases development time.

3 The Parallel Problems Server

The Parallel Problems Server (PPServer) combines many aspects of the approaches we have described so far. Like standard linear algebra packages, the PPServer neatly encapsulates basic functionality; however, because it is a server with a general communication protocol, interaction with arbitrary programs (with their own user interfaces) is possible. Also, the server implements a robust protocol for accessing compiled libraries. Thus, extending the functionality of the PPServer is a simple, modular task.

3.1 The Client-Server Model

The client-server model is ubiquitous. There are HTTP servers that allow access to data via the World Wide Web and database servers that admit access to specially indexed data. Because these servers implement robust protocols for communicating the information they provide, it is possible to build useful clients, such as web browsers.

We believe that this model is also a useful one for scientific computation. First, there is no need to force a client to operate in parallel by endowing it with communication primitives; rather, such communication remains implicit. As a result, the user is not responsible for managing data among various processes. The user simply issues the client's standard commands; these are then transparently executed on multiple machines.

Secondly, there is no need to use the client as the computational engine. While this has the possible short-term disadvantage of the server's functionality being different than the client's, we gain extremely high performance. We are free to use the fastest distributed memory implementations of the algorithms that we need. Furthermore, we are not required to use the client's data representation. For example, Matlab uses double precision numbers. For the very large operations that concern us, it often preferable to use single precision, gaining significant time and space advantages when accuracy is not a concern.

A high-level view of our implementation of the PPServer is shown in Figure 1. Clients make requests of the server. Data are created in a distributed fashion and managed among worker processes, which may live on different machines. Currently we support row and column distributed dense arrays, column distributed sparse arrays, and replicated arrays in single precision. Communication and synchronization among the workers is accomplished using the MPI [8] message passing library. This is a standard library available on a wide range of platforms; it is currently the most portable way to develop applications on distributed memory computers.

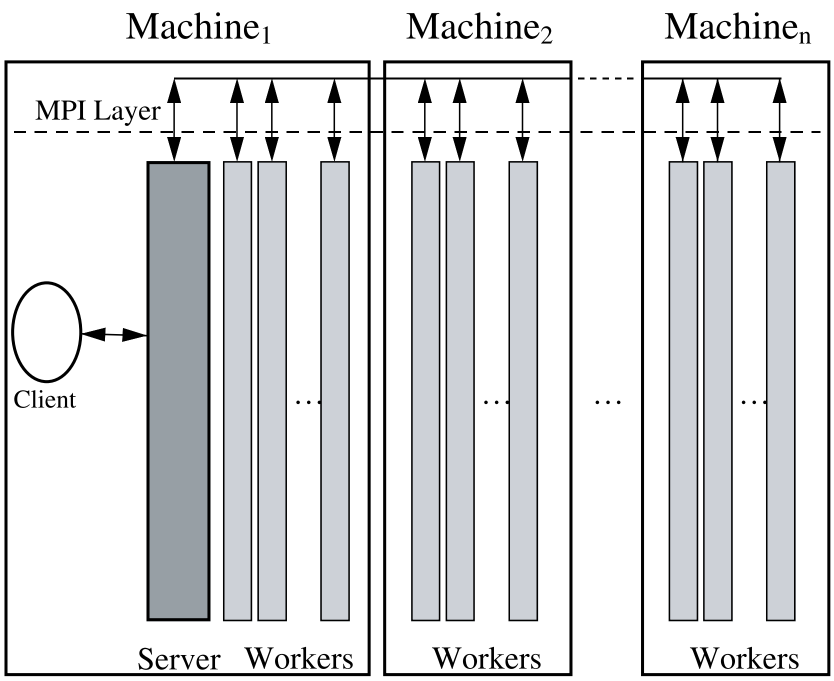

Fig. 1. The General Organization of the Parallel Problems Server. The server process provides an interface to any client that implements its communication protocol.

3.2 Communication and Extensibility

We use the client-server model in two ways. First, there is a protocol for communicating with clients. Just as importantly, there is a separate plug-in architecture that allows for straightforward run-time extensibility of the PPServer.

The Client Interface While we believe that servers are crucial, they remain only academic oddities without useful clients. HTTP servers are useful but they are much more useful when powerful browsers exist. Therefore, it is important that the client interface be simple to use but powerful enough to allow for arbitrary operations.

The PPServer uses standard Unix sockets for client communication. The protocol is straightforward. A client sends a request, consisting of a command and arguments. A command is a string, naming a function. Functions may request data or the loading, saving, or creating of data. Furthermore, they may require that specific operations be performed on already existing data or that library extensions to be included with the server. Arguments are lists of characters, integers and real numbers. Once a command has been completed, it is acknowledged with a message from the server that includes any errors and returned values.

A C++ library (and source) is provided that implements this protocol, including automatic conversion between standard C/C++-style data types and a form suitable for transmission to/from the server. Clients need only provide a suitable wrapper for these functions.

The Server Interface The PPServer is extensible (see Figure 2). It includes a robust function interface using C++ objects. New functions are defined using this interface. These new functions are compiled into dynamically loadable libraries, dubbed "packages" and loaded on demand. Each package is its own name space, so new functions can be loaded "on top" of others, hiding functions of the same name in other packages. Like the PPServer itself, package functions use MPI. These functions enjoy access to the basic functionality of the Server, including direct access to data and the ability to execute all the same commands that are available to clients, including those in other packages.

Figure 3 shows the code for a sample package. It contains one function `sumall` that sums the elements of a distributed matrix. This example shows the mechanisms for extracting input arguments, accessing the elements of the matrix, and returning results to the client. With only a handful of exceptions, all current server functionality is written in this way.

We have used the PPServer as the core of several applications, implementing packages that provide access to ARPACK, SCALAPACK and S3L, Sun's optimized version of SCALAPACK. The functions in the packages are merely short wrappers for the underlying functions provided by the libraries.

Portability The use of standard C++ and MPI has allowed us to develop a system that is highly portable. Although the PPServer was originally developed on a network of symmetric multiprocessors from Sun Microsystems, we have been able to port it to a cluster of SMPs from Digital Equipment Corporation with minimal effort. We are currently working on a port to Pentium-driven Linux systems.

3.3 Other Client-Server Models

There have been previous library systems that implement a similar model. Both RCS [2] and Netsolve [6] act as fast back-ends for slower clients. In their model,

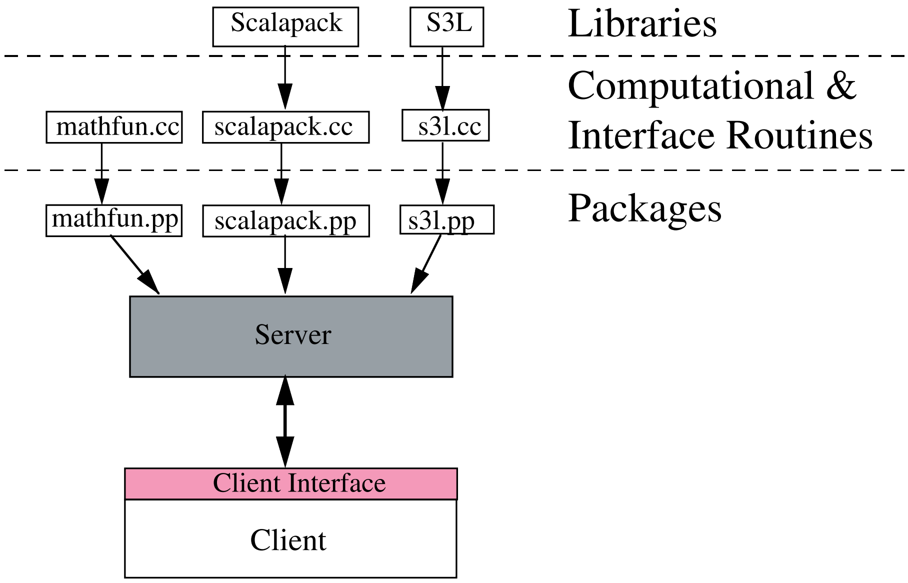

Fig. 2. Extending the PPServer. A client communicates with the PPServer using a simple command-argument protocol. The Server itself uses a "package" mechanism to implement all but its most basic functions. New functionality can be added to the PPServer and managed in a reasonable way. (S3L is Sun's optimized version of some ScaLAPACK routines)

```
void sumall(PPServer &theServer, PPArgList &inArgs, PPArgList &outArgs)
{
  // Get the matrix identifier that was passed in
  PPMatrixID srcID=*(inArgs[0]);

  // Make sure that we're passing in a dense matrix
  if(!theServer.isDense(srcID)) {
    // Return the corresponding error
    outArgs.addError(BADINPUTARGS,"Expecting a Dense Matrix");
    outArgs.add(0);
    return;
  }

  // Get a pointer to the actual matrix
  PPDenseMatrix *src = (PPDenseMatrix *) theServer.getData(srcID);
  float sum=0, answer;

  // Find the local sum of all of the elements
  for(int i=0;i < src->numRows();i++)
    for(int j=0;j < src->numCols();j++)
      sum+=src->get(i,j);

  // Add the local sums to find the global sum
  MPI_AllReduce(&sum,&answer,1,MPI_FLOAT,MPI_SUM,MPI_COMM_WORLD);

  // Return an error code
  outArgs.addNoError();

  // Return the result to the client
  outArgs.add(answer);
}

// Register this function to the server
extern "C" PPError ppinitialize(PPServer &theServer);
PPError ppinitialize(PPServer &theServer)
{
  theServer.addPPFunction("sumall",sumall);
  return(NOERR);
}
```

Fig. 3. A Sample Server Extension. This code is essentially complete other than a few header files

clients issue requests, arguments are communicated to the remote machine and results sent back. Clients have been developed for Netsolve using both Matlab and Java.

Our approach to this problem is different in many respects. Our clients 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 (see Figure 4 for an example). 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.

We also support transparent access to server data from clients. As we shall see below, given a sufficiently powerful client, PPServer variables can be created remotely but still be treated like local variables.

Both Netsolve and RCS assume that the routines that perform needed computation have already been written. Through our package system we support on-the-fly creation of parallel functions. Thus, the server is a meeting place for both data and algorithms.

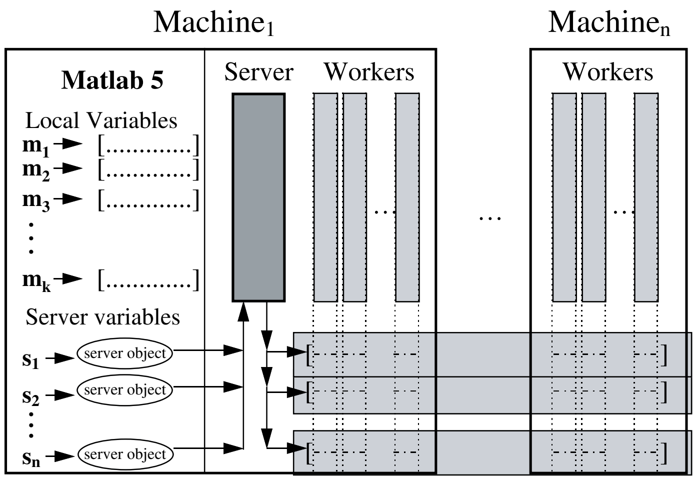

Fig. 4. MITMatlab Variables. Use of the PPServer by Matlab is almost completely transparent. PPServer variables remain tied to the server itself while Matlab receives "handles" to the data. Using Matlab scripts and Matlab's object and typing mechanisms, functions using PPServer variables invoke PPServer commands implicitly.

4 MITMatlab

Using the client interface, we have implemented a Matlab front end, called MITMatlab. At present, we can process gigabyte-sized sparse and dense matrices "within" Matlab, admitting many of Matlab's operations transparently (see Figure 5). By using a client as the user interface, we take advantage of whatever interactive mechanisms are available to it. In Matlab's case, we inherit a host of parsing capabilities, a scripting language and a host of powerful visualization tools.

For example, we have implemented BRAZIL, a text retrieval system for large databases. BRAZIL can process queries on a million documents comprised of hundreds of thousands of different words. Because of Matlab's scripting capabilities, little functionality had to be added to the server directly; rather, most of BRAZIL was "written" in Matlab.

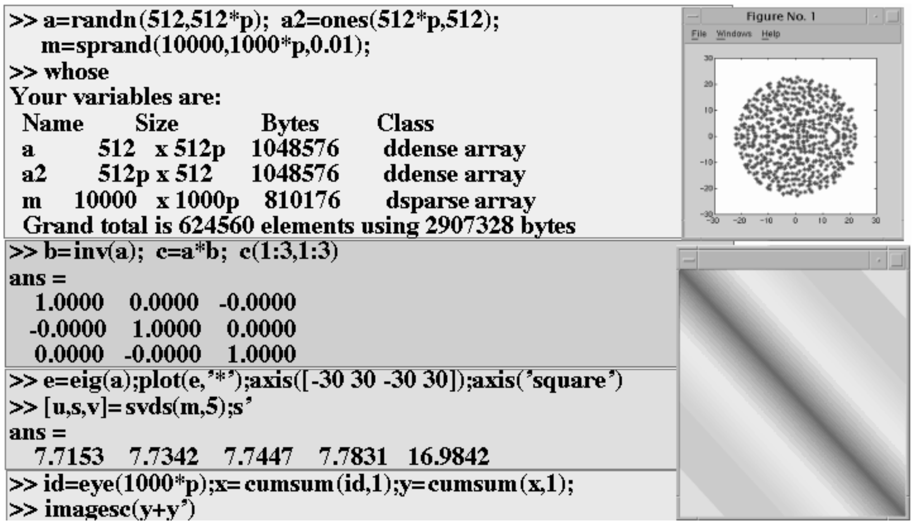

Fig. 5. A Screen Dump of a Partial MITMatlab Session. Large matrices are created on the PPServer through special constructors. Multiplication and other matrix operations proceed normally.

5 Performance

In this section we present results demonstrating the performance of the PPServer. We begin with experiments comparing the efficiency of individual operations in Matlab with the same operations using MITMatlab. We conclude with a case study of a computation that requires more than a single individual operation. We compare the performance impact of implementing a short program in Matlab, directly on the PPServer, and using optimized Fortran.

5.1 Individual Operations

We categorize individual operations into two broad classes, according to the amount of computation that is performed relative to the overhead involved in communicating with the PPServer. For *fine grained* operations, most of the time is spent communicating with the server. A typical fine grained task would involve accessing or setting an individual element of a matrix. *Coarse grained* operations include functions such as matrix multiplication, singular value decompositions, and eigenvalue computations where the majority of the time is spent computing instead of communicating input and output arguments with the server.

Below we assess MITMatlab's performance on both kinds of operations. Experiments were performed on a network of Digital AlphaServer 4/4100s connected with Memory Channel.

Fine Grained Operations These operations are understandably slow. For example, in order to access an individual element, the client sends a message to the server specifying the matrix and location, the server locates the desired element among its worker processes, and then finally sends the result back to the client.

MITMatlab cannot compete with the local function calls that Matlab uses for these operations. For example, accessing an element in Matlab only takes 139 microseconds on average, while on a request from the server such can take 2.8 milliseconds. This result can be entirely explained by the overhead involved in communicating with the server; a simple "ping" operation where MITMatlab asks the PPServer for nothing more than an empty reply takes 2 milliseconds.

Coarse Grained Operations For coarse grained operations, the overhead of client/server communication is only a small fraction of the computation to be performed.

Table 1 shows the performance of dense matrix multiplication using Matlab and MITMatlab. Large performance gains result from the parallelism obtained by using the server; however, even in the case where the server is only using a single processor, it gains significantly over Matlab. This is due in part because the PPServer can use an optimized version of the BLAS. This illustrates one of the advantages of our model. We can use the fastest operations available on a given platform.

Using PARPACK, MITMatlab also shows superior performance in computing singular value decompositions on sparse matrices (see Table 2).

It is worth noting that Matlab's operations were performed in double precision while the PPServer's used single precision. While this clearly has an effect on performance, we do not believe that it can account for the great performance difference between the two systems.

Table 1. Matrix multiplication performance of the MITMatlab on p processors. Time are in seconds. Here "$p = 3 + 3$" means 6 processors divided between two machines.

	Matrix Size N		
	1Kx1K	2Kx2K	4Kx4K
Matlab	41.1	267.1	2814.9
MITMatlab			
with $p = 1$	5.5	45.1	357.9
$p = 2$	2.8	21.5	175.6
$p = 4$	3.9	12.9	94.7
$p = 3 + 3$	1.4	14.4	64.5

Table 2. SVD performance of MITMatlab on p processors using PARPACK. These tests found the first 5 singular triplets of a random 10K by 10K sparse matrix with approximately 1, 2, and 4 million nonzero elements. Matlab failed to complete the computation in a reasonable amount of time. Times are in seconds.

Processors	Nonzeros		
used	1M	2M	4M
2	136.8	169.2	433.5
4	88.8	91.9	241.0
$3 + 3$	75.2	78.8	168.6

Discussion These results make it clear what types of tasks are best performed on the server. Computations that can be described as a series of coarse grained operations on large matrices fare very well. By contrast, those that use many fine grained operations may be slower than Matlab. Such tasks should be recoded to use coarse grained operations if possible, or incorporated directly into the server via the package system. Note that on many tasks that involve computation on large matrices, fine grained operations occupy a very small amount of time and so the advantages that we gain using the server are not lost.

5.2 Executing Programs

Figure 6 shows the Matlab function that we used for this experiment. It performs a matrix-vector multiplication and a vector addition in a loop. Table 3 shows the results when the function is executed: 1) in Matlab, 2) in Matlab with server operations, 3) directly on the server through a package, and 4) in Fortran. Experiments were performed using a Sun E5000 with 8 processors. The Fortran code used Sun's optimized version of LAPACK.

The native Fortran version is the fastest; however, the PPServer package version did not incur a substantial performance penalty. The interpreted MIT-Matlab version, while still 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.

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

function X=testfun(A,x0,Q,n)

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

Fig. 6. Matlab code for the program test. The Matlab version that used server operations included some garbage collection primitives in the loop.

Table 3. The performance of the various implementations of the program test. Although Matlab takes some advantage of multiple processors in the SMP we list it in the $p = 1$ row.

	Time (sec)			
Processors Used	Fortran	Server Package	Matlab with Server	Matlab
1	3.07			49.93
2	1.61	1.92	2.43	
4	0.90	1.02	1.49	
6	0.62	0.78	1.26	
8	0.55	0.67	1.84	

6 Conclusions

Applying fast scientific computing algorithms to large everyday problems represents a major engineering effort. We believe that a client-server architecture provides a robust approach that makes this problem much more manageable.

We have shown that we can create tools that allow easy interactive access to large matrices. With MITMatlab, researchers can use Matlab as more than a prototyping engine restricted to toy problems. It is now possible to implement full-blown algorithms intended to work on very large problems without sacrificing interactive power. MITMatlab has been used successfully in a graduate course in parallel scientific computing. Students have implemented algorithms from areas including genetic algorithms and computer graphics. Packages encapsulating various machine learning techniques, including gradient-based search methods, have been incorporated as well.

Work on the PPServer continues. Naturally, we intend to incorporate more standard libraries as packages. We also intend to implement out-of-core algorithms for extremely large problems, as well implement interfaces to other clients, such as Java-enabled browsers. Finally, we wish to use the PPServer as real tool for understanding the role of interactivity in supercomputing.

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

References

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Criz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. Siam Publications, Philadelphia, 1995.
2. P. Arbenz, W. Gander, and M. Oettli. The Remote Computation System. Technical Report 245, ETH Zurich, 1996.
3. Falcon Group at the University of Illinois at Urbana-Champaign. The Falcon Project. http://www.csrd.uiuc.edu/falcon/falcon.html.
4. 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.
5. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhilon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. Scalapack Users' Guide. http://www.netlib.org/scalapack/slug/scalapack_slug.html, May 1997.
6. Henri Casanova and Jack Dongarra. Netsolve: A Network Server for Solving Computational Science Problems. In *Proceedings of SuperComputing 1996*, 1996.
7. Peter Drakenberg, Peter Jacobson, and Bo Kagstrom. 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.
8. William Gropp, Ewing Lusk, and Anthong Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.
9. PETSc Group. PETSc - the Portable, Extensible Toolkit for Scientific Computation. http://www.mcs.anl.gov/home/gropp/petsc.html.
10. J. Hollingsworth, K. Liu, and P. Pauca. *Parallel Toolbox for MATLAB PT v. 1.00: Manual and Reference Pages*. Wake Forest University, 1996.
11. 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.

12. 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.
13. Anne E. Trefethen, Vijay S. Menon, Chi-Chao Chang, Gregorz J. Czajkowski, Chris Myers, and Lloyd N. Trefethen. MultiMATLAB: MATLAB on Multiple Processors. http://www.cs.cornell.edu/Info/People/lnt/multimatlab.html, 1996.