Software for Parallel Computing: the LAM Implementation of MPI

By: Christopher A. Swann

Abstract:

Many econometric problems can benefit from the application of parallel computing techniques, and recent advances in hardware and software have made such application feasible. There are a number of freely available software libraries that make it possible to write message passing parallel programs using personal computers or Unix workstations. This review discusses one of these—the LAM (Local Area Multiprocessor) implementation of MPI (the Message Passing Interface).

Keywords: Econometrics | Parallel Computing | LAM | MPI

Article:

1 INTRODUCTION

Parallel computing involves using multiple central processing units (CPUs) to solve one computational problem. For many years researchers have used parallel computing when working on computationally challenging problems such as nuclear weapons simulation. More recently, economists have applied parallel computing to large-scale matrix problems (Nagurney and Eydeland, 1992; Chabini *et al.*,1994), the solution of non-linear dynamic models (Coleman, 1993), and simulation (Liu and Rubin, 1996). Nagurney (1996) provides an overview of the application of parallel computing to economics.

In addition to these applications, parallel computing can be used to reduce the time it takes to solve estimation problems in econometrics. While this review will not discuss the details of the estimation process, a simple illustration is helpful.[1] Consider estimating the parameters of a maximum likelihood model with data on 4800 independent observations. Suppose that a four processor Unix workstation is available to solve the problem, and let T_s be the amount of time it takes to solve the problem in serial. If the data can be broken up into 1200 person subsets so that all four processors are used to solve the problem at the same time, the time it takes to find a solution may be approximately one fourth of T_s.

There are a number of ways to take advantage of parallelism. First, some Fortran compilers have a compilation switch that will automatically parallelize loops under certain conditions. Second, High Performance Fortran is an extension to the Fortran 90 specification that allows data to be distributed across multiple processors (Koelbel *et al.*, 1994). Finally, message-passing software allows one to send messages from one processor (or computer) to another. As one might expect, there is a tradeoff between the cost and benefit of each of these options. For example, the cost of writing a message-passing parallel program is higher than the cost of flipping a switch at compile time, but the payoff can be much greater.

Because of its greater potential and flexibility, this review focuses on installing and using a message-passing library. I provide a detailed discussion of one of the message-passing libraries: the LAM (Local Area Multicomputer) implementation of MPI (The Message Passing Interface). LAM was originally developed at the Ohio Supercomputing Center at Ohio State University and is now being developed at the Laboratory for Scientific Computing at the University of Notre Dame (http://www.mpi.nd.edu/lam/). It has been tested using single and multiprocessor PCs running Linux and OpenBSD and single and multiprocessor Unix workstations running AIX, HP-UX, IRIX, and Solaris.[2] The next section provides some background on MPI including a simple program. Section 3 discusses how to obtain, configure, and install LAM. Section 4 details how to start LAM; how to compile, run, and debug parallel jobs; and how to terminate jobs and shut down LAM. Finally, Section 5 presents timings from a simple maximum likelihood problem that demonstrate the potential time saving from parallel computing.

2 THE MESSAGE PASSING INTERFACE

2.1 Background

In 1992, a group of researchers from industry and academia formed the Message Passing Interface Forum (MPIF) with the intent of developing a standard for portable message-passing parallel programs. At the time the MPIF convened, there were a number of options for writing parallel computer code. Proprietary programming languages were available for specific parallel computers; Parallel Virtual Machine (PVM) was a widely used, freely available message-passing library; and there were a number of other somewhat less widely used free libraries.[3] Each of these languages and libraries had similar functionality, but they were incompatible with each other, and a goal of the MPIF was to make it easier to write portable message-passing programs.

The MPI-1 standard, available at http://www.mpi-forum.org/, was released by the MPIF in June 1994 and essentially contains a superset of the features of the existing parallel libraries and languages. The MPIF continued to meet after the adoption of the MPI-1 standard, and the MPI-2 standard was released in 1997. The new standard incorporates recent advances such as parallel I/O into the specification. Gropp, Lusk, and Skjellum (1999) provide a helpful introduction to

MPI-1; Gropp, Lusk, and Thakur (1999) discuss the new features in MPI-2; and the MPI web site http://WWW.ERC.MsState.Edu/labs/hpcl/projects/mpi/ lists a number of other books and tutorials.

There are a number of different implementations of MPI. Some of these are developed by computer manufacturers and are tuned for the company's specific hardware. Others are freely available and run on many different hardware platforms. Two of these are the MPICH implementation developed at Mississippi State University and Argonne National Laboratory (http://www.mcs.anl.gov/mpi/mpich/) and LAM. LAM was chosen for this review because it is straightforward to install and use, because it has a graphical user interface, XMPI, which is not discussed here because of space limitations, and because there is an active e-mail list where one can turn for help.[4]

2.2 Example MPI Program

Although the focus of the review is on the software used for parallel programming rather than the actual syntax of MPI itself, it may be helpful to begin by examining a simple message-passing MPI program. The example program uses the ‘master/slave’ model. In this model, the parallel job consists of multiple copies of the same executable. When the job is run, each copy is assigned a unique identification number (its ‘rank’), and the work within the program is divided up based on the rank of the process. In the following example, the process with *my_rank=0* is the master, and the process with *my_rank=1* is the slave. The slave passes the number 10 to the master, and the master prints the number.

- program parallel
- implicit none
- include ‘mpif.h’
- integer ier, my_rank, status (MPI_STATUS_SIZE), np, number
- call MPI _INIT (ier)
- call MPI _COMM_SIZE(MPI_COMM_WORLD,np,ier)
- call MPI_COMM_RANK(MPI_COMM_WORLD,my_rank,ier)
- if (my_rank.eq. 0) then
- call MPI_RECV(number,1,MPI_INTEGER,1,0,MPI_COMM_WORLD,
- & status,ier)
- write (6,*) ‘Number is’,number

- else if (my_rank.eq. 1) then
- call MPI_SEND (10, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, ier)
- end if
- call MPI_FINALIZE (ier)
- stop
- end

The first three calls to MPI subroutines initialize MPI for this particular run, find out the number of copies of the program (given when the parallel job is started), and find out the rank of the process in the parallel job, respectively. If the rank is 1, the process calls MPI_SEND to send a message to the master. The first four arguments are what is to be sent (in this case the number 10, but it could also be an array), the number of elements (1), the MPI data-type (MPI_INTEGER), the rank of the receiving process (0 in this case since the number is sent to the master), and three other arguments which are not important in the present discussion.

The master calls the subroutine MPI_RECV to receive the message. It stores the message in a buffer called number. The second argument is again the number of elements (1); the third is the data-type (MPI_INTEGER); and the fourth argument is the rank of the sending process (1). The last four arguments are not important in the present discussion. After receiving the message, the master prints the value of number. Finally, after the work is finished, both the master and the slave call MPI_FINALIZE to ‘tidy up’.

Perhaps surprisingly, these six subroutine calls are all of the MPI that one needs to know in order to write message-passing parallel programs to solve maximum likelihood problems.

3 OBTAINING AND INSTALLING LAM

3.1 Obtaining the Software

LAM is available at no cost from http://www.mpi.nd.edu/lam/download/. At the present time, the latest stable version of LAM is 6.5. The source code is available in compressed tar files. If gunzip is available, download the file lam-6.5.tar.gz to, for example, /usr/local/src. The commands gunzip lam-6.5.tar.gz and tar xvf lam-6.5.tar create a source tree at /usr/local/src/lam-6.5. Basic installation instructions are available in LAM Team (2001a) and in the README, INSTALL, and RELEASE_NOTES files in the source directory.

3.2 Building LAM

A configure script must be run to determine machine-specific settings and to ensure that the necessary system files and tools are available. To use the default configuration values, change directories to /usr/local/src/lam-6.5 and type./configure.

There are a number of reasons to choose non-default configure script options. For example, by default LAM uses rsh to remotely access other computers even though rsh is insecure. This option (and many more) can be changed on the command line. For example, the appropriate command to configure LAM using ssh for remote access is ./configure --with-rsh="ssh -x".

After the configure script runs successfully, installing LAM is straightforward. While in the directory /usr/local/src/lam-6.5, type make at the command line to compile the source code. Assuming there are no errors, typing make install copies the executables and other supporting files to the installation directory (by default /usr/local/lam).

If there are error messages when the configure script runs (or when compiling LAM or running parallel programs), there are a number of places to obtain help. The LAM FAQ (LAM Team, 2001b) has some discussion of configuration and is located athttp://www.mpi.nd.edu/lam/faq/. If the FAQ is not helpful, one can search the LAM mail archive (http://www.mpi.nd.edu/ MailArchives/lam/). Failing that, submitting a question to the e-mail list will likely result in an answer from one of the developers or from another user.

3.3 Considerations for Network Installation

While LAM can run parallel jobs on a single multiprocessor computer, it is also frequently used to create a parallel computer out of a cluster of networked workstations or PCs.[5] Installing LAM on a cluster is no harder than installing it on a single machine. However, it is important to be aware of a few issues before beginning. Most importantly, using LAM on a cluster is easiest if one of the machines is an NFS server from which home directories will be exported to other machines. If the home directories are not NFS-mounted, then the executable may have to be manually copied to each machine. Ideally, the installation directory for LAM will also be NFS exported. If this is not possible, it is best if the installation directory is the same on each of the machines. Finally, as noted above, ssh may be preferred over rsh for security reasons.

4 RUNNING PARALLEL PROGRAMS

There are three steps involved in running a parallel job: 'booting' LAM, compiling the Fortran or C code, and submitting the executable. Before any of these steps can be completed, however, the directory where the LAM executables are located should be added to the user's path. The default location for the executables is /usr/local/lam/bin.

4.1 Booting LAM

LAM uses something called a boot schema (or hosts file) to define the machines that make up the parallel computer. The default boot schema is called lam-bhost.def, and it is located in /usr/local/lam/etc.

The default boot schema contains only the local machine. While this is appropriate if LAM will be run on a single multiprocessor, it is not correct if the parallel computer will consist of a cluster of machines. To understand what a more typical boot schema file looks like, suppose that the multicomputer consists of three machines named itchy, scratchy, and krusty. In this case the boot schema would have three lines:

- itchy cpu=2
- scratchy
- krusty

where the option cpu=2 indicates that itchy has two processors. It is also acceptable to include a separate line for each processor rather than the number of processors (i.e. the hosts file could have two lines for itchy).

The command lamboot boots the computer described in the default boot schema. The option −v provides more verbose output, and if there is a problem booting, the option −d provides detailed debugging output. A non-default hosts file may be specified by including the file name on the command line: lamboot ./config1 boots LAM using the boot schema config1 located in the current directory.

4.2 Compiling Programs

Compiling a Fortran program to use LAM is straightforward. LAM creates compiler 'wrappers' to compile Fortran, C, and C++ programs called mpif77, mpicc, and mpiCC, respectively, that automatically link the necessary LAM libraries. The wrapper compilers accept the same flags as the underlying Fortran, C, or C++ compilers.

Assuming the example program above is called parallel.f, the command mpif77 parallel.f will create an executable called a.out from the Fortran source file parallel.f.

4.3 Running a Parallel Job

The command to run a job is mpirun. The syntax of mpirun makes it straightforward to run a job on all (or a subset) of the available nodes or CPUs, but also allows for complex jobs. The LAM FAQ contains discussion of the syntax used to run more complex jobs. In some cases these jobs require an 'application schema' which is similar to a boot schema and defines which executable(s) run on which node(s).

In the most common case, the parallel job consists of multiple copies of the same executable. The syntax for running *R* copies of the executable a.out is mpirun -np *R* a.out. The processes are assigned to computers or processors according to the ordering in the hosts file. For example, given the form of the hosts file above, mpirun -np 2 a.out runs a parallel job with two processes on itchy while mpirun -np 3 a.out runs a parallel job with two processes on itchy and one on scratchy. If the hosts file did not indicate that itchy is a multiprocessor, the latter command would run a parallel job with one process on each of the three machines. Finally, Version 6.5 introduces a shortcut for running jobs on multiprocessor computers. The command mpirun C a.out runs the executable a.out on all available processors. Thus, given the hosts file above, mpirun C a.out runs four copies of the executable a.out—two on itchy and one on each of scratchy and krusty.

4.4 Aborting Jobs and Stopping LAM

In the event that a job must be terminated (or to 'clean up' after a job that has died), the command lamclean can be used to terminate all user processes related to the job and to remove any temporary files that may exist. LAM itself is stopped by using the command lamhalt.

4.5 Debugging

Until recently, debugging a program in LAM meant inserting print statements liberally throughout the program because it was not possible to run a parallel job under a debugger. Naturally this added greatly to the difficulty of debugging complicated programs. Beginning with Version 6.3.2, it is now possible to use mpirun to launch non-MPI programs such as debuggers.

To debug a parallel program, a short, executable script must be created. This script will be run by mpirun, and it will launch the debugger. Consider the following script (adapted from the LAM FAQ):

- # !/bin/csh -f
- xterm -e dbx $ *
- exit 0

The first line identifies the script as a csh script; the second line opens an xterm running dbx; and the third is the exit code.

The syntax for mpirun looks a little different when running the debugger. Let the script be called run_dbx.csh. The syntax to debug the executable a.out on 2 nodes is

- mpirun -np 2 -x DISPLAY run_dbx.csh a.out where -x DISPLAY is the syntax to export the environment variable DISPLAY to the remote nodes before executing the program. This mpirun command runs two copies of run_dbx.csh each of which opens an X-terminal running dbx to debug the executable a.out.

It is best to limit the number of processes to two when debugging. It becomes a challenge to keep track of (and switch among) three or more separate windows, and two processes are usually all that is necessary to debug message-passing problems.

Debugging a parallel program is similar to debugging a serial program. However, there are a number of places where the processes must synchronize before any of them can proceed. For example, each process must call MPI_INIT before *any* of them can execute code beyond MPI_INIT. A typical strategy is to run the above script with two debugging windows and step through the initial calls to MPI_INIT, MPI_COMM_WORLD, and MPI_COMM_RANK in each debugging window in order to find out each process's rank. Once each rank is known then breakpoints can be set as usual.

5 EXAMPLE

A simple example demonstrates the time savings that are possible when using MPI. Consider estimating the parameters of a five-choice multinomial logit model using maximum likelihood. For each choice there are five explanatory variables, and there are 4800 independent observations.[6]

The test platform is a Sun E420R server configured with four 450 Mhz processors and four gigabytes of RAM and running Solaris 8 and LAM 6.3.3 (a beta version of 6.5). LAM is compiled with gcc 2.95.2 and uses the sysv rpi for communication, and the Sun Workshop 5.0 Fortran compiler is used to compile the estimation code with the compiler options -fast -xtarget=ultra2. Because the parameters can be estimated in serial in only a few seconds on this machine, this problem does not require parallel computing. To make the ratio of computational work to message passing more similar to what one would expect from a more realistic application, the estimation program is modified so that the choice probability for each person is computed 10,000 times.

Table I presents the run times for a number of different estimation scenarios. The first row of Table I shows that it takes 49 minutes and 48 seconds to estimate the (altered) model in serial. This longer runtime better approximates the more realistic case where message passing will be dominated by the CPU time it takes to compute likelihood values or moment conditions.

Table I. Multinomial logit estimation times

Number of processors	**Time**	**Speedup**
1	49 : 48	—

4[a]	46 : 52	—
2	22 : 58	2.17
3	15 : 21	3.24
4	11 : 38	4.28

a Parallel compiler option turned on.

The Sun Fortran compiler has the ability to parallelize loops automatically. To provide some insight into the effectiveness of such a strategy, the code was recompiled with the -parallel option (and the environment variable PARALLEL was set equal to 4). The second row shows that this change results in a small gain in performance of about 3 minutes.[7]

The remaining lines show the run times for two, three, and four processors using MPI and LAM. In order to estimate the parameters in parallel, the estimation program must be ported to MPI. The pseudo-code listed below shows the basic strategy for performing maximum likelihood in parallel.[8]

```
C ------------ Parallel Pseudo-code ----------------
if (my_rank = 0 ) then
do i = 1, number of slaves
send subset of data to slave
update num_sent
end do
do while (num_received < 4800)
receive likelihood contribution from a slave
update likelihood function and derivatives
update num_received
if (num_sent < 4800) then
send subset of data to the slave that
& just sent likelihood value
update num_sent
```

```
endif
end do
update parameters
else
100 receive subset of data from master
compute likelihood function and derivatives
send results to master
goto 100
endif
```

The master initially sends subsets of the data array to each of the slaves. As it does this, it updates *num_sent*, a counter of the number of observations that have been sent to the slaves. It then waits for the likelihood contribution and derivatives from each of the slaves. When the master receives this information, it updates the global likelihood function, derivatives, and *num_received*, which indicates the number of observations for which a likelihood contribution has been received. If there are more observations to be sent, it sends them to the slave that sent the most recent information. Once all of the observations have been sent and all the likelihood contributions have been received, the parameters are updated. Although not shown here, the process is repeated if the function has not converged and stops if it has. The slaves simply receive the subset of the data array from the master, compute the appropriate likelihood function and derivatives, send the results to the master, and wait for more data.

The third line in Table I shows that when two processors are used with MPI, the run time falls to 22 : 58. Thus, using two processors is actually slightly more than twice as fast as using one processor. A measure of the scalability of a particular application is its ‘Speedup’. Speedup is defined as the serial time divided by the parallel time, and speedups for two, three, and four processors are given in the third column of Table I. While one would typically expect the speedup to be bounded above by the number of processors, Table I shows that the speedup for two processors is greater than two. The greater than linear speedup in the application is most likely caused by better cache and memory usage patterns when the application is spread across multiple processors.[9]

Adding the third and fourth processors continues to dramatically reduce run time. In fact, it is possible to solve the problem over four times as fast using four processors. These improvements are specific to the problem at hand, but it is reasonable to expect that a highly parallel application (such as maximum likelihood or method of moments) that is CPU bound will experience an almost linear speedup on a multiprocessor with no other users. Whether similar speedups can be achieved on a cluster depends on the speed of the networking connections and on the relative speed of the machines in the cluster.

6 CONCLUSION

Parallel computing can be used to achieve significant reductions in the estimation time for maximum likelihood or method of moments problems in econometrics. The Message Passing Interface is the standard message-passing library in use today, and LAM is a freely available implementation of MPI that allows one to use single or multiprocessor PCs or Unix workstations to create a powerful ‘multicomputer’. This review has discussed installing and using LAM and has shown that significant reductions in run time are possible when running estimation code in parallel.

Acknowledgements

The author would like to thank James MacKinnon and Jeffrey Squyres for helpful comments.

REFERENCES

Chabini I, Drissi-Kaitouni O, Florian M. 1994. Parallel implementations of primal and dual algorithms for matrix balancing. In*Computational Techniques for Econometrics and Economic Analysis*, BelsleyD (ed.). Kluwer Academic Publishers: New York.

Coleman WJ II. 1993. Solving nonlinear dynamic models on parallel computers. *Journal of Business & Economic Statistics* **11**:325–330.

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

Gropp W, Lusk E, Skjellum A. 1999. *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (2nd edn). The MIT Press: Cambridge, MA.

Gropp W, Lusk E, Thakur R. 1999. *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press: Cambridge MA.

Koelbel CH, Loveman DB, Schreiber RS, Steele GL Jr, Zosel ME. 1994. *The High Performance Fortran Handbook*. The MIT Press:Cambridge, MA.

LAM Team. 2001a. LAM Version 6.5. University of Notre Dame. http://www.mpi.nd.edu/lam/6.5/.

LAM Team. 2001b. LAM FAQ. University of NotreDame. http://www.mpi.nd.edu/lam/faq/.

Liu C, Rubin DB. 1996. Markov-Normal analysis of iterative simulations before their convergence. *Journal of Econometrics* **75**: 69–78.

Nagurney A. 1996. Parallel computation. In Handbook of Computational Economics (vol. **1**), AmmanHM, KendrickDA, RustJ (eds), Elsevier: Amsterdam.

Nagurney A, Eydeland A. 1992. A splitting equilibration algorithm for the computation of large-scale constrained matrix problems: theoretical analysis and applications. In *Computational Economics and Econometrics*, AmmanH, BelsleyD, PauL (eds). Kluwer Academic Press: New York.

Sterling TL, Salmon J, Becker DJ, Savarese DF. 1999. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. The MIT Press: Cambridge, MA.

- 1 Pseudo-code for estimating the parameters of this model and timings using one, two, three, and four processors are presented in Section 5. Swann ('Maximum likelihood estimation using parallel computing: an introduction to MPI', 2000, unpublished manuscript) provides a more detailed discussion.
- 2 A complete list of supported operating systems can be found at http://www.mpi.nd.edu/lam/6.5/support.php3
- 3 The PVM homepage is located at http://www.epm.ornl.gov/pvm/, and a good summary of PVM can be found in Geist *et al.* (1994).
- 4 While the MPI standard specifies the subroutine calls and the syntax of each call, it leaves the actual method of implementing the functionality up to each developer. Consequently, there can be performance differences among implementations, and it may be beneficial to compare performance under different implementations.
- 5 Sterling *et al.* (1999) provide an excellent overview of how to build PC clusters for parallel computing.
- 6 See Swann (unpublished manuscript, 2000) for information about the data and this multinomial logit problem.
- 7 The compiler option -loopinfo can be used to determine which loops were parallelized and which were not (and why). Using this option shows that very few of the loops were actually parallelized because of compiler concern about dependencies in many loops.
- 8 See Swann (unpublished manuscript, 2000) for a more detailed discussion.
- 9 My thanks to Jeffrey Squyres of the LAM Team for suggesting this explanation