Maximum Likelihood Estimation Using Parallel Computing: An Introduction to MPI

By: Christopher A. Swann

Swann, C.A. (2002). Maximum Likelihood Estimation Using Parallel Computing: An Introduction to MPI. *Computational Economics, 19(2)*, 145-178.

The final publication is available at Springer via http://dx.doi.org/10.1023/A:1015021911216

Abstract:

The computational difficulty of econometric problems has increased dramatically in recent years as econometricians examine more complicated models and utilize more sophisticated estimation techniques. Many problems in econometrics are `embarrassingly parallel' and can take advantage of parallel computing to reduce the wall clock time it takes to solve a problem. In this paper I demonstrate a method that can be used to solve a maximum likelihood problem using the MPI message passing library. The econometric problem is a simple multinomial logit model that does not require parallel computing but illustrates many of the problems one would confront when estimating more complicated models.

Keywords: parallel computing | parallel programming | MPI | maximum likelihood estimation

Article:

1. Introduction

The computational difficulty of econometric problems has increased dramatically in recent years as econometricians estimate more complicated models and utilize more sophisticated estimation techniques. One way to meet the increasing computational requirement is to use a faster single processor computer, but continually pursuing the fastest computer can be very expensive and does not scale well as problem size increases. A second possibility is to break down the computational problem into a number of smaller problems that can be solved simultaneously on less expensive computers utilizing parallel computing methods.

Parallel computing has recently been used in a number of different applications by economists.[1] These include the solution of 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).[2] Parallel computing may also be used to solve estimation problems in econometrics. Many econometric problems involve summing some quantity over a large number of independent observations (e.g., maximum likelihood or the method of moments), and such problems are potentially highly parallelizable.

Despite its potential, and notwithstanding the aforementioned papers, parallel computing has not been widely used by economists. There are at least two reasons for this. First, historically the hardware necessary to utilize parallel techniques was very expensive and not widely available, and second, learning to write parallel programs requires a perhaps significant investment of time and energy. With the advent of inexpensive and powerful personal computers, increasingly fast networking solutions, and freely available message passing libraries, it is now possible to build a relatively inexpensive parallel computer. The purpose of this paper is to address the second issue.

In this paper, I take a very simple econometric problem – a five choice multinomial logit model – and demonstrate one way to solve that problem using the MPI message passing library. The econometric problem is obviously so simple that one would never use parallel processing to solve it, but its simplicity allows it to ‘get out of the way’ of the message passing.

The assertion that this problem is ‘too simple’ for parallel computing raises an important question: When is it worthwhile to write an estimation program to run in parallel rather than serial? Unfortunately there is no universal answer. There are, however, a few guidelines that may be helpful. First, because message passing is slow relative to computation, parallel computing will have a larger impact when the CPU requirements of the program are large relative to the amount of message passing. The algorithm outlined in Section 6 requires that only a small number of messages be passed, and timings reported in Section 8.1.1 show that the parallel performance improves as the problem becomes more CPU intensive. Second, one must balance the trade-off between programming time and run time. Even though theMPI library consists of standard FORTRAN routines and is not a new language, it takes additional time to program in parallel. Consequently, even if the potential parallel solution may have significant CPU utilization relative to communication, a program that only takes two days to run in serial is not likely to be a good candidate to be parallelized. Third, although the main focus in this paper is on the time it takes to find a solution, problems with large memory requirements may benefit if it is possible to decompose the data in such a way that it fits into the physical memory of multiple machines.

The remainder of the paper is organized as follows: the next section introduces MPI, Section 3 introduces the multinomial logit problem, and the data are discussed in Section 4. The serial solution to the problem is discussed in Section 5. Section 6 examines the parallel implementation. Sections 7 and 8 discuss alternative ways to send multiple messages and assorted other issues such as debugging programs, respectively. Finally, Section 9 concludes.

2. An Introduction to MPI

2.1. BACKGROUND ON MESSAGE PASSING LIBRARIES

Parallel computing has been used in computer science for many years and is frequently used to solve ‘grand challenge’ problems such as weather forecasting and nuclear weapon simulations. Initially parallel computers were very specialized machines that required one to write programs specifically for a particular machine. Recently, however, the capability to run parallel code has become much more widely available.

One of the main reasons for the more widespread use of parallel computing is the development of freely available message passing libraries. The first such library that was widely used is called PVM.[3] The introduction of PVM helped make parallel processing more accessible for at least two reasons. First, because PVM is a library of FORTRAN subroutines (and C functions), programmers do not have to learn a new programming language in order to write parallel code. Second, PVM was designed for use in heterogeneous computing environments and runs on wide variety of hardware platforms from massively parallel processors to unix workstations and PCs. This last feature made possible a new class of parallel computer: the network of workstations (NOW). The idea is that many institutions have a (perhaps large) number of workstations that are used only part of the day. PVM allows these machines to be used together as one parallel machine.[4]

One shortcoming of PVM is the lack of an agreed upon standard. PVM is essentially the result of an on-going research project rather than an implementation of an industry standard. This shortcoming led to the formation of the Message Passing Interface Forum and ultimately to the MPI specifiction.[5] Like PVM, MPI is a library of function and subroutine calls. Unlike PVM, there is an official MPI standard, and there are a number of different implementations of MPI. Some of these come from computer manufacturers (e.g., IBM, SUN, and Intel) and are optimized for the manufacturer's particular hardware, and some are freely available (e.g., LAM6 and MPICH7). Like PVM, MPI runs on a wide array of machines ranging from Cray supercomputers to UNIX workstations to PCs running Linux.

2.2. USING MPI

There are a number of steps involved in running a parallel program using MPI. In this section, I discuss a number of basic MPI subroutine calls and illustrate them with short programs and code fragments. While the subroutine calls are the same regardless of which implementation one uses, the commands to start MPI and compile and runMPI programs vary by implementation. In this and subsequent sections I focus on implementation independent aspects of MPI though I briefly consider the steps necessary to install and use one of the freely available implementations of MPI in the Appendix.

An obvious first question is 'how does an MPI program differ from a non- MPI program?' To answer this question, consider the FORTRAN version of the canonical first C program that prints the message 'Hello, World!':

```
program hello-world
implicit none
write(6,*)'Hello, World!'
stop
end
```

Suppose we want to rewrite this program in parallel so that each process prints the message. Using MPI we could write

```
program parallel_hello_world
implicit none
```

```
include ’mpif.h’
integer ier,my_rank,status(MPI_STATUS_SIZE),numprocs
call MPI_INIT(ier)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nlimprocs,ier)
call MPI_COMM_RANK(MPI_COMM_WORLD,my_rank,ier)
write(6,*)’Hello, World from process’,my_rank,’!’
call MPI_FINALIZE(ier)
stop
end
```

The specifics of compiling and running this program will vary by implementation. If we were to run this program with four processes, the output would look something like

```
Hello, World from process 1!
Hello, World from process 0!
Hello, World from process 2!
Hello, World from process 3!
```

This output illustrates an aspect of parallel computing that is different from serial computing: the output is not printed in order.

There are a number of differences between the serial program and the parallel version. First, the parallel program has an include statement for the file mpif.h. This file is part of the MPI distribution. It contains variable definitions required by MPI and must be included in any source file that uses MPI. For example, the elements of the array status and the argument MPI_STATUS_SIZE are both defined in mpif.h. The estimation programs discussed below make use of this array.

The first MPI call in each source code file must be MPI_INIT, and the last call must be MPI_FINALIZE.[8] The first call initializes MPI at the program level, and calling MPI_FINALIZE ensures that this particular execution of the program exits in an orderly fashion.

The middle two MPI subroutine calls find out the total number of processes and the ‘rank’ of the calling process, respectively. MPI organizes processes into groups or ‘communicators’. At the time MPI is initialized only the default communicator is initialized. It is possible to create additional groups of processes, but it is not necessary for any of the examples in this paper. The default communicator contains all of the processes created when the program was run and has the name MPI_COMM_WORLD.[9] Each process does not initially know how many total processes there are and must call MPI_COMM_SIZE to find out. The first argument to MPI_COMM_SIZE is the name of the communicator, and the subroutine call returns the number of processes in the communicator as the second argument. Depending on the implementation of MPI, the number of processes will either be given on the command line when the executable is run or in a batch file that starts the parallel job.

Within a communicator, a process is identified by its rank. Ranks run from 0 to numprocs-1 where numprocs is the number of processes in the communicator. The rank of a process is important because it can be used to differentiate between the ‘master’ process with rank 0 and ‘worker’ processes with ranks greater than 0. To find out its rank, the process calls MPI_COMM_RANK. The first argument is again the name of the communicator, and the second argument returns the rank of the calling process.

One aspect of parallel computing that is conspicuous by its absence in this program is message passing. This trivial example is parallel only because multiple copies of it are run at the same time. The next step in building up to a non-trivial example is to include some message passing. The parallel version of hello-world can be made more interesting – if no more useful – by having each worker process pass its rank to the master process and the having the master process print out the rank:

```
program parallel_hello_world_v2
implicit none
include ’mpif.h’
integer ier, my_rank, status(MPI_STATUS_SIZE),
& numprocs,rank, i
call MPI_INIT(ier)
call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ier)
call MPI_COMM_RANK(MPI_COMM_WORLD,my_rank,ier)
if ( my_rank eq. 0 ) then
do i = 1, numprocs - 1
call MPI_RECV(rank, 1, MPI_INTEGER, MPI_ANY_SOURCE,
& MPI_ANY_TAG, MPI_COMM_WORLD, status, ier)
write(6,*)’Hello, World from process’, rank,’!’
end do
else
call MPI_SEND(my_rank, 1, MPI_INTEGER, 0, 0,
& MPI_COMM_WORLD, ier)
end if
call MPI_FINALIZE(ier)
stop
end
```

Sample output from this program when run with four processes looks like

```
Hello, World from process 1!
Hello, World from process 3!
Hello, World from process 2!
```

This output looks similar to the output above except that the rank of the master process is not printed.

There are two main differences between the first parallel program and this one. The processes executing the second program execute different lines of code depending on the value of my_rank. In the examples in this paper, the process with my_rank equal to zero will be the master process, and the others will be workers. The master process will partition data and send it to workers, update parameters, and generally be the central point of communication within the program.[10] The worker processes will do the computational work and pass their results back to the master. In the example program above, the master receives some data (the worker's rank) from the worker and prints the worker's rank. The workers send data to the master and do nothing else.

The second difference is that the second version of the program involves message passing. In this example, each of the three worker processes calls MPI_SEND once, and the master calls MPI_RECV three times. Thus, for each send there is one and only one matching receive. Because sending and receiving data is such a fundamental part of using MPI, it is worthwhile to spend some time discussing the basic send and receive commands. The syntax for sending a message in MPI is

```
MPI_SEND(buffer, elements, MPI_DATA_TYPE, destination,
tag, MPI_COMMUNICATOR, ier),
```

where buffer is the name of the scalar or array to send (my_rank in the example above), elements is the number of elements in buffer (1 in the example above because my_rank is a scalar), MPI_DATA_TYPE is the type of data, destination is the rank of the process to which we are sending the message, tag is a scalar integer that can be used to provide additional information to the receiving process, and MPI_COMMUNICATOR is the name of the relevant communicator (exclusively MPI_COMM_WORLD in this paper). For the examples in this paper, only a few MPI data types are used. These are MPI_INTEGER which is the same as integer in FORTRAN, MPI_REAL which is the same as single precision real in FORTRAN, and MPI_DOUBLE_PRECISION which is the same as double precision in FORTRAN.

For a more specific example, suppose the master needs to send a twenty element real array of parameters called beta from process 0 to process 2. The correct syntax is

```
if ( my_rank eq. 0 ) then

tag = 0
call MPI_SEND(beta,20,MPI_REAL,2,tag,MPI_COMM_WORLD,ier)
endif
```

Sending the message with a tag equal to zero may provide some information to the worker (and it must be set to something in any case). For example, in the econometric application below, worker processes use the value of the tag to decide whether to receive data, do work, or exit.

Receiving data is similar to sending it with one fundamental difference. When sending data, you must specify the rank of the receiving process, and you must specify a tag even if the tag is meaningless. When receiving data, you can specify a specific process from which to receive data and/or a specific tag for which to look, or you can specify wildcards for either or both of these components. This feature will allow the master to receive data from any worker without worrying about rank of the specific worker. The syntax for the MPI_RECV is

```
MPI_RECV(buffer, elements, MPI_DATA_TYPE, source, tag,
MPI_COMMUNICATOR, status, ier).
```

The syntax is very similar to the sending call except for the argument status. The integer array status is defined in the header file mpif.h. This array contains the scalar rank of the sender in location status (MPI_SOURCE), the message tag in status(MPI_TAG), and an error code.11 In cases where wildcards are used for source or tag, we can obtain this information from status. The following example will illustrate how to obtain this information.

To understand how wildcards are used, suppose that we are estimating a maximum likelihood problem where the worker processes compute partial likelihoods over subsets of the data. The master process receives these values and adds them up to compute the value of the likelihood function. The syntax to receive the partial likelihoods and add them up is

```
if ( my_rank eq. 0 ) then
likely = 0.0d0
do j = 1, numprocs-1
call MPI_RECV(partial_likely, 1,
& MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
& MPI_ANY_TAG, MPI_COMM_WORLD, status, ier)
likely = likely + partial_likely
sender_rank = status(MPI_SOURCE)
tag = status(MPI_TAG)
end do
endif
```

In this example the master process loops over all of the workers, receives a message (which it calls partial_likely) from one of the workers and adds this number to likely. Because the rank of the source process is specified using the wildcard MPI_ANY_SOURCE, the master is willing to receive the message from any worker. The rank of the process sending the message is obtained from status and is stored in the scalar sender_rank. Knowing the rank of the sending process will be important in the econometric example because the master process will send more data to the worker from which it most recently received data.

The subroutines MPI_SEND and MPI_RECV that send data from one process to another are examples of point-to-point communications routines. MPI also has a number of subroutines that allow one process to communicate with many other processes at one time. These collective communication routines are similar to the point-to-point routines discussed above in their basic syntax with one difference. When a collective communication routine is used, all processes in the relevant communicator must call that routine. In the examples below, there will be occasions where it is convenient to ‘broadcast’ data from the master to all of the workers as a group. The MPI command MPI_BCAST is used for this purpose. The syntax is

```
MPI_BCAST(message, elements, MPI_DATA_TYPE, source,
MPI_COMM_WORLD, ier),
```

where the components should be familiar by now. Note that each receiving process must call MPI_BCAST as well. Consider this code fragment

```
if ( my_rank .eq. 0 ) then
```

```
call get_data(X)
call MPI_BCAST(X,500,MPI_REAL,0,MPI_COMM_WORLD,ier)
else
call MPI_BCAST(X,500,MPI_REAL,0,MPI_COMM_WORLD,ier)
endif
```

In this code fragment, the master process calls a subroutine that obtains some data in an array called X. The call to MPI_BCAST indicates that the array X consists of 500 real numbers; perhaps it contains information on five variables for one hundred people. The master then broadcasts this array to all of the workers by calling MPI_BCAST. Each worker process must also call MPI_BCAST with the same arguments.

2.3. SUMMARY

As surprising as it might seem, the seven commands that have been introduced are sufficient to use MPI to estimate the parameters of a maximum likelihood problem. (Actually the subroutine MPI_BCAST is not needed to estimate the model.) There are, of course, many other commands that may be useful depending on the problem.[12] I next introduce the econometric problem, the data, and the serial solution before resuming the discussion of MPI.

3. The Econometric Problem

Consider a model where women choose whether to marry, work, or participate in the Aid to Families with Dependent Children program (AFDC) each year.[13] If we assume that married women cannot receive AFDC and single women cannot receive AFDC and work, then each period each woman must choose one of five possible choices.14 Assume that the utility of choice *j* for person *i* is

$$U_{ij} = X_i \beta_j + \epsilon_{ij} \; . \qquad (1)$$

If the _'s are iid Extreme Value, then this is a standard multinomial logit problem.
Let *β* denote the vector of *βj* 's. The probability that choice *j* is chosen is given by

$$P_{ij}(\beta) = \frac{e^{X_i \beta_j}}{\sum_{k=1}^{5} e^{X_i \beta_k}} \, , \qquad (2)$$

and the log-likelihood function for *N* observations is given by

$$\ln L(\beta) = \sum_{i=1}^{N} \sum_{j=1}^{5} d_{ij} \ln P_{ij}(\beta) \qquad (3)$$

where *dij* = 1 if choice *j* is chosen by person *i* and *dij* = 0 otherwise. The econometric problem is to find the value of β that maximizes the value of the likelihood function.

Clearly this problem does not require parallel programming – or any programming beyond a standard statistical package such as SAS or Stata. However, its simplicity makes it the perfect setting in which to illustrate parallel programming because the econometric problem does not get in the way of the message passing yet we face many of the same conceptual problems as we face when parallelizing a harder problem.

4. Data

The data for the project come from the Panel Study of Income Dynamics (PSID).[15] The initial sample extract consists of all women who are observed as both children and adults during the time period 1968 to 1988.16 The data used in the econometric application are the first 4800 person-years from the initial sample. For simplicity, I treat each person-year as an independent observation. Descriptive statistics are presented in Table I. The average age of women in the sample is twenty-six years. On average these women have just about a high school education. Almost half of the women are black, and these women have just over one child. These characteristics reflect the fact that the data include women from the PSID's poverty sample.

Information about the choices made by these women is contained in Table II. Women choose to receive AFDC about fifteen percent of the time. They choose the other single choices about twenty-seven percent of the time, and they choose to marry about sixty percent of the time.

Table I. Descriptive statistics.

Variable	Mean	Std. deviation
Age	26.07	5.02
Years of education	11.91	1.66
Black	0.49	0.50
Number of children	1.37	1.18

Table II. Choices.

Choice	Person-years	Percentage
Single, not working, no AFDC	239	4.98
Married, not working, no AFDC	1346	28.04
Single, working, no AFDC	1073	22.35
Married, working, no AFDC	1456	30.33
Single, not working, AFDC	686	14.29

5. The Serial Solution

Because the serial solution provides a baseline against which we can compare the parallel solution, I begin with the serial solution to the multinomial logit problem. The FORTRAN code for this program is longer than the program fragments introduced above. Consequently, I have listed the code for this program at the end of the paper in Listing 1. I have only included the main program in Listing 1. I will describe the subroutines as needed, but in order to save space, I do not list them.[17] None of the subroutines change as the program is converted from serial to parallel.

The program serial does a number of things. At lines 7 and 8 it calls subroutines to read the initial guess of the parameters and to read the data, respectively. Although it is not obvious from the listing, the data are made available in a common block that can be accessed by other subroutines as needed. Subroutine get_parameters returns three arguments: an array called parm that contains the initial guess of the parameters, an array called alabel that contains variable labels, and an array called estimflag. Each element of estimflag is equal to one if the corresponding coefficient is being estimated and zero otherwise.

The loop in lines 19 to 32 computes the value of the likelihood function for a given guess of the parameters. For each person, a call to the subroutine get_probability returns the probability of the chosen choice in the scalar p and the vector of derivatives of the probability with respect to each of the parameters in the array dp. The value of the log-likelihood function, the gradient vector (gradient), and the approximation to the hessian (hessian) are then input into the subroutine update_parms.[18] This subroutine checks for convergence and, if necessary, updates the parameter estimates. If the parameters have converged, the process stops; if they have not, the steps are repeated until convergence.

The estimated coefficients are presented in Table III.[19] Because this is a discrete choice model, the coefficients are only identified relative to one of the choices. The decision to be single, not work, and not receive AFDC is taken as the base choice. The results show that being black, having children, and being young are all associated with AFDC participation. Education is positively associated with both work and marriage. Being black is negatively associated with marriage. Children are positively associated with marriage and negatively associated with employment. In summary, there is nothing remarkable about the results.

6. A Parallel Solution

The serial code in Listing 1 can be parallelized in a number of ways. The most natural place to look for possible parallel gains is in loops. There are obvious candidates in Equations (2) and (3). In Equation (2) there is a sum over choices, and there is a sum over individuals in Equation (3). Although it is not obvious from the equations, there is also a loop to compute $X\beta$ in the FORTRAN code. Any of these loops are candidates to be executed in parallel, but some of them will require more work to code and may have less benefit than others.

Deciding on the appropriate place to parallelize the program involves balancing computation and communication. First, consider parallelizing the loop that computes $X\beta$. This loop can be made parallel by passing one component of X and one component of β to a worker, having that worker pass back the product, and then adding the results. Because there are five *X*s, parallelizing this loop results in passing ten messages each time $X\beta$ must be calculated. This results in what is called a *fine-grained* level of parallelism. In other worlds, it involves passing many small messages. This level of parallelism may be appropriate on a massively parallel computer, but it is almost certainly too fine for most other platforms.

The loop that sums the $eX\beta$'s in Equation (2) is a better candidate because it is less fine-grained. However, the best candidate is the loop over individuals in Equation (3). This is a relatively *coarse-grained* level of parallelism, but it is easy to implement and requires relatively little message passing. The basic idea is that the master process will give each worker a subset of data, and each worker will pass to the master the value of the likelihood function, gradient, and approximation to the hessian for that subset. If there is more work to do, the master will give the worker another subset of the data. Once the master has received the partial likelihoods, gradients, and hessians for the whole sample, it updates the parameters in the same manner that the serial program above did. If the likelihood function has converged, the process stops, and if it has not converged, the master passes the updated parameters to the workers and the process is repeated. If one were fortunate enough to have sufficient processors to exhaust the gains to parallelizing Equation (3), then one might achieve additional gains from parallelizing Equation
(2).

Table III. Multinomial logit results.

Variable	Estimate	T-statistic
Married, not Working, no AFDC		
Constant	1.113	1.90
Age	–0.063	–3.86
Education	0.182	3.87
Black	–2.353	–14.46
Children	1.020	14.33
Single, working, no AFDC		
Constant	–2.420	–4.07
Age	0.026	1.64
Education	0.297	6.17
Black	–0.356	–2.18
Children	–0.053	–0.74
Married, working, no AFDC		
Constant	–0.703	–1.20
Age	0.017	1.06
Education	0.201	4.19
Black	–1.663	–10.42
Children	0.561	7.83
Single, not working, AFDC		
Constant	1.291	1.95
Age	–0.081	–4.66
Education	–0.032	–0.60
Black	0.756	3.76

Children	1.063	13.99
Likelihood function	–6141.397	
Observations	4800	

Even with this relatively coarse-grained level of parallelism, there is another decision to be made that will affect performance: how much of the data should the workers be given at one time? The answer to this question depends on the architecture on which the code will be run. Suppose the parallel machine consists of *M* identical unix workstations which are not shared with others. In this case it makes sense to give each worker *N/M* observations where *N* is the sample size. Doing so minimizes the number of messages that must be passed.

Now suppose that one of two cases exists: 1) there are *M* identical workstations that are shared with others or 2) there are *M* workstations that vary in speed. In either of these cases, some of the workers will finish more quickly than others. If the data are split evenly, the faster processors will sit idle while the slower processors are finishing their work. In this case, each worker should initially be given some number of observations less than *N/M*. Then when the faster workers finish, they can be given more data on which to work. This technique is called *dynamic load balancing*. In the extreme, each worker could work on one observation at a time. While there may be cases where this solution is appropriate, it almost certainly involves too much communication. The 'best' number of observations is hard to know – particularly when the machines are shared and the work load on any given machine can change as other users' jobs start or finish. I will provide some limited evidence in Section 8.1.2 on how different decisions affect run-time.

Because the parallel code is longer than the serial code, I present it in three separate listings. The first listing contains the code that is executed by both the master and workers, the second listing contains the master's code, and the third listing contains the worker's code.

6.1. COMMON CODE

The code that is executed by both the master and the workers is presented in Listing 2. This code has the same basic structure of the simple message passing example above. In particular, each process must call MPI_INIT, MPI_COMM_WORLD, MPI_COMM_RANK, and MPI_FINALIZE. The remainder of this code declares relevant variables.

There are several differences between the common code and the analogous parts of the serial code. Most of them are minor, but a few warrant some discussion. First, the parallel code defines a parameter called COUNT. This parameter is not used in the serial program. In the parallel program, COUNT is the number of observations that the master passes to the workers as a subset of the data. Second, the parallel program defines two other variables that are not defined in the serial program: number_sent and number_received. As the master is sending observations to the workers in subsamples of COUNT observations, the variable number_sent is used to keep track of how many observations have been sent to the workers so that the master stops sending data after all 4800 observations have been sent. At the same time as the master is sending observations to the workers, it is also receiving likelihood contributions from the workers. The variable number_received keeps track of how many observations it has received from the workers. Once it has received all 4800 observations, it will no longer attempt to receive more.

A third difference is the fact that the parallel program includes the common block data in the main routine while the serial version does not. In the parallel version of the program, the master process is the only process that has access to the data initially, and it must pass the data to the workers. Because the subroutine get_data makes the arrays *x* and *y* available through the common block data, the main routine of the parallel program must have access to this common block.

6.2. THE MASTER'S CODE

The process whose rank is zero is denoted the master process. The code executed by the master process is contained in Listing 3. The master process obtains the parameters and data by calling the subroutines get_parameters and get_data, respectively. Because the master process is the only process that has access to the data initially, both x and y must be passed to the workers. The MPI_BCAST subroutine is used to broadcast the data to all of the workers. The workers also need to know which parameters are being estimated, and this information is broadcast to them as well. Thus, the subroutine calls in lines 5, 6, and 7 of Listing 3 broadcast the arrays x, y, and estimflag to the workers. These arrays are unchanging throughout the execution of the program and are only passed once.

Once the data have been broadcast to the workers, the master initializes the likelihood function, gradient, and hessian. It then loops over the worker processes in lines 18 to 23 and passes each worker an array called index. In order tominimize the amount of message passing, the master passes all of the data to all of the workers initially and then passes the beginning and ending observation numbers to the workers during computation.[20] The array index is a two element integer array containing the first observation and last observation in the receiving process's subset of the data. This message is sent with tag = 1, and the workers will use this information to determine what to do after they receive the message. As the observations are sent, the scalar number_sent is updated in line 22.

At this point in the program's execution, the workers have received the data, and they have received the beginning and ending points for their subsection. However, they have not yet received the initial parameter estimates. These estimates were obtained in the subroutine call to get_parameters and only the master has access to the initial guess. The array parm is broadcast to the workers using MPI_BCAST in line 24. The fundamental difference between the arrays x, y, and estimflag and the array parm is that the elements of parm change over the course of the program's execution while the data do not.

The likelihood function is calculated in loop contained in lines 26 through 48. Section 6.3 will discuss the worker's code, but for now assume that the worker is able to calculate and send three messages in order: the likelihood contribution, gradient, and hessian over its part of the data. The master receives the partial likelihood and in line 28 finds the rank of the sending process. It uses this information in the MPI_RECV's for the gradient and the hessian. Note that the wildcard MPI_ANY_SOURCE that is present in the receive for part_likely has been replaced with sender.

After receiving all three messages, the master updates the likelihood function, gradient, hessian, and the number received. It then checks to see whether there is more data to be sent in line 39. If there is, it sends the beginning and ending observations with tag = 2 and updates the number sent. If all of the data have been sent, the master waits to receive data. Once the number sent and the number received are both equal to 4800, the subroutine update_parms is called as it was in the serial version of the program. There are no differences between the serial and parallel versions of these supporting subroutines.

After the likelihood function has converged, the master again sends each worker a message consisting of the array index. The contents of the message is less important than the value of the tag. In this case the tag is equal to 0.

The master has sent workers the array index at three different points in its code. In each place the tag was different. First it was 1, then it was 2, and now it is 0. There is a reason for this. In this last case, there is no more work for the workers to do, and we will see in the worker's code that they expect no more work when the tag is 0. What is the difference between the first two cases? In the first case, after index is sent, the master broadcasts the array parm. In the second case, there is no need to pass the parameters because they have not changed. The master is using the value of the tag to tell workers whether they need to call MPI_BCAST, simply do more work, or expect no more work.

6.3. THE WORKERS

The code executed by each worker process is listed in Listing 4. The workers first call MPI_BCAST to receive the arrays x, y, and estimflag. They next receive the array index and immediately obtain the tag from the array status.

As noted at the end of Section 6.2, the value of the tag determines the worker's course of action. If the tag is equal to zero, the worker does nothing which results in the worker calling MPI_FINALIZE from Listing 2.

The only difference between the worker's actions if the tag is equal to one or two is that the worker calls MPI_BCAST to receive the array parm if the tag is equal to one. For simplicity the computation of the worker's contribution to the likelihood function, gradient, and hessian has been moved to a subroutine called calc_likely. This subroutine is listed in Listing 5. It takes estimflag, parm, and index as input and returns part_likely, part_gradient, and part_hessian. These latter arrays are passed back to the master in three separate messages. Once the worker has passed the results to the master, it returns to line 4 to wait for another array of beginning and ending observation numbers on which to work.

6.4. RESULTS

The results of the estimation of the parallel version of the model are the same as the results for the serial case. The important issue for parallel computing is the amount of time it takes to find the answer, and I postpone that discussion until Section 8.1.

7. Reducing the Number of Messages

While this program requires relatively little message passing, it is using three messages to send part_likely, part_gradient, and part_hessian from each worker to the master. (It is also broadcasting x, y, and estimflag as separate messages, but this happens only once.) It may be possible to save time if these three messages can somehow be collapsed into one message. There are two ways this goal can be accomplished.

7.1. PACKED MESSAGES

The simpler possibility is to pack all three messages into a new buffer and send that message to the master. MPI provides the commands MPI_PACK and MPI_UNPACK to pack and unpack messages, respectively. The syntax for MPI_PACK is

```
MPI_PACK(message, elements, MPI_DATA_TYPE, buffer, size_of_buffer, position,
MPI_COMM_WORLD, ier),
```

where message is the array to be packed, buffer is the name of the buffer into which message is being packed, size_of_buffer is the size of the buffer in bytes, and position is the position in the buffer that message begins. For the first message to be packed, position = 0; position is updated automatically for subsequent messages. Because only the number of bytes matters, the FORTRAN data type of buffer does not matter. The only requirement is that the array be large enough (in bytes) to store the message. The following code fragment packs and sends the three messages as one packed message

```
character buffer(10000)
integer position
. . . . . . .
position = 0
call MPI_PACK(part_likely, 1, MPI_DOUBLE_PRECISION,
& buffer, 10000, position, MPI_COMM_WORLD, ier)
call MPI_PACK(part_gradient, 25, MPI_DOUBLE_PRECISION,
& buffer, 10000, position, MPI_COMM_WORLD, ier)
call MPI_PACK(part_hessian, 25*25, MPI_DOUBLE_PRECISION,
& buffer, 10000, position, MPI_COMM_WORLD, ier)
call MPI_SEND(buffer, 10000, MPI_PACKED, 0, my_rank,
& MPI_COMM_WORLD, ier)
```

In this example, the message buffer is a 10000 byte array called buffer. Each of the three messages is packed into this array, and the array is sent using a standard MPI_SEND. The only difference between this send and the previous sends is the data type. Packed data have a special MPI data type called MPI_PACKED. There is no analogous data type in FORTRAN.

The message is received using MPI_RECV with the data type MPI_PACKED. Once the packed message has been received, it can be unpacked using the code

```
position = 0
```

```
call MPI_UNPACK(buffer, 10000, position,
& part_likely, 1, MPI_DOUBLE_PRECISION,
& MPI_COMM_WORLD, ier)
call MPI_UNPACK(buffer, 10000, position,
& part_gradient, 25, MPI_DOUBLE_PRECISION,
& MPI_COMM_WORLD, ier)
call MPI_UNPACK(buffer, 10000, position,
& part_hessian, 25*25, MPI_DOUBLE_PRECISION,
& MPI_COMM_WORLD, ier).
```

Packing and sending data reduces the number of messages that must be sent by each worker from 3 to 1, but it requires three calls to MPI_PACK and three calls to MPI_UNPACK. It would be nice if there were a way to reduce the number of messages without incurring the cost of six subroutine calls for each message that is sent.

7.2. USER DEFINED DATATYPES

There is a way to accomplish the goal of sending fewer messages without packing and unpacking data each time a message is sent. MPI allows for a broad range of user defined datatypes. Creating a user defined datatype is more complicated than sending three separate message or sending one packed message, but there may be significant performance gains.

For (relative) simplicity, I focus on sending the elements of a FORTRAN common block as one message. Because each of these user-defined datatypes is constructed from the beginning memory address of an array, the number of elements in the array, and the extent of the array (the number of bytes), it is simplest to deal with arrays guaranteed to be located contiguously in memory. In FORTRAN, the elements of a common block are guaranteed to be located contiguously in memory. Thus, if the common code defines a common block consisting of part_likely, part_gradient, and part_hessian, we can implement this solution.

Before looking at the relatively complicated case of sending 2 arrays and a scalar, consider the case of sending one double precision number and one integer. Suppose that the program requires that we send the value of the likelihood function and the number of observations from the master to a worker and that these variables are part of a common block called example.

Rather than sending two separate messages, MPI commands can be used to create a new data type called, for example, new datatype that consists of one double precision number and one integer. Thus, there are two members of the new data type, and each of them has one element.[21] In defining newdatatype, we must specify the number of members (2), the number of elements in each member (1 for each), the displacement in bytes from the beginning of the buffer for each member, and each member's MPI data type. Integer arrays are used to store the numbers of elements, the displacements, and the data types. Consider the following code

```
integer blocklengths(2), types(2),
& displacements(2), extent, newdatatype,
& ier, observations
```

```
real*8 likely
common / example likely, observations
blocklengths(1)1
blocklengths(2)1
displacements(1)0
call MPI_TYPE_EXTENT(MPI_DOUBLE_PRECISION, extent, ier)
displacements(2) = extent
types(1) = MPI_DOUBLE_PRECISION
types(2) = MPI_INTEGER
```

The arrays blocklengths, displacements, and types are arrays whose lengths are defined by the number of members of the common block. Each element of the array blocklengths contains the number of elements in the appropriate member of the common block. The displacement of the first member of the common block is zero because that scalar or array is the starting point in memory. The displacement for the second member depends on the number of elements in the first component of the common block and on the extent of each of those elements. The extent is the number of bytes used by a scalar of the relevant MPI data type. The command MPI_TYPE_EXTENT is used to find out the appropriate value for extent. In this example, this command is used to find out the number of bytes in a double precision number. Finally the array types contains the relevant MPI data types.

Once these arrays have been constructed, the new data type can be defined and ‘committed’ for use in the program:

```
call MPI_TYPE_STRUCT(2, blocklengths,
& displacements, types, newdatatype, ier)
call MPI_TYPE_COMMIT(newdatatype, ier).
```

The first argument to MPI_TYPE_STRUCT is the number of members of the structure (common block). After MPI_TYPE_COMMIT has been called, the entire common block example can be passed using standard MPI_SEND and MPI_RECV commands where the data type is newdatatype. Once the user defined data type is no longer required, it can be freed by calling the subroutine MPI_TYPE_FREE:

```
call MPI_TYPE_FREE(newdatatype, ier).
```

The code to construct newdatatype must be executed by all of the processes. After the new data type has been committed, messages can be passed using this data type in place of, for example, MPI_REAL using the normal MPI send and receive (or broadcast) commands.

I now turn to the syntax for sending part_likely, part_gradient, and part_hessian in one message. Because it takes more code to do this than will comfortably fit in a code fragment in the text, I list the code in Listing 6 at the end of the paper.

The code in Listing 6 is a complete program that defines a new data type and calls subroutines to send and receive a message consisting of the new data type. Because the common block partials has three members, each of the arrays describing the new data type has a length of three. The blocklengths are the number of elements in part_likely (1), part_gradient (25), and part_hessian (625). As with the simple example above, the initial displacement is zero. The displacement for the array part_gradient is just the length of one number of type MPI_DOUBLE_PRECISION. The displacement for the beginning of the array part_hessian is the length of the single double precision number part_likely plus the displacement of the twenty-five double precision numbers in part_gradient. Because the displacement of a scalar of type MPI_DOUBLE_PRECISION is extent, this total displacement is extent + 25*extent. Finally, each of these arrays is MPI data type MPI_DOUBLE_PRECISION.

Once the basic structure has been defined, MPI_TYPE_STRUCT is called to formally define newdatatype, and MPI_TYPE_COMMIT is called to make the data type available for use in the program. After the new data type has been made available, messages using the new datatype can be sent and received like messages using any predefined datatype as evidenced by the MPI_RECV issued by the master. There are a couple of interesting aspects to the sends and receives using newdatatype. First, the buffer that is being sent and received is called part_likely. This variable is the first element of the common block and is essentially a pointer to the beginning of the area of memory occupied by the common block partials. Second, the message contains only one element. In this case, we are passing one copy of partials.[22]

There are at least three ways to pass the likelihood, gradient, and hessian to the master. Which one should be used? As with many aspects of parallel programming, the answer depends on the number of messages, the size of the messages, and the particular parallel architecture under consideration. I will provide some simple numbers below for the example econometric application, but those numbers are not generalizable.

In most applications that pass a reasonably large number of messages, creating a user defined data type is likely to be the most efficient solution. It has the advantage of minimizing the number of messages that are passed without incurring the repeated overhead of packing and unpacking data. If at all possible, however, it is best to conduct tests that mimic the specific problem under consideration on the specific platform that will be used.

8. Timing, Profiling, and Debugging

Timing programs can help one decide on the number of observations to pass to workers at one time and whether to send multiple messages, pack and unpack data, or create a specific data type. MPI contains timing commands to make this possible. Beyond simple timings, it can be very helpful to be able to visualize program execution so that one can understand the communication patterns in the program. MPI provides logging capability, and different MPI implementations have different ways to read these log files. Finally, the ability to debug programs is critically important working working on a complicated estimation program. Debugging capability is highly implementation specific. In this section, I briefly touch on each of these issues.

8.1. TIMING PROGRAMS

There are two reasons to write parallel code. In some cases the problem may simply be too large to solve on a serial machine, but in most cases it will be possible to solve the problem in serial and in parallel. In the latter cases the primary reason for using parallel computing is the desire to reduce the time it takes to find a solution. Consequently it is useful to time the parallel program. MPI provides specific calls for timing purposes.

The FORTRAN function MPI_WTIME() returns the number of seconds since some fixed point in the past (that is guaranteed not to change during program execution). The following code fragment demonstrates how MPI_WTIME can be used to time a segment of a program.

```
real*8 starttime, endtime, totaltime
starttime = MPI_WTIME()
! CODE TO BE TIMED
endtime = MPI_WTIME()
totaltime = endtime - starttime
write(6,*)’total time =’,totaltime
```

This function can be used to provide some simple answers to some of the questions raised at the beginning of this section.

8.1.1. *Calculating Speedup*

Perhaps the most fundamental questions one would like to answer when parallelizing a serial program are ‘how much faster is the parallel version?’ and ‘how much faster does the program run as the number of processors is increased?’. These questions get at the notion of the parallel speedup of the program. The speedup for *p* processes, *Sp*, is calculated as *Sp* = *T*1/*Tp* where *Tn* is the time for *n* processes.[23] If a problem takes 30 minutes to solve on one processor and 20 minutes on three processors, then the speedup for three processors is 1.5. An ideal speedup is equal to the number of processors, but this goal will be limited by the amount of message passing and the fact that some of the code is inherently serial.[24]

The serial version of the multinomial logit program takes about 3.38 seconds to run on a Pentium III personal computer with dual 550 Mhz processors running the Linux operating system.25 Running the parallel version of the code on the same machine using three processes (2 workers and a master) with COUNT = 2400, the program takes about 1.82 seconds. Thus, the speedup is about 1.86 which is less than two because messages must be passed which adds overhead that is not present in the serial version.[26]

Speedup is not necessarily linear as processors are added. Some problems simply do not scale well beyond some number of processors, and others are limited by the hardware configuration (or algorithm). Reestimating the parallel version on a cluster consisting of the dual-PIII 550 and a dual Pentium Pro 200 results in a time of 1.61 seconds. The execution time still falls but the speedup from one processor to four processors is only 2.09. These numbers are specific to the set-up used to conduct the test, and there are at least two reasons for this result. First, the networking connection between these two machines is relatively slow. Paying more attention to this aspect of performance would improve the speedup somewhat.[27] Second, the additional processors are less than half as fast as the first two processors.

However, even conditional on the physical setup, the speedup for this problem is constrained by the relatively small amount of computation relative to communication. In order to approximate a more realistic computational situation, I modified the estimation program so that the probability calculation routine is called 10,000 times per individual. Only the last result is used in calculations so this change slows the program without changing the answer. With this change the program takes 51.4 minutes to solve in serial, 25.9 minutes to solve in parallel with two workers, and 19.5 minutes to solve using all four processors. Thus, the speedup in moving from serial to parallel with two workers is 1.98, and the speedup of moving from serial to four workers is 2.64. Once again, the speedup is not linear in part because of the physical characteristics of the machine, but changing the ratio of computation to communication (to a more realistic level) has improved the speedup dramatically. As the CPU demands become larger and larger, the message passing overhead becomes less and less important. Additionally, although it is not a factor in this problem, spreading a large problem across multiple machines may improve the memory performance of a program. In a scenario, where the serial version of the program is so large that it does not fit in physical memory, this effect could be dramatic.

Table IV. Timings for different values of COUNT.

COUNT	Seconds to completion
1	38.57
100	1.98
300	1.60
500	2.27
600	1.61
1200	2.68

8.1.2. *Calculating the Optimal Value of* COUNT

Simple timings can be used to decide on the optimal value for the parameter COUNT. Table IV consists of the execution time for various levels of the parameter COUNT for the two computer cluster running 5 processes. The first row illustrates very clearly that parallel programming does not necessarily lead to a speedup over serial programming; it takes over 10 times longer to estimate the parameters using MPI and sending one observation at a time than it did to estimate the same parameters in serial!

The situation improves dramatically when COUNT is increased to 100. Smaller increases are observed when increasing COUNT to 300 or 600, and the execution time increases when the number of observations is increased to 1,200. Thus, it appears as though subsamples of between 300 and 600 observations are optimal for this problem given the specific hardware configuration. Interestingly, the program is noticeably slower when COUNT = 500 is used. This occurs because of poor dynamic load balancing characteristics when COUNT = 500.

8.1.3. *Deciding Whether to Send Multiple Messages, Pack Messages, or Create a User-Defined Data Type*

This simple timing routine can also be used to learn about the relative speed of the different methods of passing the information from the workers to the masters. In this simple example, there is very little difference among the three methods.

Run times were approximately 1.61 seconds when the user defined data types were used, 1.63 when the three messages were send separately, and 1.68 when messages were packed and unpacked.28 These differences may become more significant on more formidable problems.

8.2. PROFILING & DEBUGGING

8.2.1. *Profiling*

While timing programs can provide valuable information about performance, simple timings do not provide information about message passing patterns within the program. A number of profiling tools exist that can help the user understand where communication bottlenecks are occurring. Some of these, such as XMPI,[29] jumpshot,[30] and AIMS,[31] are freely available. These tools tend to be somewhat application specific. For example, XMPI is designed to work with the LAM implementation of MPI, and jumpshot has been developed alongside the mpich implementation of MPI.

While each of these tools is somewhat different, they share a number of common features. Perhaps the most useful is the capability to visualize program execution and performance. Typically this feature is implemented as a time line for each process running in the communicator. It may use different colors to denote the time spent engaging in CPU-intensive activity as opposed to waiting for data. The profilers will also typically indicate when messages passing is occurring and who the senders and receivers are.

8.2.2. *Debugging*

Debugging parallel code can be a daunting task because message passing adds an additional layer of complexity on top of the serial code. Depending on the specific error, a program may exit immediately, appear to run but give an incorrect answer, or simply hang. Many times the first instance is easiest to debug. For example, if COUNT = 2400 in the parallel multinomial logit program and the program is run with 4 worker processes, the program will exit immediately because there is no work for the workers to do in the initial loop.32

The other cases are typically harder to debug. In many cases a parallel program hangs because there is a communication problem. MPI_SEND and MPI_RECV are each ‘blocking’ in the sense that a send does not complete until a matching receive is called. Thus, if there are two processes and both processes call MPI_SEND and neither calls MPI_RECV, the program will hang. Finally, a program may obviously give a wrong answer either because of a problem with the serial code or the parallel code.

In many cases, the best debugging strategy is to take make only incremental changes in the program and to use print statements to understand where problems occur. Recently, however, it has become easier to debug programs using a symbolic debugger. The debugging capabilities vary significantly by implementation. When using either LAM or mpich, it is possible to open an xterm with a debugger for each process. Thus, if there are four processes, four separate xterms are opened. It is then possible to step through the code that is being executed by each process. It is typically easiest to debug with only two processes as that simplifies the number of windows one must keep track of. Nonetheless, debugging in two separate windows adds a new dimension to debugging.

9. Conclusion

As econometric problems become increasingly complicated, parallel computing is likely to be used more frequently to solve them. MPI has recently become the standard message passing language of choice. Because MPI has been implemented on a number of architectures, one can test and debug a program on a PC or a small cluster of PCs and then estimate the program on a higher end platform if one is available.

Many people who may be interested in using parallel computing may be unwilling to pay the full cost of learning to use the techniques. The purpose of this paper is to lower the cost of using MPI to solve a maximum likelihood problem. The paper introduces a number of simple MPI commands using examples that help build up the econometric application. The parallel solution to the econometric application involves subsetting the data and having different machines (or processors) work on different subsets in parallel. The program performs almost twice as fast using two processors than it does with one processor. Parallel computing can be used for econometric applications beyond solving maximum likelihood problems. On a fundamental level, the ScaLAPACK parallel library contains routines to solve systems of linear equations and eigenvalue problems.[33]

Monte Carlo problems may also be solved using parallel computing. The basic set-up should be obvious: N Monte Carlo replications may be distributed across M machines in the ways discussed above. The only thing that is not obvious is the manner in which (pseudo-) random numbers are generated and distributed so that they appear random and are repeatable. This topic is beyond the scope of this paper, but Foster (1995) contains a chapter on random number generation as well as references to other sources.

Acknowledgements

The author would like to thank Todd Stinebrickner and an anonymous referee for valuable comments that improved both the content and the clarity of the presentation.

Appendix

A. A Using the LAM Implementation of MPI

A.1. OBTAINING AND INSTALLING LAM

If an MPI implementation is not currently installed on your workstation, you will need to either have someone download it and install it for you or download and install it yourself. In this appendix I will briefly discuss obtaining and installing the LAM implementation of MPI (LAM Team 2000). The basic steps for installing other libraries such as the mpich implementation of LAM are very similar.

There are four steps in the process of installing LAM. First you must obtain the source code. To do this, go to http://www.lam-mpi.org/ and click on download. As of this writing, the latest stable version of LAM is 6.3.2. If you are the system administrator on your workstation (you have root access), save the file lam-6.3.2.tar.gz to /usr/local/src. Next you must uncompress, gunzip lam-6.3.2.tar.gz, and untar the file: tar -xvf lam-6.3.2. tar. Now you will have a /usr/local/src/lam-6.3.2 directory.

The second step in the installation process is to configure the compilation options for your system. By default LAM will be installed in /usr/local/lam-6.3.2, it will use rsh for network communication, and it will use your default FORTRAN compiler (eg, f77 or g77). You can change any of these options when you build LAM. For example, you might wish to compile LAM to use ssh rather than rsh for security reasons. To configure LAM in this manner, you would type./configure –with-rsh=”ssh -x”.

Once you have successfully configured LAM, you must actually compile the source code. To do this step, you type make. If there are no errors, you now have a working LAM installation. In order for users to be able to run parallel jobs, they must add the path /usr/local/lam-6.3.2/bin to their path environment variable.

A.2. STARTING AND STOPPING LAM

Before any parallel jobs can be run, MPI must be in some sense started or ‘booted’. Booting is analogous to physically turning on a PC. The difference is that we are taking several (already running) machines and making them aware of each other and capable of message passing. In the LAM implementation the lamboot command is used to boot the parallel computer, and the command wipe shuts it down in an orderly fashion. LAM uses a ‘boot schema’ to define the computers that make up the parallel machine. See the bhost man page for information about the syntax for the boot schema.

A.3. COMPILING AND RUNNING PARALLEL PROGRAMS

The FORTRAN source code must be linked to theMPI library in order to be able to call the MPI subroutines. The source code may be compiled before or after issuing the lamboot command; the parallel computer does not have to be booted in order to compile parallel code. The specifics of compilation vary by distribution, but for the LAM implementation the command mpif77 src.f compiles the source code in src.f and creates an executable file called a.out. The compiler wrapper mpif77 automatically links the appropriate MPI libraries to the source code contained in src.f. Finally, once the machine has been booted and the source code has been compiled, the parallel job is run with the command mpirun -np X a.out. This command runs X copies of the executable named a.out on the parallel machine.

B. Listings

```
! --- LISTING 1: SERIAL FORTRAN CODE
 !
      program serial
      implicit none
      logical converged
      character*20 alabel(25)
      integer iter, i, j, l, m, estimflag(25)
      real*8 p, dp(25), gradient(25), hessian(25,25),
     &       parm(25), likely
      call get_parameters(beta,alabel,estimflag)
      call get_data
      converged = .false.
      iter = 1
      do while ( .not. converged )
         do i = 1, 25
            gradient(i) = 0.0d0
            do j = 1, 25
               hessian(i,j) = 0.0d0
            end do
         end do
         likely = 0.0d0
         do i = 1, 4800
           call get_probability(parm,i,p,dp)
            likely = likely + dlog(p)
            do l = 1, 25
               if ( estimflag(l) .eq. 1 ) then
                  gradient(l) = gradient(l) + dp(l)/p
                  do m = 1, 25
                     if ( estimflag(m) .eq. 1 ) then
                        hessian(l,m) = hessian(l,m)
     &                                 + (dp(l)/p)*(dp(m)/p)
                    endif
                  end do
               endif
            end do
         end do
        call update_parms(gradient, hessian, parm, converged,
     &       likely, iter, estimflag, alabel)
        iter = iter + 1
     end do
     stop
     end
```

```
! --- LISTING 2: COMMON CODE FOR PARALLEL
!
      program parallel
      implicit none
      include 'mpif.h'
      logical converged
      character*20 alabel(25)
      integer i, j, l, m, estimflag(25),iter, y, numprocs,
     &        my_rank, status(MPI_STATUS_SIZE), index(2), tag,
     &        number_sent, sender, number_received, ier, COUNT
      parameter ( COUNT = 200 )
      real*8 parm(25), gradient(25), hessian(25,25), x, likely,
     &       part_likely, part_gradient(25), part_hessian(25,25)
      common / data /x(5,4800),y(4800)
      call MPI_INIT(ier)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ier)
      call MPI_COMM_RANK(MPI_COMM_WORLD,my_rank,ier)
      if ( my_rank .eq. 0 ) then

         MASTER'S CODE GOES HERE

      else

         WORKER'S CODE GOES HERE

      endif

      call MPI_FINALIZE( ier )
      stop
      end


! --- LISTING 3: PARALLEL MASTER'S CODE
      call get_parameters( parm, alabel, estimflag )
      call get_data
      converged = .false.
      iter = 1
      call MPI_BCAST(x, 5*4800, MPI_DOUBLE_PRECISION, 0,
     &     MPI_COMM_WORLD, ier)
      call MPI_BCAST(y, 4800, MPI_INTEGER, 0, MPI_COMM_WORLD, ier)
      call MPI_BCAST(estimflag, 25, MPI_INTEGER, 0,
     &     MPI_COMM_WORLD,ier)
      do while ( .not. converged )
         likely = 0.0d0
         do i = 1, 25
            gradient(i) = 0.0d0
            do j = 1, 25
               hessian(i,j) = 0.0d0
            end do
         end do
         number_sent = 0
         index(1) = 1
         do i = 1, numprocs-1
            index(2) = index(1)+ COUNT - 1
            call MPI_SEND(index,2,MPI_INTEGER,i,1,
     &           MPI_COMM_WORLD,ier)
            index(1) = index(2) + 1
            number_sent = number_sent + COUNT
         end do
         call MPI_BCAST(parm,25,MPI_DOUBLE_PRECISION,0,
     &        MPI_COMM_WORLD,ier)
         number_received = 0
         do while ( number_received .lt. 4800 )
            call MPI_RECV(part_likely, 1, MPI_DOUBLE_PRECISION,
     &           MPI_ANY_SOURCE, MPI_ANY_TAG,MPI_COMM_WORLD,
     &           status,ier)
            sender = status(MPI_SOURCE)
            call MPI_RECV(part_gradient, 25, MPI_DOUBLE_PRECISION,
     &           sender, MPI_ANY_TAG, MPI_COMM_WORLD, status, ier)
            call MPI_RECV(part_hessian, 25*25, MPI_DOUBLE_PRECISION,
     &           sender, MPI_ANY_TAG, MPI_COMM_WORLD,
     &           status, ier)
            number_received = number_received + COUNT
            do l = 1, 25
               gradient(l) = gradient(l) +  part_gradient(l)
               do m = 1, 25
                  hessian(l,m) = hessian(l,m)
     &                         + part_hessian(l,m)
               end do
            end do
            likely = likely + part_likely
            if (number_sent .lt. 4800 ) then
               index(2) = index(1) + COUNT - 1
               if(index(2).gt.4800)index(2)=4800
               call MPI_SEND(index,2,MPI_INTEGER,sender,
     &              2,MPI_COMM_WORLD,ier)
               index(1) = index(2) + 1
               number_sent = number_sent + COUNT
            endif
         end do
         call update_parms(gradient, hessian, parm, converged,
     &        likely, iter, estimflag, alabel)
         iter = iter + 1
      end do
      do j = 1, numprocs-1
         call MPI_SEND(index,2,MPI_INTEGER,j,0,MPI_COMM_WORLD,ier)
      end do
```

```
!---LISTING 4: PARALLEL WORKER's CODE
 !
 1    call MPI_BCAST(x, 5*4800, MPI_DOUBLE_PRECISION, 0,
     &     MPI_COMM_WORLD,ier)
 2    call MPI_BCAST(y,4800,MPI_INTEGER,0,MPI_COMM_WORLD,ier)
 3    call MPI_BCAST(estimflag, 25, MPI_INTEGER, 0, MPI_COMM_WORLD, ier)
 4    call MPI_RECV(index, 2, MPI_INTEGER, 0, MPI_ANY_TAG,
     &     MPI_COMM_WORLD, status, ier)
 5    tag = status(MPI_TAG)
 6    if ( tag .eq.1 ) then
 7       call MPI_BCAST(parm,25,MPI_DOUBLE_PRECISION,0,
     &        MPI_COMM_WORLD,ier)
 8       call calc_likely(part_likely, part_gradient, part_hessian,
     &        estimflag, index, parm)
 9       call MPI_SEND(part_likely, 1, MPI_DOUBLE_PRECISION,
     &         0, my_rank, MPI_COMM_WORLD, ier)
10       call MPI_SEND(part_gradient, 25, MPI_DOUBLE_PRECISION,
     &         0,my_rank,MPI_COMM_WORLD,ier)
11       call MPI_SEND(part_hessian, 25*25, MPI_DOUBLE_PRECISION,
     &         0, my_rank, MPI_COMM_WORLD, ier)
12       goto 4
13    else if ( tag .eq. 2 ) then
14       call calc_likely(part_likely, part_gradient, part_hessian,
     &        estimflag, index, parm)
15       call MPI_SEND(part_likely, 1, MPI_DOUBLE_PRECISION,
     &         0, my_rank, MPI_COMM_WORLD, ier)
16       call MPI_SEND(part_gradient, 25, MPI_DOUBLE_PRECISION,
     &         0, my_rank, MPI_COMM_WORLD, ier)
17       call MPI_SEND(part_hessian, 25*25, MPI_DOUBLE_PRECISION,
     &         0, my_rank, MPI_COMM_WORLD, ier)
18       goto 4
19    endif
```

```
! --- LISTING 5: SUBROUTINE CALC_LIKELY
 1    subroutine calc_likely(part_likely, part_gradient,
     &           part_hessian,estimflag,index,parm)
 2    implicit none
 3    real*8 part_likely, part_gradient(25), part_hessian(25,25),
     &       parm(25),p,dp(25)
 4    integer estimflag(25),index(2),j,k,iper,l,m
 5    part_likely = 0.0d0
 6    do i = 1, 25
 7       part_gradient(i) = 0.0d0
 8       do j = 1, 25
 9          part_hessian(i,j) = 0.0d0
10       end do
11    end do
```

```
      do iper = index(1), index(2)
         call get_probability(beta, iper, p, dp)
         do l = 1, 25
            if (estimflag(l).eq.1) then
               part_gradient(l) = part_gradient+dp(l)/p
               do m = 1, 25
                  if(estimflag(m).eq.1)then
                     part_hessian(l,m) = part_hessian(l,m)
     &                                 + ((dp(l)/p)*(dp(m)/p))
                  endif
               end do
            endif
         end do
      end do
      return
      end
```

```
! --- LISTING 6: CREATING A USER DEFINED DATATYPE
!
      program datatype
      implicit none
      include 'mpif.h'
      integer numprocs, my_rank, ier, status(MPI_STATUS_SIZE),
     &        array_of_blocklengths(3), array_of_types(3),
     &        array_of_displacements(3), extent, newdatatype
      real*8 part_likely, part_gradient, part_hessian
      common / partials / part_likely, part_gradient(25),
     &                    part_hessian(25,25)
      call MPI_INIT(ier)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ier)
      call MPI_COMM_RANK(MPI_COMM_WORLD,my_rank,ier)

      array_of_blocklengths(1) = 1
      array_of_blocklengths(2) = 25
      array_of_blocklengths(3) = 25*25

      array_of_displacements(1) = 0
      call MPI_TYPE_EXTENT(MPI_DOUBLE_PRECISION, extent, ier)
      array_of_displacements(2) = extent
      array_of_displacements(3) = extent + extent*25

      array_of_types(1) = MPI_DOUBLE_PRECISION
      array_of_types(2) = MPI_DOUBLE_PRECISION
      array_of_types(3) = MPI_DOUBLE_PRECISION
      call MPI_TYPE_STRUCT(3, array_of_blocklengths,
     &     array_of_displacements, array_of_types, newdatatype,
     &     ier)
      call MPI_TYPE_COMMIT(newdatatype, ier)

      if ( my_rank .eq. 0 ) then

         call MPI_RECV(part_likely, 1, newdatatype, MPI_ANY_SOURCE,
     &        MPI_ANY_TAG, MPI_COMM_WORLD, ier)
      else

         call MPI_SEND(part_likely, 1, newdatatype, 0, my_rank,
     &        MPI_COMM_WORLD, ier)

      endif

      call MPI_TYPE_FREE( newdatatype, ier)
      call MPI_FINALIZE( ier )
      stop
      end
```

Notes

1 Parallel computing is widely used in other disciplines as well. For example, Ansorge et al. (2000) discuss the application of parallel computing to the design of aMagnetic Imaging Resonance system; Adeli (2000) provides a survey of the use of parallel computing in engineering; and Mineter and Dowers (1999) discuss its application in Geographical Information Systems.

2 Nagurney 1996 provides an overview of parallel computing and its application to economic problems.

3 The PVM homepage is located at http://www.epm.ornl.gov/pvm, and Geist et al. (1994) is a helpful reference for PVM.

4 Recently the NOW model has been refined to a new class of supercomputer known as 'Beowulf' clusters. A Beowulf is a cluster of off-the-shelf PCs running a freely available unix-like operating system such as Linux and using freely available compilers and message passing libraries. Much of the software for the Beowulf project was developed at the Goddard Space Flight Center (http://beowulf.gsfc.nasa.gov/). Sterling et al. (1999) provides an overview of Beowulf design and implementation.

5 See http://www.ERC.MsState.Edu/labs/hpci/projects/mpi/ for more information onMPI. Gropp, Lusk, and Skjellum (1999) provides an introduction to MPI, and Snir et al. (1996) is the standard reference guide (available at www.netlib.org).

6 httpi://www.lam-mpi.org.

7 http://www-unix.mcs.anl.gov/mpi/mpich.

8 There is an exception to MPI_INIT being the first MPI call; the subroutine MPI_INITIALIZED can be called to see if MPI_INIT has been called. See Gropp, Lusk, and Skjellum (1999) for more information.

9 In MPI-1, it is not possible to add or delete processes from a running job. This feature is present in PVM and is part of MPI-2.

10 Programs can be written so that the master process also does part of the work. To keep things simple, I assume that the master process does none of the computational work.

11 The variables MPI_SOURCE and MPI_TAG are used for the locations because the MPI specification does not dictate which information resides in which location. Using MPI_SOURCE and MPI_TAG assures that code is portable across implementations of MPI.

12 Gropp, Lusk, and Skjellum (1999) is an excellent introduction toMPI; Gropp, Lusk, and Thakur (1999) focus on the new MPI-2 standard; and Snir et al. (1996) provides the complete MPI-1 specification. This book is also available at http://www.netlib.org/utk/papers/mpi-book/mpi-book.html and as a postscript file.

13 The AFDC program was a means-tested welfare program that provided cash assistance to single parent families (and a small number of two parent families) in the United States from 1935 to 1996. Moffitt (1992) provides an excellent survey of the AFDC literature.

14 For this simple exercise, I ignore the fact that childless women are not eligible for AFDC.

15 The PSID is a large, U.S. panel data set that began in 1968. Its homepage is located at http://www.isr.umich.edu/src/psid/.

16 For more information about sample construction, see Swann (1996).

17 To save notation, many values that might normally be left as variables (e.g. the number of observations) are hard-coded in the listings. To save space, I have omitted any comments from the listings as well.

18 The program uses the outer product of the gradient as an approximation to the matrix of second derivatives.

19 The outer product of the gradient is used to compute the standard errors of the coefficients.

20 One's ability use this technique depends on the amount of data that must be stored. In some cases it might be faster to pass the relevant portions of the data (if the worker machines were memory constrained for example).

21 I use the term 'member' to refer to a scalar or array listed in a common block and the term 'element' to refer to a specific array entry. Thus the common block example has two members and each member has one element.

22 While there would be no reason to send more than one copy of a common block, there are other user defined datatypes where more than one element may be sent.

23 The time calculation *T*1 should be done with the best serial implementation which is not necessarily the parallel version with numprocs = 1.

24 Amdahl's Law (Amdahl 1967) provides an upper bound on speedup. Let *fs* be the fraction of the program that is inherently serial and cannot be parallelized. Amdahl's Law states that the maximum speedup is 1/*fs* .

25 Even though the machine has two processors, the serial version uses only one processor. The timings are taken after the data and parameters are read and after the parameters have converged. The FORTRAN compiler pgf77 from The Portland Group was used with the optimization flags –fast -Mvect for all of the runs discussed.

26 For this example I am computing speedup based on the number of worker processes (or processors) rather than the number of total processes.

27 The profiling tool XMPI (discussed in section 8.2) shows that the two processes running on the Pentium Pro node spend more time waiting for data than the processes running on the PIII node (where the master is located). A faster networking set-up will reduce the wait time and improve performance.

28 Timings were begun in the master routine immediately before broadcasting the data and concluded prior to sending the message to the workers to exit. Thus the overhead to create the user-defined datatype is not included in the timing. The rationale for not including this overhead is that it is a small component of run-time for realistic applications.

29 httpi://www.lam-mpi.org/software/xmpi

30 Jumpshot is available as part of the mpich distribution at http://www-unix.mcs.anl.gov/mpi/mpich/.

31 http://www.nas.nasa.gov/Groups/Tools/Projects/AIMS/.

32 The program could of course be modified to be robust to this possibility.

33 http://www.netlib.org/scalapack/index.html.

References

Adeli, H. (2000). High performance computing for large-scale analysis, optimization, and control. *Journal of Aerospace Engineering*, **13**, 1–10.

Amdahl, G.M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings,* 483–485.

Ansorge, R.E., Carpenter, T.A., Hall L.D., Shaw N.R. and Williams G.B. (1999). Use of parallel supercomputing to design magnetic resonance systems. *IEEE Transactions on Applied Superconductivity*, **10**, 1368–1371.

Atkinson, A.C. (1994). Fast very robust methods for the detection of multiple outliers. *Journal of the American Statistical Association,* **89** (428), 1329–1339.

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

Coleman II, W.J. (1993). Solving nonlinear dynamic models on parallel computers. *Journal of Business & Economic Statistics*, **11** (3), 325–330.

Foster, I. (1995). *Designing and Building Parallel Programs*. Addison-Wesley.

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

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

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

LAM Team, *LAM Version 6.3.2*. University of Notre Dame, http://www.mpi.nd.edu/lam/6.3.

Liu, C. and Rubin, D.B. (1996). Markov-normal analysis of iterative simulations before their convergence. *Journal of Econometrics*, **75**, 69–78.

Mineter, M.J. and Dowers, S. (1999). Parallel processing for geographical applications: a layered approach. *Journal of Geographical Systems,* **1**, 61–74.

Moffitt, R. (1992). Incentive effects of the U.S. welfare system: a review. *Journal of Economic Literature*, **30**, 1–61.

Nagurney, A. (1996). Parallel computation. In H.M. Amman, D.A. Kendrick, J. Rust (eds.),*Handbook of Computational Economics, Volume 1*. Elsevier.

Nagurney, A. and Eydeland, A. (1992). A splitting equilibration algorithm for the computation of large-scale constrained matrix problems: theoretical analysis and applications. In H. Amman, D. Belsley and L. Pau (eds.), *Computational Economics and Econometrics*. Kluwer Academic Press.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. (1996). *MPI: The Complete Reference*. The MIT Press.

Sterling, T.L., Salmon, J., Becker, D.J., and Savarese, D.F. (1999). *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. The MIT Press.

Swann, C.A. (1996). A dynamic analysis of marriage, labor force participation, and participation in the AFDC program. PhD Dissertation, University of Virginia.