Parallelizing Econometric Computations: MPITB for GNU Octave

Michael Creel

18th November 2004

1 Introduction

The use of parallel computation within economics and econometrics in particular has a relatively long history[1], but it is clear that a relatively small part of the computational work done in economics makes use of parallel computing. Within research that uses computational methods, the set of problems that researchers are willing to investigate at a point in time is determined in part by the amount of computational time that is needed to solve the problems. Parallel computing offers the possibility of solving more computationally challenging problems in a reasonable time frame, relative to ordinary serial computing. However, the adoption of a tool depends not only upon its efficacy, but also upon the effort and time needed to master its use. It seems that parallel computing tools have historically had too steep a learning curve to allow for their adoption by a significant proportion of the general body of researchers. This has the effect of limiting the computational complexity of the set of problems that are on the mainstream research agenda.

This paper hopes to make four main points:

1. parallel computing implemented as distributed computing on a cluster of commodity computers can offer important reductions in the time to complete computations
2. some econometric problems of central interest are "embarassingly parallelizable"
3. high-level, interpreted matrix programming languages[2] can be extended to make use of parallel computing

[1]See Nagurney (1996); Doornik, *et. al.* (2002 and forthcoming); and Stern (2002) for citations of applied work that uses parallel computing.

[2]Examples are MATLAB (TM, the Mathworks, Inc.), Ox (TM OxMetrics Technologies, Ltd.) and GNU Octave (`www.octave.org`).

4. parallelization can in some cases be hidden from end users

None of these points is new to this paper, but this paper explores their intersection with a focus somewhat different from that of previous work.

Points 2 and 3 will be addressed below, but the other two points deserve comment here. With regard to the first point, Moore's Law is an empirical regularity that predicts that the performance of CPU's grows at an annual rate of roughly 50%. In order to achieve a 10-fold improvement in computational performance of a certain class of computer, one would need to wait nearly 5 years and then purchase a new computer of a comparable class. This paper shows with examples that a 10 fold improvement in performance can be achieved immediately, using distributed parallel computing and a high-level interpreted matrix programming language. This allows the research agenda on computationally complex problems to accelerate beyond what would be possible using serial computing.

With regard to the fourth point, hiding parallelization is probably essential to making use of parallel computing attractive to ordinary users of high level matrix programming languages. The examples of how this may be done are probably the most important contribution of this paper. Of course, someone must do the underlying parallel programming, though. Swann (2002) notes that the decision about whether or not a program should be written to use parallel computing depends both upon the speedup that can be obtained as well as upon the added programming time needed to implement parallelization. An additional point is how often the final program is to be used. If it is possible to parallelize algorithms and methods of general interest that may be used many times by many people, and if the implementation is such that end users do not have to deal with the underlying parallelization, then the programming effort of a small body of people who know how to write parallel programs can be benefitted from by a much larger group of end users. There are economies of scope in such cases. The paper shows that estimation by maximum likelihood (ML) or generalized method of moments (GMM) and Monte Carlo simulations can all be transparently parallelized. Such tasks clearly have a wide scope.

While most readers will probably be interested in observing the speed gains that are possible, and in knowing how to use the programs, some readers will be interested in how the parallelization is done, so that they may write their own code. The ML example is discussed in detail for these last readers.

2 Software environment

This paper focuses on examples from econometrics, but the methods could certainly be applied to other areas in economics and other disciplines. Econometric computations in research work are commonly done using high-level interpreted matrix programming languages such as MATLAB, GAUSS (TM Aptech Systems, Inc.), Ox, Octave, and others. These languages offer an intuitive programming syntax, and quite good performance of code that is vectorized. There is also a smaller but nevertheless considerable body of work that uses compiled languages such as FORTRAN and C. Compiled languages usually offer better performance than interpreted languages, depending upon the extent to which code can be vectorized. However, interpreted languages usually provide means to dynamically link to C or FORTAN code, so that bottlenecks[3] in interpreted code can be removed. Thus, interpreted languages are popular even with researchers who need the speed that C and FORTRAN can provide.

The Message Passing Interface (Message Passing Interface Forum, 1997) is a specification of a mechanism for passing instructions and data between the nodes of a cluster of computers. This specification has been implemented in a number of packages, including LAM/MPI (LAM team, 2004) and MPICH (Gropp, *et. al.*, 1996). These packages provide libraries of C and FORTRAN functions, along with support programs to use the functions. To make direct use of the libraries, one must program in C, C++, or FORTAN.

MPI capabilities can be brought to interpreted matrix programming languages through the dynamic linking capabilities of these languages. The library functions of the MPI packages can be incorporated in dynamic extensions, in the ordinary way the languages link to C, C++ or FORTRAN code. This has been done using the LAM/MPI implementation of MPI to create the MPI Toolbox (MPITB) by Fernández Baldomero *et. al.* (2002 and 2004) for the MATLAB and GNU Octave languages, respectively. These packages provide bindings for almost all of the MPI-1.2 specification, and for some of the MPI-2 specification (in particular, MPI_Comm_spawn is supported). Doornik *et. al.* (2002 and undated) have developed bindings to a subset of the MPICH implementation of the MPI-1.2 specification for the Ox language. It appears that MPITB is at the moment the most complete and functional set of MPI bindings for a high-level interpreted matrix programming language.

This paper uses MPITB for GNU Octave for its examples. This choice is motivated primarily by its completeness and functionality, and by the fact that both MPITB and GNU Octave are

[3]Loops are the most common cause of performance bottleneck in interpreted languages.

"free" software, even for commercial users. This means that their source code is available and modifiable, subject to some restrictions which require that new software that incorporates parts of MPITB or Octave must also be made freely available. This licensing guarantees that development of the software can always be continued by anyone who is interested in doing so. The availability of the source code for MPITB is also interesting in that it serves as an example of how bindings could be written for new implentations of the MPI standard.[4] The rest of this section briefly describes the rest of the software environment that is used to obtain the results presented in the subsequent sections.

2.1 GNU Octave

GNU Octave is a freely available high-level interpreted matrix programming language that has a syntax that is mostly compatible with MATLAB's. Programs written in MATLAB will usually run directly in Octave, though they sometimes need minor editing. It may be also be the case that a program may depend on third-party software for MATLAB that is not available for Octave. Nevertheless, Octave provides a foundation for programming that is essentially equivalent to that provided by MATLAB, and as such, it should be clear that Octave is a language suitable for doing econometrics.[5] There is no reason that an active body of users could not develop a well-rounded set of econometric extensions similar to those that exist for MATLAB, GAUSS and Ox, for example. The `octave-forge` package available at `http://octave.sourceforge.net/` provides many extensions and applications, some of which are useful for econometrics. The code it contains is also freely available (most of it is licensed according to the GNU General Public License). In particular, the minimization functions and serial versions of the ML and GMM functions that are used below are contained in the `octave-forge` package. Both Octave and the `octave-forge` extensions will run under the Windows, Linux and Mac OS X operating systems.

2.2 ParallelKnoppix

The parallel versions of the example programs for econometrics require a computing environment that supports MPI-based parallel processing. Two possibilities are to use a single symmetric multiprocessor (SMP) computer, or to use a cluster of computers for distributed parallel

[4]The Open MPI project at `http://www.open-mpi.org/` is a very promising new implementation of MPI-2. It would be desirable to have Octave bindings to this implementation's functions. MPITB could serve as an example to anyone who would like to do this.

[5]Eddelbuettel (2000) discusses the use of Octave for econometric work.

processing. The SMP solution is simple to use, since software only has to be installed on a single computer, but the speedup that can be obtained is limited by the number of processors the machine has.

The distributed solution has the advantage that many processors may be accessed. Universities often have large arrays of ordinary desktop computers available for the use of student. ParallelKnoppix (Creel, 2004) is a bootable CD that allows creation of a Linux cluster for MPI parallel processing on a network of computers of the IA-32 architecture (*e.g.,* Intel Pentium or Xeon, or AMD Athlon or Duron computers) in roughly ten minutes. The cluster can be made up of homogeneous or heterogeneous computers, and they need not have Linux installed. When the cluster is shut down, the machines are in their original state[6], so use of the machines for clustering does not interfere with their ordinary use. ParallelKnoppix may be modified to add software packages and data. A version of ParallelKnoppix was created that contains LAM/MPI, Octave, MPITB and all the program examples discussed below. This CD was used to generate the results that are reported below, and using it is the most convenient means of replicating the results.[7]

3 Three example problems: description

This section introduces the three example problems, and shows that they are "embarrassingly parallelizable", which simple means that parallelization is relatively easy to implement.

3.1 Monte Carlo

A Monte Carlo study involves repeating a random experiment many times under identical conditions. In this paper, we assume that the experiment to be repeated can be written as a function of some input arguments, and that the result is a vector of outputs. Doornik *et. al.* (2002) argue that a Monte Carlo experiment done on a cluster should give numerically identical results to those of the experiment done on a single computer. My opinion is that Monte Carlo experiments should always be repeated enough times to verify that the set of random draws used in a given trial does not influence the results in any important way. If this is done, then it is not necessary to control the values of the random draws in individual experiments. The need to repeat the experiment

[6]This is true except for the computer the CD is booted on, where a working directory is created. This computer may be returned to its original state by deleting the working directory. Or the working directory may be left on the computer so that it is available for use in future sessions.

[7]An image of this CD is available at `http://pareto.uab.es/mcreel/ParallelKnoppix/mpitb_paper.iso`. This image is about 600 MB in size.

several times does add interest to the possibility of making the experiment execute more rapidly, though. In this paper, interest centers on how to make the experiment run in parallel, rather than on the issue of how random numbers are generated. In what follows, we provide an example where each run gives results that at numerically slightly different.[8]

Listing 1 shows a function that generates data according to the classical linear regression model, finds the OLS estimator of the coefficients and the error variance, and returns the result. This function is illustrative of the format that will be required for Monte Carlo simulation of a function: it receives a single argument, and it returns a row vector that holds the results of one random simulation. The single argument in this case is a cell array that hold a fixed set of regressors in its first position, and a fixed true coefficient vector in the second position. It generates a random result though a process that is internal to the function, and it report some output in a row vector. Subsequent calls to this function are independent of one another, and clearly can be executed on different processors. A set of R calls to this function is obviously parallelizable.

```
 function output = olsmc(f_args)
       # generate data according to classical model
       x = f_args{1};
       theta = f_args{2};
       epsilon = randn(rows(x),1);
       y = x*theta + epsilon;
       # estimate by OLS
       [b, sigsq] = ols(y,x);
       # results of interest
       output = [b’, sigsq];
endfunction
```

Listing 1: DGP for OLS Monte Carlo

[8]The ParallelKnoppix CD mentioned above contains a Monte Carlo function that ensures that a given set of random draws is used. Results are comparable to those in the paper, but are omitted to save space.

3.2 ML

For a sample $\{(y_t, x_t)\}_n$ that represents n observations on a set of dependent and explanatory variables, the maximum likelihood estimator of the parameter θ^0can be defined as

$$\hat{\theta} = \arg\max s_n(\theta)$$

where

$$s_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} \ln f(y_t|x_t, \theta)$$

Here, y_t may be a vector of random variables, and the model may be dynamic since x_t may contain lags of y_t. As Swann (2002) points out, this can be broken into sums over blocks of observations, for example two blocks:

$$s_n(\theta) = \frac{1}{n}\left\{\left(\sum_{t=1}^{n_1} \ln f(y_t|x_t, \theta)\right) + \left(\sum_{t=n_1+1}^{n} \ln f(y_t|x_t, \theta)\right)\right\}$$

Analogously, we can define up to n blocks. Again following Swann, parallelization can be done by calculating each block on separate computers.

3.3 GMM

For a sample as above, the GMM estimator of the K-dimensional parameter θ^0 can be defined as

$$\hat{\theta} \equiv \arg\min_{\Theta} s_n(\theta)$$

where

$$s_n(\theta) = m_n(\theta)' W_n m_n(\theta)$$

and

$$m_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} m_t(y_t|x_t, \theta)$$

is a g-vector, $g \geq K$, with $\mathcal{E}_\theta m_n(\theta) = 0$, and W_n converges almost surely to a finite $g \times g$ symmetric positive definite matrix W_∞. Since $m_n(\theta)$ is an average, it can obviously be computed blockwise, using for example 2 blocks:

$$m_n(\theta) = \frac{1}{n}\left\{\left(\sum_{t=1}^{n_1} m_t(y_t|x_t, \theta)\right) + \left(\sum_{t=n_1+1}^{n} m_t(y_t|x_t, \theta)\right)\right\}$$

Likewise, we may define up to n blocks, each of which could potentially be computed on a different machine.

Thus we see that the three problems have a structure that can easily be parallelized.

4 Hidden parallelization

This section illustrates that the example problems can be executed in parallel transparently to the user. Functions that implement the three examples serially and in parallel are called, and the results are compared.

4.1 Monte Carlo

Listing 2 show an Octave script that executes a Monte Carlo study of the OLS estimator, using the function listed in Listing 1. The main thing to notice about this script is that lines 10 and 14 call the function `monte_carlo`. The last argument of this function is the number of slave hosts to use[9]. We see that running the Monte Carlo study on one or more processors is transparent to the user - he or she must only indicate how many processors are to be used.

```
1  # number of monte carlo replications
2  reps = 10000;
3  # size of data matrix
4  n = 30;
5  k = 3;
6  # true coefficients and fixed data matrix
7  theta = ones(k,1);
8  x = [ones(n,1) rand(n,k-1)];
9  # run on master
10 output = montecarlo("olsmc",{x, theta}, reps, 0);
11 mean_s = mean(output)';
12 var_s = var(output)';
13 # run on master and one slave
14 output = montecarlo("olsmc",{x, theta}, reps, 1);
15 mean_p = mean(output)';
16 var_p = var(output)';
```

[9]The total number of processors used to obtain the results is the number of slaves plus one, the processor that runs the original instance of Octave.

```
printf("Monte Carlo results: mean of %d replications\n", reps);
# print results
labels = str2mat("Mean (s)", "Mean (p)", "Var. (s)", "Var. (p)");
prettyprint_c([mean_s mean_p var_s var_p], labels);
```

Listing 2: OLS Monte Carlo

Running this last script gives the output in Listing 3. There is a bit of a difference between the serial and parallel results, which indicates that 10,000 replications is perhaps not enough for acceptable precision. Note that the regressor matrix is fixed across replications, which is why the variances of the coefficients are different.

```
Monte Carlo results: mean of 10000 replications
  Mean (s) Mean (p) Var. (s) Var. (p)
     1.000    0.998    0.318    0.312
     0.989    0.998    0.395    0.388
     1.014    1.006    0.587    0.581
     1.001    1.000    0.074    0.074
```

Listing 3: OLS Monte Carlo Results

4.2 ML

Listing 4 shows an Octave script that calculates the maximum likelihood estimator of a parameter. The data is read, the name of the density function is provided in the variable `model`, and some controls are set. In line 7, the function `mle_estimate` performs ordinary calculation of the ML estimator using the computer from which the script is run, while in line 9 the function `mle_estimate_parallel` does the estimation using one or more computers, where the last argument of the function is the number of slave computers, in this case 1. A person who runs the program sees no parallel programming code - the parallelization is transparent to the end user, beyond having to select the number of slave computers.

```
1 load data; # load the data
2 control = {Inf,0,1,1}; # controls for minimization algorithm
3 model = "NegBinSNP"; # name of function that calculates loglikelihood
4 modelargs = {1};      # other arguments
5 theta = [zeros(columns(data-1),1);0;0;0]; # starting values
```

```
# Estimation: serial
theta_s = mle_estimate(theta, data, model, modelargs, control);
# Estimation parallel
theta_p = mle_estimate_parallel(theta, data, model, modelargs, control, 1);
printf("MLE estimation results: NegBinSNP, %d observations\n", rows(data));
# print results
labels = str2mat("serial", "parallel");
prettyprint_c([theta_s theta_p], labels);
```

Listing 4: Script to perform ML estimation

The result of running this script is in Listing 5. We see that the results are the same.

```
MLE estimation results: NegBinSNP, 7930 observations
    serial  parallel
     1.559    1.559
     0.007    0.007
     0.007    0.007
     0.011    0.011
    -0.006   -0.006
     0.173    0.173
     0.176    0.176
    -0.019   -0.019
     0.039    0.039
     1.612    1.612
    -4.453   -4.453
     1.366    1.366
    -0.000   -0.000
```

Listing 5: MLE Example - Results

4.3 GMM

Listing 6 shows an Octave script that performs GMM estimation of a parameter. Lines 1-7 read data and define starting values, the weight matrix, and other details. In line 9, `gmm_estimate` is called to do the estimation, serially. Line 11 defines the number of slave computers to use, and line 12 calls a parallelized version of the estimation function. Again, the main point in this section is to note the similarity of the way estimation is done in parallel to the way it is done serially. The user interface is virtually identical except that the number of slave processors must be provided.

```
load data; # read data
k = 5; # number of regressors
theta = zeros(k,1); # start values
weight = eye(columns(data) - 1 - k); # weight matrix
moments = "poisson_iv_moments"; # name of function that calculates moments
momentargs = {k}; # additional information needed to calculate moments
control = {Inf,0,1,1}; # bfgs controls
# gmm estimation: serial
theta_s = gmm_estimate(theta, data, weight, moments, momentargs, control);
# gmm estimation: parallel
nslaves = 1;
theta_p = gmm_estimate_parallel(theta, data, weight, moments, momentargs, control,
    nslaves, 0);
printf("GMM estimation results: %d simulated observations\n", rows(data));
# print results
labels = str2mat("serial", "parallel");
prettyprint_c([theta_s theta_p], labels);
```

Listing 6: Script to perform GMM estimation

Listing 7 shows the output from the script in Listing 6. We see that the serial and parallel version obtain the same results.

```
GMM estimation results: 1000 simulated observations
    serial  parallel
    -0.068   -0.068
     0.954    0.954
     1.144    1.144
     1.021    1.021
     1.092    1.092
```

Listing 7: GMM Example - Results

This section has given examples that show that the three problem under consideration can be parallelized, and that end users can make use of the parallelized routines in more or less the same way as they use the ordinary serial routines. Next we provide some results that show that MPITB for Octave can provide interesting speedups for the three problems.

5 The speedup from parallelization

This section provides timing results for the three problems. We use two computing environments. The first is a single SMP machine with two Pentium IV 3.06GHz Xeon CPUs, each with 512KB level 2 cache, and 2GB of RAM. Hyperthreading is enabled, so the SMP machine has 4 "virtual" CPUs. Testing on the SMP machine is done in an ordinary desktop environment, with a light load due to other running processes. The results are mean to be indicative of what a user might experience when using MPITB on a SMP desktop computer. The second test environment is a Linux cluster created with ParallelKnoppix in a university computer room which is ordinarily used by students for their work. The nodes are homogeneous uniprocessor Pentium IV machines running at 2.8 GHz, each with 1MB of level 2 cache, 512MB of RAM, with hyperthreading enabled. Each machine has a 3COM 3c905 Tornado network card, and they are connected on a 100MB/s ethernet network using a 3COM OfficeConnect dual speed switch. A working directory that contains MPITB and all needed programs is shared by NFS among the nodes of the cluster. For the runs on the cluster, there was no CPU load additional to that of the test programs, other than the basic overhead of the operating system. Thus, these results are more reliable for analyzing performance.

5.1 Monte Carlo

For testing the serial and parallel performance of the `montecarlo` function, we use the same trace test example as do Doornik, *et. al.* (2002). This is a Monte Carlo simulation of the sampling distribution of a test for the existence of cointegration[10]. Since the hardware used for the test is different from what they used, timings are not comparable, but it is perhaps interesting to use the same example to show how the software implementation differs between Ox, using the `MPI.DLL` extension, and Octave using MPITB. The `tracetest` function for Octave appears in Listing 8. Monte Carlo simulation of this function is done by running the program that appears in Listing 9. These two listings may be compared with Doornik *et. al.*'s Listings 2 and 5, which are functionally similar.

[10]See Doornik, *et. al.* (2002) for motivation and references.

```
function test_stat = tracetest(args)
 t = args{1};
 n = args{2};
 e = randn(t,n);
 p = inv(chol(e'*e/t));
 e = e*p;
 s = lag(cumsum(e),1);
 s(1,:) = s(1,:) - s(1,:); # lag fills with 1, test needs 0
 fac = e'*s;
 ev = eig(fac*inv(s'*s)*fac'/t);
 test_stat = -t*sum(log(1 - ev/t));
endfunction
```

Listing 8: Trace Test function

```
outfile = "SMP-";
maxslaves = 1;
T = 1000;
dim = 5;
reps = 100000;
for nslaves = 0:maxslaves;
  tic;
  montecarlo("tracetest", {T,dim}, reps, nslaves);
  t = toc;
  results(nslaves+1,:) = [nslaves, T, dim, reps, t];
endfor
eval (sprintf("save \"%stracetest-%d-%d-%d_results.out\" results", outfile, T, dim,reps))
    ; # print timing results to file
```

Listing 9: Monte Carlo of Trace Test

The timings for the Monte Carlo runs are found in Tables 1 and 2, column 2. We define the lower limit to the runtime in parallel as the runtime of the serial version, divided by the number of nodes used for the parallel run. This is a lower limit, since it ignores the fact that a program may have non-parallelizable portions, and it implicitly assumes that communication between the parallelized portions is done at no cost. Figure 1 reports the actual runtime and the lower limit for the Monte Carlo study of the `tracetest.m` function. We can see that the parallel version is very efficient, achieving a runtime close to the lower limit. This is due to the underlying efficiency of LAM/MPI when using TCP communication, and to the fact that the MPITB implementation of the binding functions is efficient, as well as to the fact that the `montecarlo.m` function is emminently parallelizable, with very little communication overhead and virtually no non-parallelizable code. Table 3 reports the speedup (serial runtime divided by parallel runtime) and efficiency (speedup divided by number of nodes) for this problem as well as for those discussed immediately below. We see (in column 2) that the speedup is almost equal to the number of nodes used in the run, and (in column 5) that efficiency is very high.

5.2 ML

To test the performance of maximum likelihood estimation in serial and parallel, consider estimation of a reshaped negative binomial density for count data. The negative binomial base density is multiplied by a squared polynomial, then normalized to sum to one. References that explain the approach are Gallant and Nychka (1987), Cameron and Johannson (1997) and Guo and Trivedi (2002). Since this specific likelihood function is used only to provide an example of computational gains, we do not go into details here. The reshaped density is

$$f_Y(y|\psi,\lambda,\gamma) = \frac{\left[h_p\left(y|\gamma\right)\right]^2}{\eta_p(\phi,\gamma)} \frac{\Gamma(y+\psi)}{\Gamma(y+1)\Gamma(\psi)} \left(\frac{\psi}{\psi+\lambda}\right)^{\psi} \left(\frac{\lambda}{\psi+\lambda}\right)^{y},$$

where

$$h_p\left(y|\gamma\right) = \sum_{k=0}^{p} \gamma_k y^k, \tag{1}$$

and

$$\eta_p(\psi,\lambda,\gamma) = \sum_{k=0}^{p}\sum_{l=0}^{p} \gamma_k\gamma_l m_{k+l}(\psi,\lambda) \tag{2}$$

In this last equation, the $m_{k+l}(\psi,\lambda)$ are the raw moments of the negative binomial density. Since this double sum leads to very long analytic expressions when p is at all large, evaluation of the density is relatively computationally intensive, and it may benefit from parallelization. This is the problem that is estimated in Listing 4. A modified version of that script was run, that uses additional slave nodes.

For maximum likelihood estimation, two parallelized functions were written, `mle_estimate_parallel` and `mle_estimate_parallel_alt`. These use two different implementations in the way they pass messages from the slaves to the master node. The `mle_estimate_parallel` version passes the entire vector of log likelihood values for the block of data that is evaluated on each of the slave nodes back to the master computer. Then the master computer assembles all these values into a vector, and averages it to obtain the average log likelihood value for the trial value of the parameter. In the course of maximization of the log likelihood function, this version follows exactly the same solution path as does the ordinary serial function `mle_estimate`, from `octave-forge`, since the value of the loglikelihood function is exactly the same at any given point during the BFGS iterations. The `mle_estimate_parallel_alt` version passes only the sum of the log likelihoods of each block back to the master computer. This is more efficient in terms of communication overhead, since much less data flows from the slave nodes back to the master node. However, the average log likelihood calculated by summing these partial sums, then dividing by the sample size, is not numerically exactly equal to the average of all the individual loglikelihood contributions, as obtained using `mle_estimate` or `mle_estimate_parallel`, due to fact that floating point values are stored with a given numerical precision. For this reason, the `mle_estimate_parallel_alt` version of the parallel implementation takes a solution path during the course of BFGS iterations that is not the same as that of the ordinary serial version, and furthermore, the path taken depends upon how many slave computers are used. Since determining the direction of search and the stepsize at each iteration require a number of evaluations of the loglikelihood function, even very small differences in the numerical value of the objective function at a given value of the parameter can cause the solution paths to differ, though they eventually converge to the same solution. With the alternative version, the number of iterations needed to satisfy the convergence criteria may differ for different numbers of slave computers. What is of fundamental importance, though, is not the exact path that is taken, but rather that the algorithms arrive to one of the set of local maxima, within a well-defined set of convergence tolerances. The time needed to do this is also important, of course.

The version that passes only the partial sums of the loglikelihood function (`mle_estimate_parallel_alt`) is about 15% faster for the test problem used here, as is expected. Unexpectedly, it often achieves convergence in fewer iterations than does the serial version or `mle_estimate_parallel`. This seems to be due to error cancelation when summing the partial sums that are returned from the slave nodes to the master node for summation. Cancellation of errors seems to cause the convergence criteria for the BFGS minimizer to be satisfied in fewer iterations. Both versions converge to the same solution. Here, we only report results for the faster version, which are in Table XX.[11]

5.3 GMM

To illustrate the speedup in GMM estimation that may be obtained using parallelization, we use an example of Efficient Methods of Moments (EMM) estimation (Gallant and Tauchen, 1996). This is a version of the method of simulated moments, where the moment conditions are the scores of a quasi-maximum likelihood estimator. Here we use a simple example where data is generated following a probit model, and a logit model is used to provide moment conditions. This problem is still somewhat computationally demanding, though, since the sample size is 2000 observations, there are 5 parameters, and 20 simulation replications are used. The function `emm_moments` appears in Listing 10. This is a general purpose function that receives the names of the functions that define the DGP and the auxiliary model (the "score generator") as elements of the third argument.

```
function scores = emm_moments(theta, data, momentargs)
 k = momentargs{1};
 dgp = momentargs{2}; # the data generating process (DGP)
 dgpargs = momentargs{3}; # its arguments (cell array)
 sg = momentargs{4}; # the score generator
 sgargs = momentargs{5}; # SG arguments (cell array)
 phi = momentargs{6}; # QML estimate of SG parameter
 y = data(:,1);
 x = data(:,2:k+1);
 # random draws passed in data to ensure fixed over estimation
 rand_draws = data(:,k+2:columns(data));
 n = rows(y);
 scores = zeros(n,rows(phi));
 reps = columns(rand_draws);
 for i = 1:reps
  e = rand_draws(:,i);
  y = feval(dgp, theta, x, e, dgpargs); # simulated data
  sgdata = [y x]; # pass simulated data to SG
  # get gradient of SG
  scores = scores + numgradient(sg, {phi, sgdata, sgargs});
```

[11] All programs, including the alternate version, are available from the author.

```
  scores = scores / reps; # average over number of simulations
 endfor
endfunction
```

Listing 10: EMM Moment Conditions

This is used to perform EMM estimation using generated data. The script in Listing 11 does the estimation. Note that in lines 31 and 34, `gmm_estimate_parallel` accepts `alt` as its last argument. This is a 0/1 switch that determines how message passing is done. When `alt`$=0$, the entire matrix of moment contributions for each block of data is passed from the slaves back to the master, which assembles them into a matrix, then averages them to obtain the vector $m_n(\theta)$. When `alt`$=1$, the slave pass back only the vector sum of the moment contributions, and the master then sums them up and divides by the sample size n, to obtain $m_n(\theta)$. This is more efficient in terms of communication overhead, but as in the case of ML estimation, the small differences in numeric values of the elements of $m_n(\theta)$ lead to a different solution path being taken during the course of the BFGS iterations.

```
1  outfile = "SMP-";
2  maxslaves = 1;
3  n = 2000; k = 5; reps = 20; # obs, params, replications
4  sg = "logit"; # the score generator
5  dgp = "probitdgp"; # the DGP
6  dgpargs = 1; # smoothing during simulations, to make objective fn differentiable
7  sgargs = {1}; # just a place holder in this example
8  control = {Inf,0,1,1}; # BFGS controls
9  theta = [-k; ones(k-1,1)]; # true parameter value
10 # generate Probit data
11 x = [ones(n,1) randn(n,k-1)];
12 e = randn(n,1);
13 y = feval(dgp, theta, x, e, 0);
14 data = [y x];
15 # QML estimation of score generator
16 phi = zeros(columns(x),1);
17 phi = bfgsmin("mle_obj",{phi, data, "logit", 0}, control);
18 moments = "emm_moments"; # define moments function for GMM
19 # arguments for emm_moments
```

```
momentargs = {k, dgp, dgpargs, sg, sgargs, phi};
# arguments for emm_estimate
theta = theta - theta; # start values
weight = eye(rows(phi)); # weight matrix
rand_draws = randn(n,reps); # fixed over iterations to avoid "chattering"
data = [data rand_draws];
results = zeros(maxslaves+1,5); # container for results
for nslaves = 0:maxslaves;
	r = nslaves;
	for alt = 0:1
		tic();
		[theta_hat, obj_value, convergence, iters0] = gmm_estimate_parallel(theta,
		    data, weight, moments, momentargs, control, nslaves, alt);
		m = feval(moments, theta1, data, momentargs);
		eff_weight = inverse(cov(m)); # get efficient weight matrix
		[theta_hat, obj_value, convergence, iters1] = gmm_estimate_parallel(theta,
		    data, eff_weight, moments, momentargs, control, nslaves, alt);
		t = toc();
		iters = iters0 + iters1;
		r = [r t iters];
	endfor
	results(nslaves+1,:) = r;
endfor
eval (sprintf("save \"%sLogitProbitEMM-%d-%d-results.out\" results", outfile, n, k,
    reps));
```

Listing 11: EMM Estimation

6 Writing parallel code: the ML example

MPITB provides the MPI bindings for Octave. It follows the LAM/MPI syntax, so function names, arguments and returns are all the same as if one were directly using the LAM/MPI C or FORTRAN libraries. Thus, any documentation for LAM/MPI will be useful. Swann (2002) provides a very useful discussion of using MPI for ML estimation using FORTAN, and discusses the relevant MPI functions in some detail. Here, we discuss the way ML estimation was done in parallel using Octave and MPITB, but do not go into great detail since it is essentially the same as what Swann covers.

We have seen that ML estimation is done by calling `mle_estimate_parallel`. Thi

7 Conclusion

place holder

Table 1: Timings, SMP

	Monte Carlo	MLE			GMM	
Soft	Time	Time	Time (alt)	Iters (alt)	Time	Time (alt)
1	363.6	191.5	185.8	29	559.4	559.4
2	209.0	112.4	121.0	31	448.2	391.1
3	209.3	92.3	87.2	28	425.7	400.9
4	213.2	79.6	79.4	29	447.2	423.2

Table 2: Timings, Cluster

	Monte Carlo	MLE			GMM	
Nodes	Time	Time	Time (alt)	Iters (alt)	Time	Time (alt)
1	291.0	177.8	179.2	34	760.4	760.4
2	147.9	102.0	80.9	28	443.6	382.4
3	101.4	74.6	76.4	37	334.9	265.2
4	76.3	60.9	51.9	31	284.6	207.9
5	61.4	52.4	44.7	31	256.0	173.7
6	51.5	47.1	39.4	30	237.1	152.0
7	43.1	43.9	33.3	27	223.7	137.6
8	38.9	42.2	34.0	29	213.7	126.3
9	34.8	40.4	33.8	30	207.1	117.7
10	31.4	40.3	32.8	30	202.0	111.4
11	28.7	39.4	33.6	31	197.9	106.9
12	26.5	39.2	30.1	28	195.1	103.2
13	24.4	39.6	32.0	30	186.9	96.3
14	22.9	39.4	30.2	28	na	na
15	21.4	40.2	32.8	30	na	na
16	20.1	40.1	33.1	30	na	na

1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
1.967183	1.809361	1.988513	0.983592	0.904680	0.994257
2.869034	2.533776	2.867153	0.956345	0.844592	0.955718
3.813271	3.125430	3.657031	0.953318	0.781358	0.914258
4.740840	3.624255	4.377911	0.948168	0.724851	0.875582
5.650051	3.985103	5.001937	0.941675	0.664184	0.833656
6.750879	4.236141	5.525056	0.964411	0.605163	0.789294
7.484439	4.463590	6.021446	0.935555	0.557949	0.752681
8.370752	4.645765	6.459166	0.930084	0.516196	0.717685
9.273175	4.785748	6.825442	0.927317	0.478575	0.682544
10.133127	4.831375	7.113459	0.921193	0.439216	0.646678
10.984220	4.869661	7.368107	0.915352	0.405805	0.614009
11.918903	4.897752	7.899982	0.916839	0.376750	0.607691

Table 3: Speedup and Efficiency, Cluster

	Speedup			Efficiency		
Nodes	Monte Carlo	MLE	GMM	Monte Carlo	MLE	GMM
1	1	1	1	1	1	1
2	1.97	1.81	1.99	0.98	0.90	0.99
3	2.87	2.53	2.87	0.96	0.84	0.96
4	3.81	3.13	3.66	0.95	0.78	0.91
5	4.74	3.62	4.38	0.95	0.72	0.88
6	5.65	3.98	5.00	0.94	0.66	0.83
7	6.75	4.24	5.53	0.96	0.61	0.79
8	7.48	4.46	6.02	0.94	0.56	0.75
9	8.37	4.65	6.46	0.93	0.52	0.72
10	9.27	4.79	6.82	0.93	0.48	0.68
11	10.13	4.83	7.11	0.92	0.44	0.65
12	10.98	4.87	7.37	0.92	0.41	0.61
13	11.91	4.90	7.90	0.92	0.38	0.61

Figure 1: Timings, Monte Carlo, Cluster

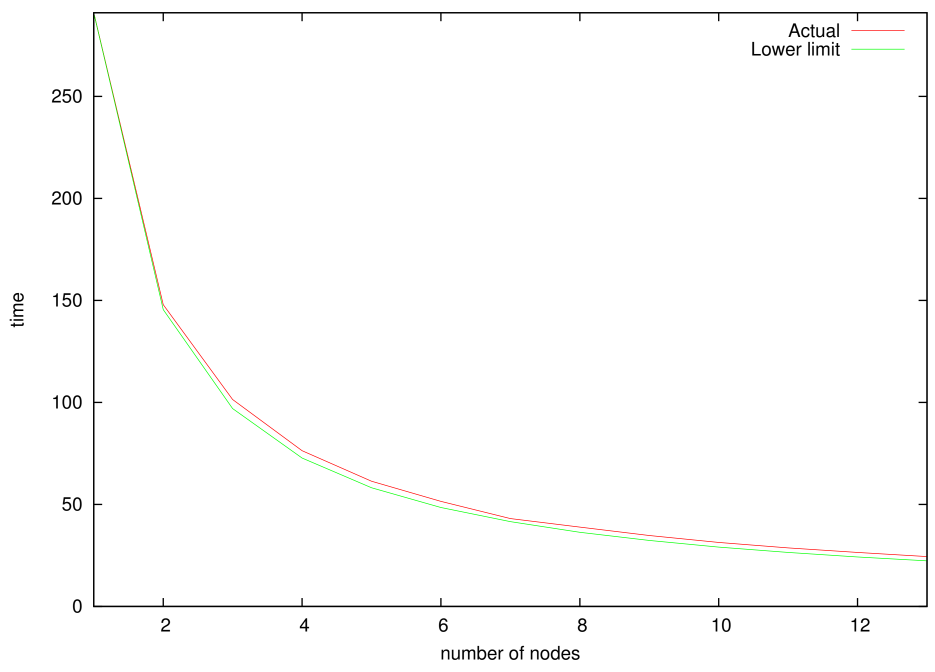


Figure 2: Timings, MLE, Cluster

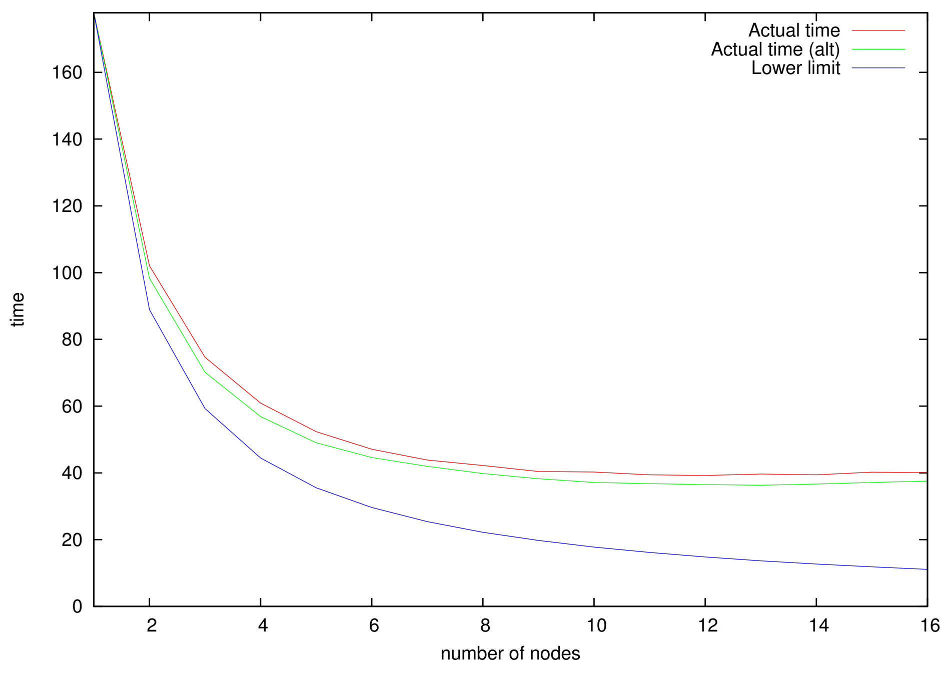
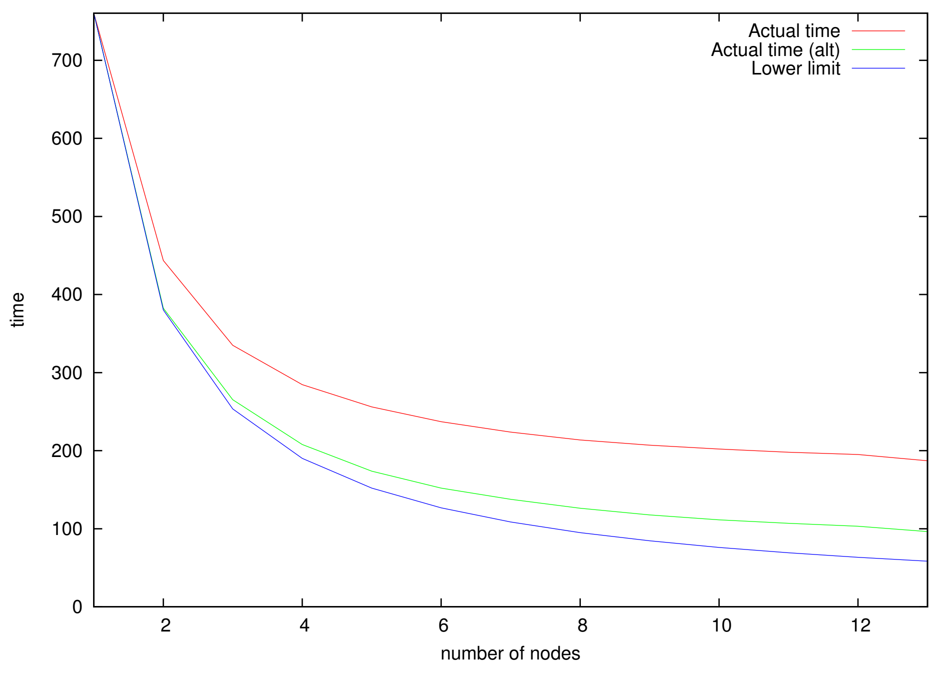


Figure 3: Timings, GMM, Cluster


Actual time
Actual time (alt)
Lower limit
time
number of nodes