User-Friendly Parallel Computations with Econometric Examples

Michael Creel*

April 15, 2005

Abstract

This paper shows how a high level matrix programming language may be used to perform Monte Carlo simulation, bootstrapping, estimation by maximum likelihood and GMM, and kernel regression in parallel on symmetric multiprocessor computers or clusters of workstations. The implementation of parallelization is done in a way such that an investigator may use the programs without any knowledge of parallel programming. A bootable CD that allows rapid creation of a cluster for parallel computing is introduced. Examples show that parallelization can lead to important reductions in computational time. Detailed discussion of how the Monte Carlo problem was parallelized is included as an example for learning to write parallel programs for Octave.

Keywords: parallel computing; Monte Carlo; bootstrapping; maximum likelihood; GMM; kernel regression

JEL classifications: C13; C14; C15; C63; C87

1 Introduction

Parallel computing can offer an important reduction in the time to complete computations. This is well-known, but it bears emphasis since it is the main reason that parallel computing may be attractive to users. To illustrate, the Intel Pentium IV (Willamette) processor, running at 1.5GHz, was introduced in November of 2000. The Pentium IV (Northwood-HT) processor, running at

*Department of Economics and Economic History, Edifici B, Universitat Autònoma de Barcelona, 08193 Bellaterra (Barcelona) Spain. *email*: michael.creel@uab.es; *Tel.:* 93-581-1696; *FAX*: 93-581-2012. I am very grateful to Javier Fernández Baldomero for his help in getting started with MPITB. This research was supported by grants SGR2001-0164 and SEC2003-05112.

3.06GHz, was introduced in November of 2002. An approximate doubling of the performance of a commodity CPU took place in two years. Extrapolating this admittedly rough snapshot of the evolution of the performance of commodity processors, one would need to wait more than 6.6 years and then purchase a new computer to obtain a 10-fold improvement in computational performance. The examples in this paper show that a 10-fold improvement in performance can be achieved immediately, using distributed parallel computing on available computers.

While it is well known that such speedups in computations are possible, and while the use of parallel computation within economics in general and econometrics in particular has a long history[1], it is also clear that only a small part of the computational work done in economics makes use of parallel computing. Two factors may explain this. Parallel programming has traditionally had a steep learning curve, at least compared to that of the popular high-level matrix programming languages. In the best of cases, programmers had to learn FORTRAN or C, and then how to use libraries of functions for parallelization. Next, parallel computing has traditionally been done on expensive mainframe computers that require generous budgets for their purchase, and skilled support personnel. More recently, clusters of commodity or workstation-class computers have become widely used. This solution is considerably less expensive in terms of hardware costs, but a dedicated cluster of computers for use by multiple users requires enough time and effort to construct and maintain that is out of the reach of most research groups that do not have a good budget for support personnel.

This paper reports on two developments that may make parallel computing attractive to a broader spectrum of researchers who do computations. The first is the existence of a bootable CDROM that allows the creation of a cluster of computers for parallel processing in less than ten minutes. The CDROM is customizable so that needed programs and data may be added quite easily, and thus it is an effective means of creating a temporary, non-dedicated cluster for individual use. The second development is the existence of extensions to some of the high-level matrix programming (HLMP) languages[2] that allow the incorporation of parallelism into programs written in these languages. Since most researchers find programming using HLMP languages considerably faster and easier than programming in compiled languages such as C or FORTRAN, is is possible that the existence of parallel extensions for HLMP languages could stimulate wider use of parallel programming.

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

[2] By "high-level matrix programming language" I mean languages such as MATLAB (TM the Mathworks, Inc.), Ox (TM OxMetrics Technologies, Ltd.), and GNU Octave (`www.octave.org`), for example.

A focus of the paper is on the possibility of hiding parallelization from end users of programs. If programs that run in parallel have an interface that is nearly identical to the interface of equivalent serial versions, end users will find it easy to take advantage of parallel computing's performance. Of course, someone must do the underlying parallel programming. Swann (2002) notes that the decision about whether or not a program should be written to use parallel computing depends 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 can use the programs in the same way they would use serial versions, then the programming effort of a small body of people who know how to write parallel programs can be benefited from by a much larger group of end users. There are economies of scope in such cases. The paper shows that several econometric estimation methods, as well as Monte Carlo simulations and bootstrapping can all be transparently parallelized. Furthemore, the programs can be used with no knowledge of parallel programming. For example, a researcher could use the provided parallel generalized method of moment (GMM) estimation routine in his or her own work, only having to program the function that calculates the moments that define the estimator. This would require ordinary serial programming using a HLMP language, a fairly simple task that is familiar to very many researchers. Then estimation could be done in parallel using the provided code. While this paper emphasizes the possibility of making parallelized routines available for use by researchers who do not know how to write parallel code, the provided code will be useful as examples for people who are interested in learning to write parallel programs. The Monte Carlo example is discussed in detail in Section 5 for this last group of readers.

The examples in this paper are all from econometrics, due to the author's familiarity with the problems, but it is clear that problems from other areas in computational economics could be attacked in a similar way.

2 Software environment

Econometric computations in research work are commonly done using high-level interpreted matrix programming (HLMP) languages. These languages offer an intuitive programming syntax, and very 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, especially if loops are unavoidable. Examples of parallelized code for compiled languages such as C and FORTRAN have been available for quite some time. Only more recently has it become possible to write parallelized programs using HLMP languages, and non-trivial example code is scarce. Since much work in computational economics uses HLMP languages, the potential for adoption of parallel programming techniques may be much larger than it was until recently.

The Message Passing Interface (Message Passing Interface Forum, 1997) is a specification of a mechanism for passing instructions and data between different computational processes, which may run on different 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.

Most HLMP offer a means of linking in compiled FORTRAN and C code, which is usally done to ease performance bottlenecks when code cannot be vectorized. MPI capabilities can be brought to HLMP languages in the same way, by writing wrapper functions that link to the library functions of a chosen MPI package. This has been done using the LAM/MPI implementation of MPI to create the MPI Toolbox (MPITB) for the MATLAB and GNU Octave languages by Fernández Baldomero *et al.* (2003 and 2004). These packages provide bindings for almost all of the MPI-1.2 specification, and for some of the MPI-2 specification. Doornik *et al.* (2002 and forthcoming) have developed bindings to a subset of the MPICH implementation of the MPI-1.2 specification for the Ox language. The Rmpi package for R (Yu, 2004) is another example that appears to be quite complete and functional.

This paper uses MPITB for GNU Octave for its examples. This choice is motivated primarily by its completeness and functionality. Another motivation is the fact that both MPITB and GNU Octave are "free" software. This allows them to be included fully configured and ready to run on the ParallelKnoppix CDROM image that is discussed below. Because all of the software in the CDROM image is free, the image can be made freely available for download. 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 implementations of the MPI standard. Section 5 describes in more detail how MPITB works to allow parallelization of Octave scripts. 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 HLMP language that has a syntax that is compatible with MATLAB's. Most MATLAB scripts will run unmodified under Octave. Octave provides a foundation for programming that is essentially equivalent to that provided by MATLAB. Eddelbuettel (2000) discusses the use of Octave for econometric work, though the language has evolved considerably since that paper was written. The `octave-forge` package available at `octave.sourceforge.net/` provides many extensions and applications, some of which are useful for econometrics. In particular, the BFGS minimization function and serial versions of the ML and GMM estimation functions that are discussed in this paper are contained in the `octave-forge` package. Both Octave and the `octave-forge` extensions will run under the Windows, Linux and Macintosh OS X operating systems.

2.2 ParallelKnoppix

Use of the MPITB package requires 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 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. Traditional dedicated multiuser clusters entail installation and maintenance costs that may place them out of reach of many researchers who could otherwise make good use of parallel computing. ParallelKnoppix (Creel, 2004a) is a re-mastering of the Knoppix (Knopper, undated) CDROM image. Knoppix is a live Linux filesystem on a bootable CD. When booted, the hardware on the computer is automatically detected, and the system configures itself to run a modified version of Debian Linux (`www.debian.org`). ParallelKnoppix adds extensions to Knoppix that allow a Linux cluster to be created with a very modest amount of intervention. The master computer is booted using the CD, and hardware is detected automatically, just as with Knoppix. Then a terminal server is launched, which provides the filesystem to the slave nodes. The slave nodes are booted across the network, each making separate use of the automatic hardware configuration of Knoppix. Thus, the nodes can be heterogeneous. The basic requirement is that the nodes be booted

from their network cards using PXE, or using a CD or floppy disk that may be obtained from `rom.o.matic.net` that simulates this ability. Then a working directory is NFS exported from the master node to all of the slave nodes. Any files placed in this directory are accessible to all of the slaves.

At this point, the cluster is ready to run MPI programs using LAM/MPI or MPICH. The entire process takes less than ten minutes for an experienced user. When the cluster is shut down, the machines are in their original state, so use of the machines for clustering does not interfere with their ordinary use. For example, the results reported below were obtained on a cluster created using computers that have their entire hard disks occupied by Windows (TM Microsoft, Inc.) and which are ordinarily used by students for their day-to-day work. ParallelKnoppix contains LAM/MPI, Octave, MPITB and all the program examples discussed below. This CD is the most convenient means of introducing oneself to MPITB and replicating the results presented here, though it is of course possible to install MPITB and GNU Octave on a dedicated cluster, too. ParallelKnoppix also contains working examples of parallel code for C, FORTRAN and R. An image of this CD is available for download at `pareto.uab.es/mcreel/ParallelKnoppix/`. A tutorial that explains in detail how how to use and modify ParallelKnoppixis available (Creel, 2004b).[3]

3 Example problems

This section introduces example problems from econometrics, and shows how they can be parallelized in a natural way.

3.1 Monte Carlo

A Monte Carlo study involves repeating a random experiment many times under identical conditions. Several authors have noted that Monte Carlo studies are obvious candidates for parallelization (Doornik *et al.* 2002; Bruche, 2003) since blocks of replications can be done independently on different computers. On a cluster of computers, care must be take to ensure that the streams of pseudo-random numbers generated on different nodes are independent of one another. Octave, with the `octave-forge` extensions, uses the Mersenne Twister[4] as the basis for uniform and normal random numbers. The starting point in the sequence is determined using the `/dev/urandom`

[3] If one would like the Octave example scripts without downloading the CD image, please contact the author.
[4] `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`

device on Linux. Since the starting point is random and the period of the Mersenne Twister is so long ($2^{19937} - 1$), the likelihood of overlapping streams of random numbers on different nodes of even large clusters is small enough to be ignored. It is possible to create completely different sequences on each node, but that has not been done in the example programs.

To illustrate the parallelization of a Monte Carlo study, we use same trace test example as do Doornik, *et. al.* (2002). Listing 1 shows the function `tracetest.m`[5] that calculates the trace test statistic for the lack of cointegration of integrated time series. This function is illustrative of the format that we adopt for Monte Carlo simulation of a function: it receives a single argument of cell type, and it returns a row vector that holds the results of one random simulation.[6] The single argument in this case is a cell array that holds the length of the series in its first position, and the number of series in the second position. It generates a random result though a process that is internal to the function, and it reports some output in a row vector (in this case the result is a scalar).

Listing 2 shows an Octave script `mc_example1.m` that executes a Monte Carlo study of the trace test by repeatedly evaluating the `tracetest.m` function of Listing 1. The main thing to notice about this script is that lines 7 and 10 call the function `montecarlo.m`. When called with 3 arguments, as in line 7, `montecarlo.m` executes serially on the computer it is called from. In line 10, there is a fourth argument. When called with four arguments, the last argument is the number of slave hosts to use[7]. We see that running the Monte Carlo study on one or more processors is transparent to the user - he or she must only indicate the number of slave computers to be used.

3.2 Bootstrapping

Bootstrapping, when the data is independently and identically distributed, is simple to parallelize (Tierney, 2003). It can be implemented in much the same way we have done for the Monte Carlo problem. We need a large number of evaluations of some function, where each evaluation used a bootstrap resampling from the original data. The interface we adopt requires that the function that is to be bootstrapped have the same characteristics as functions for Monte Carlo - there is a single cell-type argument, and the return value is a row vector.

[5] The names of scripts given here are the same as the names within the Octave example directory on the ParallelKnoppix CD.

[6] The fact that the input argument is a cell means that it may contain any number of more fundamental arguments of any type. Requiring that output must be a vector is perhaps restrictive, and certainly would be inconvenient in many cases, but this requirement could be relaxed quite easily. It is used here to simplify exposition.

[7] As long as the number of available processors is greater than or equal to the number of requested processors, 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. Otherwise the load is spread round-robin over the number of available processors.

Listing 3 shows the example program `bootstrap_example1.m` which performs a simple bootstrap of the OLS estimator with heteroscedastic errors. Parallel or serial evaluation is controlled in lines 12 and 13. If nslaves is set to zero, evaluation is serial on a single CPU. If nslaves is greater than zero, the load is split among the number of requested CPUs.

3.3 ML

For a sample $\{(y_t, x_t)\}_n$ of n observations of a set of dependent and explanatory variables, the maximum likelihood estimator of the parameter θ can 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.

Listing 4 shows the Octave script `mle_example1.m` that calculates the maximum likelihood estimator of the parameter vector of a model that assumes that the dependent variable is distributed as a Poisson random variable, conditional on some explanatory variables. In lines 1-3 the data is read, the name of the density function is provided in the variable `model`, and the initial value of the parameter vector is set. In line 5, the function `mle_estimate` performs ordinary serial calculation of the ML estimator, while in line 7 the same function is called with 6 arguments. The fourth and fifth arguments are empty placeholders where options to `mle_estimate` may be set, while the sixth argument is the number of slave computers to use for parallel execution, 1 in this case. 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. When executed, this script prints out the estimates `theta_s` and `theta_p`, which are identical.

It is worth noting that a different likelihood function may be used by making the `model` variable point to a different function. The likelihood function itself is an ordinary Octave function that is not parallelized. The `mle_estimate` function is a generic function that can call any likelihood function that has the appropriate input/output syntax for evaluation either serially or in parallel. Users need only learn how to write the likelihood function using the Octave language.

3.4 GMM

For a sample as above, the GMM estimator of the parameter θ 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

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

Since $m_n(\theta)$ is an average, it can obviously be computed blockwise, using for example 2 blocks:

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

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

Listing 5 shows the script `gmm_example1.m`, which illustrates how GMM estimation may be done serially or in parallel. When this is run, `theta_s` and `theta_p` are identical up to the tolerance for convergence of the minimization routine. The point to notice here is that an end user can perform the estimation in parallel in virtually the same way as it is done serially. Again, `gmm_estimate`, used in lines 8 and 10, is a generic function that will estimate any model specified by the `moments` variable - a different model can be estimated by changing the value of the `moments` variable. The function that `moments` points to is an ordinary Octave function that uses no parallel programming, so users can write their models using the simple and intuitive HLMP syntax of Octave. Whether estimation is done in parallel or serially depends only the seventh argument to `gmm_estimate` - when it is missing or zero, estimation is by default done serially with one processor. When it is positive, it specifies the number of slave nodes to use.

3.5 Kernel regression

The Nadaraya-Watson kernel regression estimator of a function $g(x)$ at a point x is

$$\begin{aligned} \hat{g}(x) &= \frac{\sum_{t=1}^{n} y_t K\left[\left(x - x_t\right)/\gamma_n\right]}{\sum_{t=1}^{n} K\left[\left(x - x_t\right)/\gamma_n\right]} \\ &\equiv \sum_{t=1}^{n} w_t y_y \end{aligned}$$

We see that the weight depends upon every data point in the sample. To calculate the fit at every point in a sample of size n, on the order of n^2k calculations must be done, where k is the dimension of the vector of explanatory variables, x. Racine (2002) demonstrates that MPI parallelization can be used to speed up calculation of the kernel regression estimator by calculating the fits for portions of the sample on different computers. We follow this implementation here. Listing 6 shows the script `kernel_example1.m`. Serial execution is obtained by setting the number of slaves equal to zero, in line 15. In line 17, a single slave is specified, so execution is in parallel on the master and slave nodes.

The listings provided in this section illustrate the point that parallelization may be mostly hidden from end users. For the Monte Carlo and bootstrap examples, a user just needs to program the function to be simulated. For the ML and GMM estimation problems, the user needs to write an Octave function that calculates the log-likelihood or moments that define the model. The kernel estimation routine can be called directly as is. Users can benefit from parallelization without having to write or understand parallel code.

4 The speedup from parallelization

This section provides timing results for the five example problems. A homogeneous cluster of 12 nodes was created with ParallelKnoppix in a university computer room which is ordinarily used by students for their work. Each node is a uniprocessor Pentium IV machine running at 2.8 GHz, with 1MB of level 2 cache, with hyperthreading enabled. Each machine has a 3COM 3c905 Tornado network card, and they were connected with a dedicated 100MB/s ethernet network using a 3COM OfficeConnect dual speed switch.

The test programs are `mc_example2.m`, `bootstrap_example2.m`, `mle_example2.m`, `gmm_example2.m`, and `kernel_example2.m`, all contained in the compressed archive `results_for_mpitb_paper.tgz` on the ParallelKnoppix CD. The files of the same names in the uncompressed directories may

have been changed somewhat. The uncompressed examples are recommended for learning how to use MPITB. These example files are computationally more demanding than the examples presented in section 3, but are otherwise similar.

The example programs are not particularly interesting, except that for the GMM problem. In that case we use 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. The function `emm_moments.m` appears in Listing 7. This function is used in the same way as the `Poisson_IV_moments` function is used in line 5 of Listing 5. It is a general purpose function that receives the names of the functions that define the data generating process (DGP) and the auxiliary model (the "score generator") as elements of the third argument. It is presented here just to illustrate the fact that this estimation method is not that difficult to implement in Octave.

In the case of the MLE and GMM problems, which use an iterative BFGS minimization algorithm, experimentation reveals that the solution path that is taken may vary with the number of slave nodes that are used. This is due to differences in the numeric precision of values stored internally in Octave, and as passed between nodes by MPITB and LAM/MPI. Even very small differences may eventually lead to a different solution path, since the objective function is evaluated many times in the course of the BFGS iterations. At each iteration, numeric differentiation is used to find the direction of search, which entails a number of evaluations of the objective function. Determination of the stepsize requires more evaluations. Small differences in the objective function value lead to small differences in the direction of search and the stepsize, but these differences are propogated to and amplified by subsequent iterations. This may cause the BFGS algorithm to use a different number of iterations to achieve convergence, depending upon how many slave nodes are used. For concave problems, the same solution is always found, up to the convergence tolerance that is specified. However it is possible that the solution that is found could vary for nonconvex problems.

Figure 1 show the speedups obtained for the five example programs. Speedup is defined as the serial runtime of the program when run on a single computer, divided by the time on the cluster. We can see that speedup is close to the 45-degree line for the three "embarrasingly parallelizable" problems (Montecarlo, bootstrapping and kernel regression). The two estimation problems that involve iterative minimization have a concave shape. These problems have serial

and parallelized code interleaved. The evaluation of the objective function, in the case of MLE, and the moments, in the case of GMM, is parallelized, but the rest of the BFGS algorithm is serial code. Once the parallelized part is done fast enough, the serial code leds to a floor in runtime that cannot be crossed. Once enough nodes are used to reach the floor, additional nodes will actually increase runtime due to the additional communication overhead. For the GMM problem, this point has essentially been reached with 12 nodes. However, with a more computationally demanding problem than that used in the example, it would be possible to achieve greater speedups using a larger cluster.

The MLE problem is unusual in that a portion of the speedup is above the 45-degree line. This is because the timing of the problem was unusually high on the single computer. The BFGS algorithm seems to have gotten bogged down searching for a stepsize. This did not happen when it was run on the clusters. Recall that the solution path may vary with the number of nodes. This has occured in this case.

The efficiency of computation is presented in Figure 2. Efficiency is the speedup divided by the number of nodes. For a perfectly parallelized program that performs the same number of computations independently of the number of nodes, efficiency would be 1. Lower numbers indicate lack of full parallelization and/or the effect of communication overhead. We see that the Montecarlo, bootstrapping and kernel regression examples are all highly efficient. This indicates that substantial reductions in computational time would be obtained if a larger cluster were used. The MLE and GMM problems have more rapidly declining efficiencies. These problems have a great deal of communication between nodes thoughout the entire stream of computations. The low efficiency at 12 nodes is indicative that almost all of the possible speedup has been obtained. Again, for more costly to evaluate objective functions or moment functions, greater speedups could be obtained with a larger cluster.

Figures 3 and 4 illustrate the speedups and efficiencies of the kernel regression problem, for several different sample sizes. For a small sample, there is no possible speedup, and efficiency declines rapidly. For large samples, speedup is close to the 45ž line , and efficiency declines slowly. This figure is illustrative of what would happen with MLE and GMM estimation with more computationally demanding problems than those of the examples. The time to evaluate the objective or moment functions would become larger, relative to the time to perform the serial part of the BFGS algorithm, which only depends upon the number of parameters. Since the parallelized part would occupy a larger portion of the total runtime, there would be greater

gains from parallelization. The basic conclusion is that the degree to which parallelization is useful depends upon the specific problem.

5 Writing parallel code: the Monte Carlo example

For readers who may be interested in writing their own parallel Octave code, this section discusses the `montecarlo.m` function in detail, to show how a serial function may be parallelized using MPITB, and how the parallelization may be hidden from end users. The MPITB package provides the MPI bindings for Octave, as well as supporting functions. 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 when using MPI functions from inside Octave scripts or functions. Since many sources of information are available, we do not go into the details of how to use MPI functions. Rather, we focus on the steps needed to enable the use of these functions with Octave.

The `montecarlo.m` function appears in Listing 8. In line 1, we see that `montecarlo.m` has 4 arguments: `f`, the function that is to be simulated; `f_args`, a cell that holds any arguments needed to evaluate `f`; `reps`, the number of Monte Carlo replications; and `nslaves`, the number of slave nodes if execution is to be done in parallel. This last argument is optional. Recall that the function to which `f` is assigned must return a row vector of output. Lines 2-5 call `f` one time to find out the number of elements in the row vector it returns, and then create a holder for all the results of the `reps` replications. Lines 6-14 check whether the last argument, `nslaves`, is omitted or is not positive, in which case the global variable `PARALLEL` is set to false and execution will be serial on the master computer only. Otherwise, when `nslaves` is positive, some global variables are set. In particular, `PARALLEL` is set to true, and the global variable `NSLAVES` is set to the number of slave computers, `nslaves`.

In line 16, the Monte Carlo study is done serially if `PARALLEL` is false, by simply looping over `reps` evaluations of `f` and storing the result. Otherwise, lines 18 to 51 do the Monte Carlo study in parallel using the `NSLAVE`+1 computers. We now examine these lines in detail.

Line 18 calls the `LAM_Init` function. This function ensures that the MPI environment is ready to execute the subsequent code on the requested number of computers. It is listed in Listing 9. Lines 2-6 of this function declare and set values for global variables needed by the MPITB supporting functions. Lines 7-8 check whether `MPI_Init` has been called previously, and if not,

it is called. This initialization is a requirement of the MPI specification, and is discussed in the MPI documentation. The Octave-specific, and, for our purposes, most interesting lines are 11-12. Line 11 calls the Octave function `Octave_Spawn`. This function uses the MPITB binding to the MPI-2 function `MPI_Comm_spawn` to start Octave on `NSLAVES` slave computers. When Octave starts on any slave node, the script `startupNumCmds.m` is run. Returning `LAM_Init`, next, in line 12, the `NumCmds_Init` function is called. This function, together with the actions of the `startupNumCmds.m` script that runs on each slave node, puts the slave nodes into loops where they are ready to receive a packed buffer and act upon its contents when it arrives. One part of the packed buffer may be a command, which may change during the course of events. When a buffer containing a command is received by a slave node, the command is executed. The command may contain a call to `MPI_Send`. In this way, the slaves can send the results of their calculations back to the master computer. The master computer may then send a new packed buffer that may contain new data and a new command. This send/receive cycle continues until the master computer sends an exit command to the slave nodes.

This generic description of the send/receive loop is implemented in the `montecarlo.m` function (Listing 8) in the manner we now describe. Lines 23-24 use the `NumCmds_Send` function to send symbols in Octave's memory to the slave nodes. Whatever symbols are needed to perform the requested calculations must be passed either here, or later on, before the slaves are asked to use them. The `NumCmds_Send` function is a convenient way to send all static symbols in a single packed buffer. Lines 20-21 define the string variable `cmd` that the slave nodes will execute using Octave's `eval` function. In this case, the `cmd` string contains two Octave command lines - a first line that performs the Monte Carlo replications, and a second line that sends the results back to the master computer. Line 22 determines how many replications will be done on each of the slave nodes. The symbols that are the arguments to the command in the first line of `cmd` are sent together with `cmd` using `NumCmds_Send`, in lines 23-24.

Lines 26-28 run a portion of the total number of Monte Carlo replications on the master computers, while the slave nodes are busy doing their calculations. Both the master node and the slaves use the function `montecarlo_nodes.m`, in Listing 10, to do the requested number of evaluations of `f`, in a manner very similar to the way it is done serially, in line 16 of Listing 8. Then in lines 31-36 the results of the slaves are received and incorporated into the `output` matrix. Finally, line 38 calls the `LAM_Finalize` script, which shuts down Octave on the slave nodes and clears MPI-related symbols from memory.

This description is intended to make clear how the Monte Carlo problem was parallelized, and it serves as an introduction to using MPITB for a problem of interest to econometricians. The Parallelknoppix CDROM contains the parallelized source code for the other examples discussed in this paper. The MLE and GMM programs provide slightly more advanced examples of parallel programming with MPITB for Octave. The examples are intended to be clear and relatively easy to understand, to encourage interested readers to do their own programming using MPITB. There may be some scope for performance improvements in the example implementations. For example, in the MLE and GMM examples, the `MPI_Bcast` command could be used to send the trial parameter values to all the slave nodes at once instead of using `MPI_Send` inside a loop. Also, dynamic load balancing could improve performance on heterogeneous clusters. These optimizations have been avoided in the example programs in order to retain maximum clarity.

6 Conclusion

This paper has shown that MPITB for GNU Octave can be used to achieve important reductions in computational time for several problems in econometrics. Furthermore, the implementation of parallelization is such that an econometrician who makes use of the parallel routines does not have to know anything about parallel programming. The ParallelKnoppix CD provides a simple way of creating a cluster and experimenting with the programs.

The most obvious direction for future work is to apply this relatively easy to use technology to real problems of research interest. Another possibility is to develop parallelized code that could be used in other areas of research in economics. Without doubt, researchers will be able to think of possibilities in their areas of specialization. Contributions of examples for inclusion on the ParallelKnoppix CD are welcomed.

References

[1] Bruche, M. (2003) A note on embarassingly parallel computation using OpenMosix and Ox, working paper, Financial Markets Group, London School of Economics.

[2] Creel, M. (2004a) ParallelKnoppix - rapid deployment of a Linux cluster for MPI parallel processing using non-dedicated computers, UFAE and IAE Working Papers, `http://pareto.uab.es/wp/2004/62504.pdf`.

[3] Creel, M. (2004b) ParallelKnoppix - ParallelKnoppix tutorial, UFAE and IAE Working Papers, `http://pareto.uab.es/wp/2004/62604.pdf`

[4] Doornik, J.A., D.F. Hendry and N. Shephard (2002) Computationally-intensive econometrics using a distributed matrix-programming language, *Philosophical Transactions of the Royal Society of London, Series A*, 360, 1245-1266.

[5] Doornik, J.A., D.F. Hendry and N. Shephard (in press) Parallel computation in econometrics: a simplified approach, E. Kontoghiorgies (ed.) *Handbook on Parallel Computing and Statistics*, Marcel Dekker.

[6] Eaton, J.W. (1998) Octave home page, `www.octave.org`

[7] Eddelbuettel, D. (2000) Econometrics with Octave: software review, *Journal of Applied Econometrics*, **15**, 531-42.

[8] Fernández Baldomero, J. *et al.*, (2003) Performance of message-passing MATLAB toolboxes, *Lecture Notes in Computer Science*, Volume 2565, Springer-Verlag, Heidelberg, 228 - 241.

[9] Fernández Baldomero, J. *et al.*, (2004) MPI toolbox for Octave, VecPar'04, Valencia, Spain, June 28-30 2004, `http://atc.ugr.es/~javier/investigacion/papers/VecPar04.pdf`.

[10] Fernández Baldomero, J. (2004) LAM/MPI parallel computing under GNU Octave, `atc.ugr.es/javier-bin/mpitb`.

[11] Gallant, A. R. and G. Tauchen (1996) Which moments to match?, *Econometric Theory*, **12**, 657-81.

[12] Gropp, W., E. Lusk, N. Doss and A. Skjellum (1996) A high-performance, portable implementation of the MPI message passing interface standard, *Parallel Computing*, **22**, 789-828, see also `www-unix.mcs.anl.gov/mpi/mpich/`.

[13] Knopper, Klaus (undated) KNOPPIX - Live Linux Filesystem on CD, www.knopper.net/knoppix/index-en.html.

[14] LAM team (2004) LAM/MPI parallel computing, www.lam-mpi.org/.

[15] Message Passing Interface Forum (1997) MPI-2: Extensions to the message-passing interface, University of Tennessee, Knoxville, Tennessee.

[16] Nagurney, Anna (1996) Parallel computation, *Handbook of computational economics*. Volume 1, pp. 335-404, *Handbooks in Economics*, vol. 13. Amsterdam, New York and Oxford, Elsevier Science, North-Holland.

[17] Racine, Jeff (2002) Parallel distributed kernel estimation, *Computational Statistics & Data Analysis*, **40**, 293-302.

[18] Swann, C.A. (2002) Maximum likelihood estimation using parallel computing: an introduction to MPI, *Computational Economics*, **19**, 145-178.

[19] Tierney, L. (2003) Simple parallel statistical computing in R, http://www.stat.uiowa.edu/~luke/talks/uiowa03.pdf.

[20] Yu, Hao (2004) The Rmpi package, http://cran.es.r-project.org/doc/packages/Rmpi.pdf.

Figures

Figure 1: Speedup

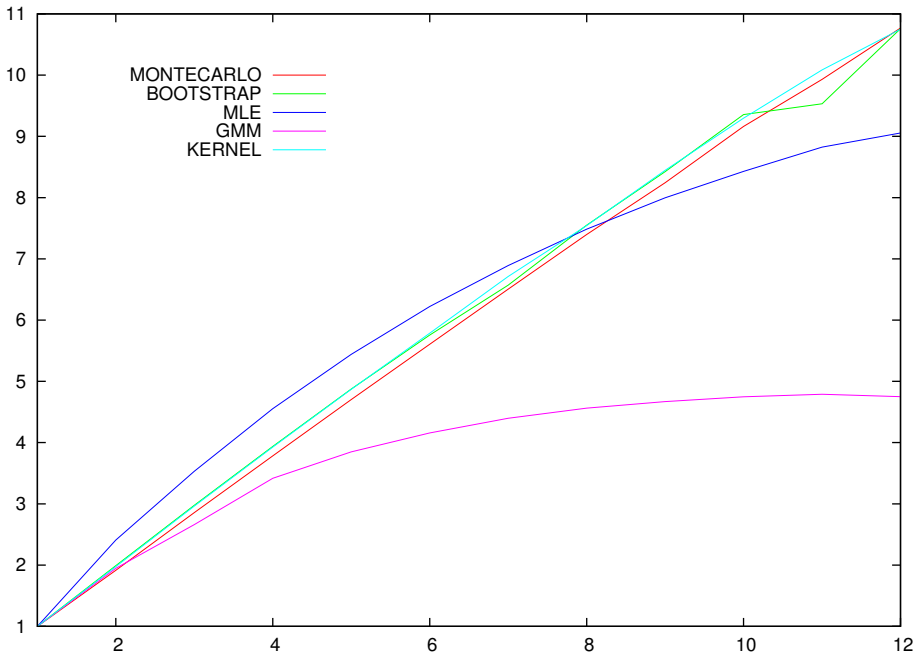


Figure 2: Efficiency

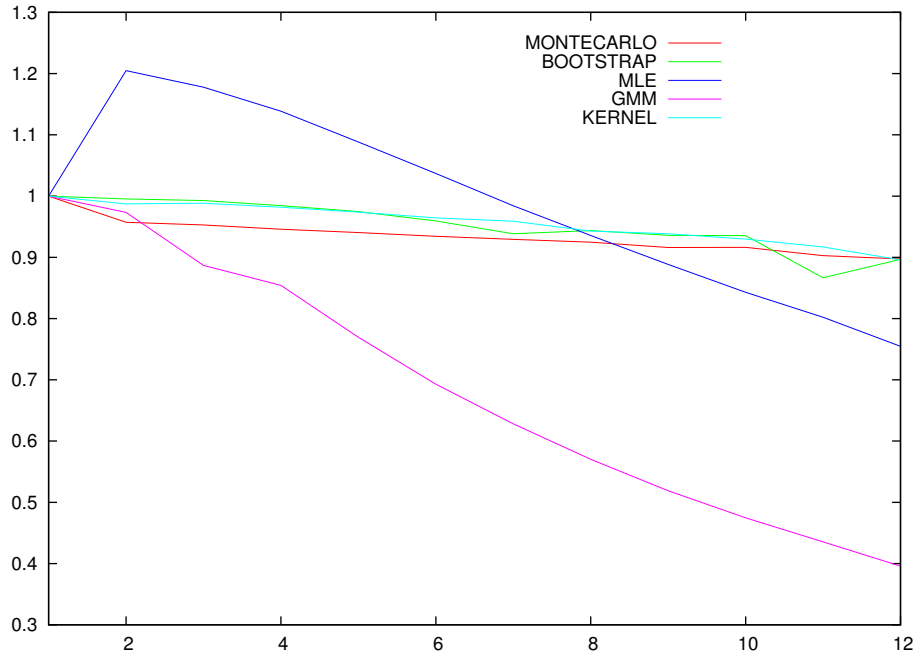


Figure 4: Efficiency, kernel regression

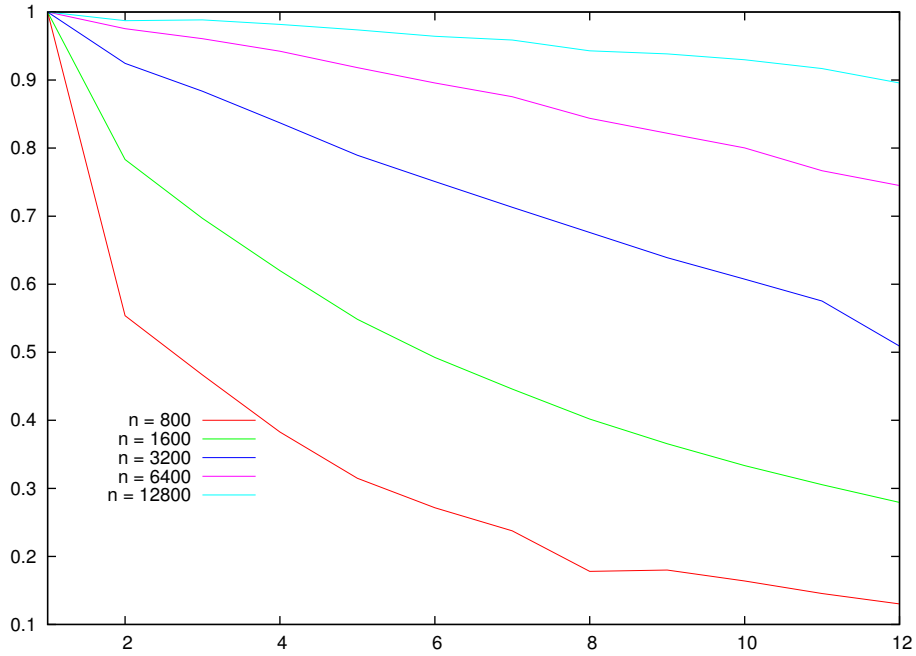
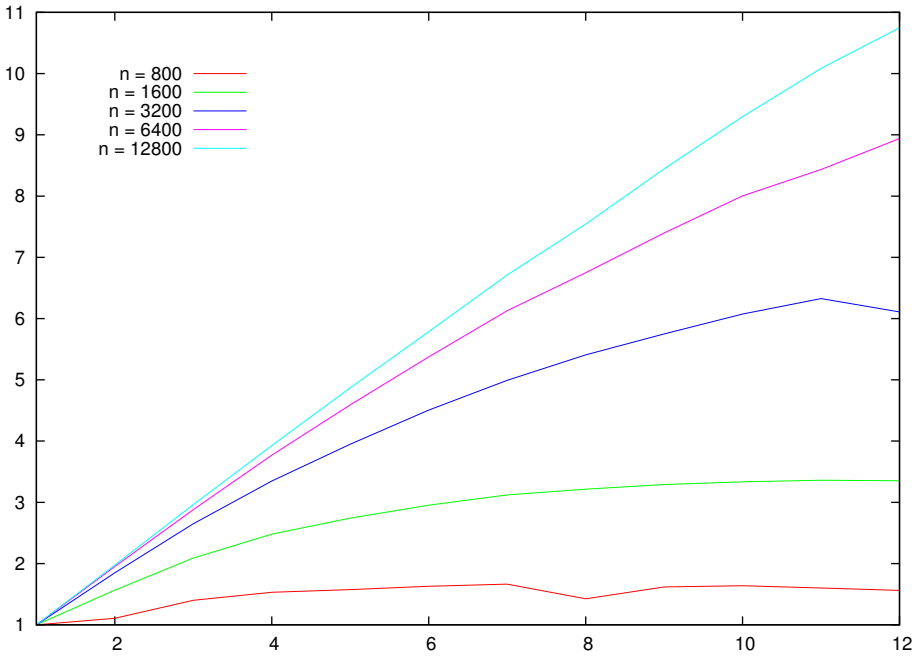


Figure 3: Speedup, kernel regression



Program Listings

```
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 1: tracetest.m

```
# number of monte carlo replications
reps = 1000;
# specifics of test
T = 1000;
dim = 5;
# run on master only
output = montecarlo("tracetest",{T, dim}, reps);
hist(output, 30);
# run on master and one slave
output = montecarlo("tracetest",{T, dim}, reps, 1);
figure(2);
hist(output, 30);
```

Listing 2: mc_example1.m

```
# bootstraps the OLS estimator using heteroscedastic data
reps = 2000;
n = 50;
k = 2;
# generate data
x = [ones(n,1) rand(n,k-1)];
e = 2*x(:,k) .* randn(n,1); # heteroscedastic errors
y = x*ones(k,1) + e;
f_args = {[y x]}; # we want a cell, remember?
f = "myols";
# get bootstrap standard errors
nslaves = 1;
output = bootstrap(f, f_args, reps, nslaves);
disp(std(output))
```

Listing 3: bootstrap_example1.m

```
load poisson_data;
model = "Poisson"; # name of function that calculates loglikelihood
theta = zeros(columns(poisson_data)-1,1); # starting values
# Estimation: serial
theta_s = mle_estimate(theta, poisson_data, model);
# Estimation parallel
theta_p = mle_estimate(theta, poisson_data, model, "", "", 1);
printf("MLE parameter estimates: %s model, %d observations\n", model,rows(poisson_data));
labels = str2mat("serial", "parallel");
prettyprint_c([theta_s theta_p], labels);
```

Listing 4: mle_example1.m

```
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}; # arguments of function assigned to "moments"
# gmm estimation: serial
theta_s = gmm_estimate(theta, data, weight, moments, momentargs);
# gmm estimation: parallel
theta_p = gmm_estimate(theta, data, weight, moments, momentargs, "", 1);
```

Listing 5: gmm_example1.m

```
# generates artificial data; does kernel smoothing, serially and in parallel
n = 2000; # sample size
# uniformly space data points on [0,2]
x = 1:n;
x = x';
x = 2*x/n;
# generate data
true = x + (x.^2)/2 - 3.1*(x.^3)/3 + 1.2*(x.^4)/4;
sig = 0.5;
y = true + sig*randn(n,1);
# define things for smoothing
data = [y x];
ww = 0.2; # window width
# serial (zero slaves)
[fits_s , cvscores_s] = kernel_regression(data, ww, 0);
# parallel (1 slave)
[fits_p , cvscores_p] = kernel_regression(data, ww, 1);
# plot serial and parallel to verify they're the same
multiplot(2,1);
mplot(x, fits_s, x, true);
mplot(x, fits_p, x, true);
```

Listing 6: kernel_example1.m

```
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 (SG)
  sgargs = momentargs{5}; # SG arguments (cell array)
  phi = momentargs{6}; # QML estimate of SG parameter
  y = data(:,1);
  x = data(:,2:k+1);
  rand_draws = data(:,k+2:columns(data)); # passed with data to ensure fixed across
      iterations
  n = rows(y);
  scores = zeros(n,rows(phi)); # container for moment contributions
  reps = columns(rand_draws); # how many simulations?
  for i = 1:reps
    e = rand_draws(:,i);
    y = feval(dgp, theta, x, e, dgpargs); # simulated data
    sgdata = [y x]; # simulated data for SG
    scores = scores + numgradient(sg, {phi, sgdata, sgargs}); # gradient of SG
  endfor
  scores = scores / reps; # average over number of simulations
endfunction
```

Listing 7: emm_moments.m

```
function output = montecarlo(f, f_args, reps, nslaves)
  # determine size of output vector from f, and create container for results
  output = feval(f, f_args);
  n_returns = size(output,2);
  output = zeros(reps, n_returns);
  # check if doing this parallel or serial
  global PARALLEL = 0; # default is serial
  if (nargin == 4)
    if nslaves > 0
      global NSLAVES NEWORLD NSLAVES TAG;
      PARALLEL = 1;
      NSLAVES = nslaves;
    endif
  endif
  if !PARALLEL # ordinary serial version
    for i = 1:reps output(i,:) = feval(f, f_args); endfor
  else # parallel version
    LAM_Init(nslaves);
    # The command that the slave nodes will execute
    cmd=['contrib = montecarlo_nodes(f, f_args, n_returns, nn); ',\
        'MPI_Send(contrib,0,TAG,NEWORLD);'];
    nn = floor(reps/(NSLAVES + 1)); # How many reps per slave? Rest is for master
    NumCmds_Send({'f', 'f_args', 'n_returns','nn','cmd'}, \
      {f, f_args, n_returns, nn, cmd}); # Send data to all nodes
    # run command locally for last block (slaves are still busy)
    n_master = reps - NSLAVES*nn; # how many to do?
    contrib = montecarlo_nodes(f, f_args, n_returns, n_master);
    output(reps - n_master + 1:reps,:) = contrib;
    # collect slaves' results
    contrib = zeros(nn,n_returns);
    for i = 1:NSLAVES
      MPI_Recv(contrib,i,TAG,NEWORLD);
      startblock = i*nn - nn + 1;
      endblock = i*nn;
```

```
35      output(startblock:endblock,:) = contrib;
36    endfor
37    # clean up after parallel
38    if PARALLEL LAM_Finalize; endif
39  endif
40 endfunction
```

Listing 8: montecarlo.m

```
1  function LAM_Init(nslaves)
2    global HOSTS RPI NSLAVES DEBUG TAG NEWORLD PARALLEL;
3    PARALLEL = true;
4    RPI = 'tcp';
5    DEBUG = false;
6    NSLAVES = nslaves;
7    [junk test] = MPI_Initialized;
8    if !test
9      MPI_Init;
10   endif
11   Octave_Spawn;
12   NumCmds_Init(0, 0);
13 endfunction
```

Listing 9: LAM_Init.m

```
1  # this is the block of the montecarlo loop that is executed on each slave
2  function contrib = montecarlo_nodes(f, f_args, n_returns, nn)
3    contrib = zeros(nn, n_returns);
4    for i = 1:nn
5      contrib(i,:) = feval(f, f_args);
6    endfor
7  endfunction
```

Listing 10: montecarlo_nodes.m