EMBARRASSINGLY EASY EMBARRASSINGLY PARALLEL PROCESSING IN R: IMPLEMENTATION AND REPRODUCIBILITY

MICHAEL S. DELGADO AND CHRISTOPHER F. PARMETER

"The only people who have anything to fear from free software are those whose products are worth even less." - David Emery

1. Overview

Bootstraps, Monte Carlo simulations, Markov Chain Monte Carlo methods, and nonlinear optimization are commonly implemented econometric procedures in all areas of applied economic research. Several common uses of the bootstrap include estimation of standard errors in nonlinear or nonparametric regression analysis, or in consistent model specification testing to approximate the distribution of the test statistic under the null hypothesis. Monte Carlo simulations are commonly used in econometrics to verify the finite sample performance of newly developed estimation methods, Markov Chain Monte Carlo methods are typically used in Bayesian analysis, while nonlinear optimization is required to estimate any nonlinear function, for example nonlinear regression functions or cross validation functions often used to select smoothing parameters in nonparametric regression models.

Given the recent availability of large microeconomic datasets, and the complexity of many advanced statistical models, each of these procedures is typically computationally intensive, requiring days, if not weeks, for a (single) computer to complete the assigned task. Relatively more complex problems often take substantially larger amounts of time, effectively rendering such procedures infeasible. Alternatively, researchers may be put off from investigating additional or more complex model specifications given the computation time required for such investigations. Such time considerations have no doubt led econometricians and applied researchers towards parallel processing algorithms that essentially divide the assigned task into multiple independent parts that can each be completed simultaneously by separate computer processors. Such computational problems, often

Purdue University and University of Miami

Date: January 17, 2013.

Key words and phrases. Parallel Processing, Reproducibility, Computational Efficiency, Bootstrap, Nonlinear Optimization, Monte Carlo.

Michael S. Delgado, Department of Agricultural Economics, Purdue University, West Lafayette, IN 47907-2056. Phone: 765-494-4211, Fax: 765-494-9176, E-mail: delgado2@purdue.edu.

Christopher F. Parmeter, Department of Economics, University of Miami, Coral Gables, FL 33124-6520. Phone: 305-284-4397, Fax: 305-284-2985, E-mail: cparmeter@bus.miami.edu. We thank Jeff Racine and Achim Zeileis for feedback on this project.

* This research was supported in part by computational resources provided by Information Technology at Purdue - Rosen Center for Advanced Computing, Purdue University, West Lafayette, Indiana.

referred to as 'embarrassingly parallel' problems, are especially suited for simple parallel implementation because they require multiple replications of *independent* computations. Many standard econometric procedures, such as the bootstrap, Monte Carlo simulations, Markov Chain Monte Carlo methods, or nonlinear optimization requiring multiple starting locations, are 'embarrassingly parallel' problems. For example, a Monte Carlo simulation requiring 1,000 independent replications might instead be assigned as 250 independent replications to four separate processors (or cores).[1] The assigned task will effectively be completed four times faster than if one single processor was assigned the 1,000 replications.

Currently, there exist parallel versions of many commonly used econometric software. For example, the Stata/MP version of Stata, the Parallel Computing Toolbox in Matlab, or the MP Connect setup in SAS. Each of these parallel versions advertise that statistical programs can be implemented in parallel with very few modifications to existing calls, and that computation time can be drastically reduced relative to the standard (sequential) version. Three points are clear. First, the existence of these parallel packages provide evidence that practitioners are facing problems that are increasing in computational difficulty, and so desire more efficient computation algorithms. Second, access to cluster or multiple processor environments is increasingly common. Third, many practitioners lack the knowledge or desire to understand and implement complex parallel algorithms in order to achieve computational efficiency, and so require user friendly parallel interfaces.

There are, however, several major drawbacks to many of the existing parallel algorithms and software. First, parallel versions of commonly used statistical software are often expensive, and many universities or research institutions may not already have licensed parallel versions available for use. Second, others may be comfortable using existing statistical software designed to run in parallel, but such programs may not offer implementation of the desired econometric procedures. Third, in the absence of available pre-packaged parallel software, many interested researchers do not find it feasible to independently construct and implement parallel algorithms given the general complexity of the computer programs required to simultaneously employ multiple processors, or fourth, they do not have access to a large cluster computer consisting of multiple processors on which to run computationally intensive procedures. However, recent developments in both standard computing technology and freely available software have greatly reduced the barriers to parallel computing. Indeed, nearly all standard desktop and laptop computers have multi-core processors, which, with the right software, can run in parallel to simultaneously complete the desired task. In addition to the wide availability of multiple processors, several recent software packages in `R` have been produced that allow the modeler to implement *any* 'embarrassingly parallel' econometric procedure in parallel, *without* a complicated or expensive setup of the parallel environment.

Several articles have been published in recent years to address the rising interest in parallel computing (Swann 2001, Swann 2002, Creel 2005, Creel 2007, Schmidberger, Morgan, Eddelbuettel,

[1]In a parallel environment, the terms 'processors', 'cores', 'cpu's', and 'nodes' are often used interchangeably. Throughout this review, we will use the term 'processor'. Note that different 'processors' need not be physically separate, as in the case of a single 'dual-core processor' available on many modern desktop/laptop computers.

Yu, Tierney & Mansmann 2009, Ho, Huynh & Jacho-Chávez 2011, Eugster, Knaus, Porzelius, Schmidberger & Vicedo 2011, Hayfield & Racine 2012). These articles primarily aim to provide a simple overview of a variety of existing software for practitioners who are interested in implementing existing parallel programs. These articles provide a thorough overview of many different parallel functions provided in each of the described software packages. While this approach is comprehensive in its review of parallel procedures, it has the potential to make parallel implementation more difficult for practitioners who wish to decrease computation time *without* having to master even basic principles in parallel computing. Also, some of these software packages and reviews are designed with specific econometric procedures in mind, and may not generally accommodate any procedure the applied econometrician desires to implement.

It is our aim to further reduce the barriers to implementing parallel algorithms and draw the attention of applied econometricians by focusing on several particular software packages available in `R` that can be used to implement parallel computing on *any* 'embarrassingly parallel' econometric procedure. We focus on the packages titled **`parallel`**, **`multicore`**, **`snow`** and **`snowfall`**, and provide 'embarrassingly easy' roadmap to implement parallel algorithms on 'embarrassingly parallel' computational problems.[2]

In accordance with recent research advocating reproducibility of econometric research (Meredith & Racine 2009, Koenker & Zeileis 2009), we discuss several convenient calls that can ensure reproducibility of econometric results. The parallel environment provides unique challenges for reproducibility of statistical results, which are not present in standard sequential setups. Reproducibility requires fixing the numerical starting point for *each* independent computation, so that (i) random number generation is different across processors, and (ii) it does not matter which processor conducts which computation. These issues are not present in a standard sequential processing environment.

In a sequential setup, replicability is easily ensured by fixing the numerical starting point of the program prior to performing any computations (i.e., setting the seed). In the parallel environment, setting a single seed prior to performing any computations does not ensure that each processor on the cluster is operating off of different seeds, nor does it ensure that each random number stream across the processors is replicable, nor does it ensure that the starting point for any particular computation (say, a single run in a bootstrap or Monte Carlo simulation) is fixed and reproducible. Indeed, if researchers are deploying parallel environments with a different number of processors, or different parallel or statistical packages, setting a single seed cannot guarantee that the numerical starting point for any particular computation is fixed and reproducible. In addition to our review of 'embarrassingly easy' parallel implementation in `R`, we provide a thorough discussion of reproducibility of parallel computations along with several techniques that can be used to ensure reproducibility of parallel statistical computations.

[2]The package `npRmpi` in `R` (Hayfield & Racine 2012) provides a parallel version of a broad range of nonparametric estimation procedures, and as such provides an alternative to the packages outlined here for certain nonparametric procedures.

Finally, we provide examples of ‘embarrassingly easy’ parallel implementation of three standard ‘embarrassingly parallel’ econometric procedures - a bootstrap, Monte Carlo simulation, and non-linear optimization - and show how computation time can be drastically reduced with only several simple commands. We further provide a detailed example that illustrates how *any* parallel computation can be fully reproducible using *any* parallel library or *any* number of processors. We show how researchers in applied econometrics or statistics can implement parallel computing on virtually *any* standard desktop or laptop computer with very few modifications to their existing computer codes. Hence, we show how computation time on standard econometric procedures can be drastically reduced *without* any fancy computer software or access to a large scientific cluster computer. We emphasize that while this review is focused towards implementing parallel processing on standard desktop/laptop computers, all of the procedures and commands discussed within, including those pertaining to parallel reproducibility, are entirely applicable on large scientific research clusters without any modification.

2. Overview And General Framework

Here we outline the anatomy of an embarrassingly parallel problem and describe the specific `R` packages and commands that one has at their disposal to implement an embarrassingly parallel problem.[3] It is important to clarify that we will not focus on all of the parallel call functions available in each package, but instead selectively highlight several simple functions that may be of primary interest to a variety of applied econometricians. We will also not discuss the technical workings of any of the call functions that we have selected to showcase. We emphasize that our purpose is to show how applied econometricians can implement *any* embarrassingly parallel yet computationally demanding task in parallel at virtually zero programming cost, and how to ensure reproducibility of results. We refer the interested reader to Meredith & Racine (2009), Schmidberger et al. (2009), Eugster et al. (2011), Knaus (2012), Redd (2012), Tierney, Rossini, Li & Sevcikova (2012), Urbanek (2012) for more technical details and a more thorough overview of these packages.

To begin we note that many embarrassingly parallel problems can be conceived as a simple loop. In a simple loop, one runs a task numerous times and the outcome of an iteration of the loop does not depend on previous iterations of the loop. This feature of the loop is what allows running in parallel to be both simple and expeditious. In the simplistic setup the loop runs over M trials and the object of desire is stored from each trial. Once the loop has finished running those M objects are then analyzed subject to the problem at hand.

In a parallel implementation, the setup is essentially the same, with a few additional steps. The general structure for any parallel program is to initialize the parallel environment and then carry out the assigned (parallel) task. When `R` is initally launched, it is typically assumed that the user will perform all computations sequentially. Hence the first step in performing the parallel

[3]Note that our review of the selected `R` packages is by no means comprehensive with respect to the available `R` packages that might either implement parallel computations or assist in reproducibility of parallel results. Other packages may be useful in the current context, but are not part of this review.

calculation is to initialize the parallel environment. In this step, the parallel package sets up a group of interconnected processors which will work simultaneously to solve the problem. Once initialized, all parallel computations can take place, after which the results from each processor can be collected and the parallel environment can be shut down. `R` then reverts back to sequential processing for analysis of the results.

One can think of running an embarrassingly parallel problem as

(1) Create/open the cluster environment;
(2) Pass necessary objects to all processors;
(3) Initialize the random number streams on each processor (if necessary);
(4) Run the components of the loop on the processors;
(5) Shut down the processors and return to sequential mode.

Steps 1, 2 and 5 are unique to the parallel environment while steps 3 and 4 are roughly identical to what a traditional implementation of a loop would incur.

With embarrassingly parallel problems, such as those emphasized in this paper, the only modification required to run existing codes in parallel is to wrap the codes into a generic function environment. The purpose of wrapping the codes into a function is to create a single and computationally independent object that can be sent to any available processor for computation. For instance, the function can be sent into the parallel computation routines to be independently evaluated a large number of times on each of the available processors. Such function wrapping could be carried out in `R` via

```
> function <- function(a) {codes}
```

in which the function is named `function` and `codes` are the existing codes from the sequential setup. For example, in a Monte Carlo setup, the `{codes}` will generate the simulated data, run the simulation, and, say, return a p-value. The function index `a` is merely a generic index that will allow any processor to repeat the function (a large number of times). This function can then be implemented using a parallel version of the `lapply` command. The examples in Section 4 provide detailed examples on how to break these problems down into single independently computable pieces, in order to construct the function to send to the parallel `lapply` function in the context of three 'embarrassingly parallel' problems.

2.1. **Available Packages And Commands.** The main packages currently available in `R` to implement embarrassingly parallel problems are `parallel`, `multicore` (Urbanek 2012), `snow` (Tierney et al. 2012), and `snowfall` (Knaus 2012). The `multicore` and `snow` ('snow' stands for Simple Network of Workstations) packages were designed to run parallel processing on computers with multiple cores or processors. The `parallel` package provides a simple parallel environment capable of calling several key functions from the packages `multicore` and `snow`, and the `snowfall` package was designed as a more user-friendly wrapper for the `snow` package.

Table 1 lists the key commands for steps 1-5 for each parallel library when implementing an embarrassingly parallel problem.

Table 1. Necessary calls from `multicore`, `parallel`, `snow` and `snowfall` to implement an embarrassingly parallel computational problem.

	Sequential	multicore	parallel	snow	snowfall
Step 1.	–	`multicore:::detectCores()`	`detectCores()`	`makeCluster()`	`sfInit()`
Step 2.	–	–	`clusterExport()`	`clusterExport()`	`sfExport()`
Step 3.	`set.seed()`	`set.seed()`	`clusterSetRNGStream()`	`clusterSetupRNGStream()`	`sfClusterSetupRNGstream()`
Step 4.	`lapply()`	`mclapply()`	`mclapply()`	`parLapply()`	`sfLapply()`
Step 5.	–	`multicore:::closeAll()`	`stopCluster()`	`stopCluster()`	`sfStop()`

2.2. **A General Overview Of The Packages.**

One of the primary advantages of the `multicore` setup above the other packages is the shared memory between the primary processor first initialized by `R` in the sequential setup and the cluster of processors spawned for parallel computation, which alleviates the necessity to manually share information between the processors (this is a consequence of the parallel setup known as forking, on which `multicore` is based). That is, any processors that are spawned for the parallel computation will have access to all of the information (e.g., data, parameter values, etc.) that was initially provided when the program was operating sequentially; in essence step 2 of our embarrassingly parallel setup is unnecessary when one uses the `multicore` package. The primary downside of `multicore` for many practitioners is that `multicore` does not run on Windows-based machines. `multicore` will, however, run on other operating systems such as UNIX, Linux, or Mac OS. Since memory is shared across each of the processors, implementation of `multicore` requires relatively fewer commands to the user then the other packages (see the alternative setups below).

The primary purpose behind the creation of `snow` was to provide novice users a straightforward parallel environment in which to implement 'embarrassingly parallel' problems. Since many standard econometric procedures require multiple independent computations, `snow` is well suited for the needs of an applied econometrician who desires to decrease computation time without learning complicated functions and programs that are often required to implement parallel processing. Several attractive features of `snowfall` for applied econometricians are the automatic calls to the cluster environment once the initialization of the cluster environment has occurred, and the ability for functions to run in both parallel and sequential mode. `snow`, `snowfall` and `parallel` all can be run on Windows-based machines.

3. Ensuring Reproducibility

There has been much interest in recent years in the reproducibility of econometric research (Koenker & Zeileis 2009, Meredith & Racine 2009), no doubt motivated by the explosion of computation-based methods made available by rapidly improving computational ability. The movement towards parallel computation has led to new issues for ensuring reproducibility. The challenge of parallel reproducibility boils down to ensuring that *each independent computation* has a reproducible starting point. We consider several alternative ways to guarantee parallel reproducibility, depending on the parallel environment deployed by the econometrician.

3.1. **Independent Random Number Streams Across Processors.** An important issue that arises in the parallel context is ensuring that streams of random numbers generated on separate processors are unique. In an ‘embarrassingly parallel’ setting, computational problems such as a bootstrap or Monte Carlo would fail if different processors used identical streams of random numbers, as the computations across processors would be repetitive. Indeed, methods for generating independent random number streams across processors have been proposed based on complex random number generator algorithms (L’Ecuyer 1999).

It is well recognized that standard linear congruential random number generators should not be used in practice to generate uniform random numbers (L’Ecuyer & Simard 2001). The reason is that the periods of linear congruential random number generators are too short, on the order of 2^{31}, which even small personal computers running sequentially can exhaust in a few minutes with the available computing power. This issue is exacerbated in a parallel context in which multiple random number streams are generated simultaneously, and are typically required to be unique.

It is now common to use multiple recursive generators, such as the combined multiple recursive generator of L’Ecuyer (1999) which has a long period, 2^{191}, or the Mersenne Twister of Matsumoto & Nisimura (1998) that has a period of roughly 2^{19937}. The length of the periods in these more complex methods ensure (at least in practice) that random number streams will not be repetitive. Hence, these complex random number generators are particularly useful in a parallel context.

Combined multiple recursive generators use numerous states (‘states’ are also often referred to as ‘seeds’) as opposed to linear congruential random number generators which have only a single state. For example the combined multiple recursive generator of L’Ecuyer (1999) has six states. These additional states beyond the linear congruential random number generator is what allows for an extended period of the random number generator. L’Ecuyer, Simard, Chen & Kelton (2002) use the combined multiple recursive generator of L’Ecuyer (1999) to provide a method to create disjoint streams of random number sequences. These sequences are ideal for parallel environments since a stream of random numbers is sent to each processor and these streams themselves have long periods, on the order of 2^{127}. The benefit of this is that in parallel environments one can ensure that each processor does not have any overlap with the random number generators on the other processors. While this method has a somewhat shorter period than other proposed multiple recursive generators, it is well suited and easily implemented for parallel environments.

3.2. **Reproducibility With A Fixed Number Of Processors.** We first consider reproducibility assuming a fixed number of processors - for example, the econometrician will *always* use, say, four processors. This situation might arise in cases in which the econometrician has limited computational resources (say, a single quad-core desktop), and does not have the option of calling additional computational resources. This implies, however, that the econometrician will never desire to replicate the computation using fewer than four processors (say, two), and that any other researchers who might wish to reproduce the computations will need to know the number of cores (in addition to the seed) on which the computations were carried out. It is also worth mentioning that a researcher with limited computational resources, say, a dual-core laptop, will be unable to reproduce the results of a researcher with, say, a quad-core desktop.

In the case of a fixed number of processors, the method of L'Ecuyer (1999) ensures that the random number streams are different across each of the processors, and the calls `seed=123` (in the `R` packages - see the previous section, or the following examples) ensure that the random number streams are reproducible. In the simplest case, this method can ensure reproducibility because of the method in which computational jobs are allocated across the processors. For example, jobs are assigned across processors so that processor 1 receives jobs $\{1, 5, 9, ...\}$, processor 2 receives jobs $\{2, 6, 10, ...\}$, etc. Hence, for the setup with a fixed number of processors, and assuming that the parallel library allocates independent computational jobs across processors in an identical fashion, ensuring reproducibility of L'Ecuyer (1999) random number streams on each processor is akin to ensuring reproducibility of results.

The advantage of this approach is that the parallel libraries in `R` have convenient calls to implement the L'Ecuyer (1999) random number streams. The potential downsides of this approach are that the econometrician must be forthcoming about the seed, the number of cores used, and the statistical software and parallel package called for computation to ensure that the distribution of computational jobs will be assigned uniformly across the processors (this rules out load balancing parallel implementation - see the following subsection). Notice that this method of reproducibility is subject to multiple potential mishaps, and is, in general, not robust.

3.3. **Robust Reproducibility Of Parallel Computations.** Perhaps the more general approach to ensuring reproducibility of parallel computations is directly controlling the computational starting point of *every* computation. This approach is perhaps the most robust in that it allows computations to be reproducible with a different number of processors, or across different statistical software, parallel packages, or parallel call functions. It is worth emphasizing that ensuring reproducibility across a different number of processors includes ensuring reproducibility across both sequential and parallel environments, with the former being a special case of the later in which the number of processors in the parallel environment is unity. The reproducibility methods presented below can ensure reproducibility across both a sequential and parallel setup, in addition to the more general case of ensuring reproducibility across different sized parallel environments.

There are many instances in which computation with a different number of processors might be either desirable or necessary. Return to the examples given above, in which an econometrician may have different computational resources available at different times, and may be required by computational constraints to implement parallel computations under varying number of processors. Or, different researchers with different computational resources may seek to reproduce the same set of results.

Alternatively, the econometrician may wish to implement load balancing parallel environments to increase computational efficiency. In contrast to the method mentioned above in which each processor is allocated an even fraction of the total computational job upon initialization of the cluster and assignment of the task, load balancing parallel processing allocates the computational jobs to processors as they (the processors) become available. That is, in a four processor environment, only jobs 1-4 (out of, say, 1,000) are initially assigned to processors 1-4. Whichever processor finishes first receives job 5, and this process is continued until all jobs have been assigned. This is potentially more efficient because depending on the starting values for the econometric procedure (e.g., in a nonlinear optimization problem), different jobs may be completed faster. 'Faster' processors will receive more jobs to assist 'slower' processors to further expedite the computation. While load-balancing may be computationally efficient, it is clear that the L'Ecuyer (1999) method described above no longer ensures reproducibility as we can no longer guarantee or replicate which processor performs any particular computation.[4]

An alternative method that guarantees reproducibility of results requires manually setting the seed prior to each computation. This approach ensures that each computation has a fixed starting point, regardless of which processor is assigned the task. The simplest way of accomplishing this is to use the function index as the seed *within* each computation. Returning to our example in which we defined `function <- function(a){codes}` to be a generic wrapper function for each independent computation, one might use `a` as the seed index at the beginning of each computation:

```
> function <- function(a){
+       set.seed(a)
+       ...
+ }
```

so that the seed is different across each computation, and fully reproducible.

One particular concern might be that, using this method of reproducibility, the seeds are not random across each computation; the seeds range sequentially over `a`.[5] However, in principle, one might use this approach with a more elaborate method of setting each seed. For instance, one might randomly sample a sequence of integers without replacement to ensure that the vector of seeds for each computation were both random and unique. For example, resampling the sequence of natural numbers from 1 to 50,000 without replacement 1,000 times would generate a list of 1,000 unique seeds which would be implemented across each computation by using the random number corresponding to the number of computation as the seed - e.g., the 5th computation would use the 5th random number as the seed. This strategy might be implemented by

```
> set.seed(123)    # So the seed index is reproducible
> seed <- sample(seq(lower,upper),1000,replace=FALSE)
>
> function <- function(a){
```

[4]Each of the packages `multicore`, `snow` and `snowfall` have load-balancing routines built in, while not discussed in this review. We refer interested readers to the manuals accompanying each package for further details.

[5]A further concern could be overlap between the seeds and the state of the random number generator, implying that at different stages of the parallel problem the random numbers generated are identical to a substream of numbers at an earlier stage. Reducing the chance of this occuring amounts to using RNGs with long periods.

```
+       set.seed(seed[a])
+       ...
+ }
```

in which we generate our seed vector by resampling the sequence from `lower` to `upper` without replacement 1,000 times. Such a more elaborate approach might reassure the econometrician that the starting values are determined randomly, while still guaranteeing general reproducibility of the results.

An alternative, and perhaps the most robust, method for choosing the seed for each independent computation is to adopt the method of L'Ecuyer (1999) to define the vector of seeds to be used for each computation. The advantage of using the L'Ecuyer (1999) method for generating the seeds is that it is a well defined algorithm for generating unique seeds, and is invariant to default settings within different parallel packages and across different statistical software that may not be known or controllable by the econometrician. Resampling a sequence of integers, for instance, may rely on different random resampling algorithms across different software, and as such, may not be fully reproducible. Adopting the L'Ecuyer (1999) method of generating seeds overcomes this potential pitfall.

To adapt the L'Ecuyer (1999) method of generating seeds we call the `gather` function in the `R` package `harvestr` (Redd 2012). Specifically,

```
> gather(a, seed=123)
```

generates a different sets of 6-element L'Ecuyer (1999) seeds that are fully reproducible from our call to `seed=123`. The `set.seed` function built into `R` optionally accepts 6-element L'Ecuyer (1999) seeds with the call to `"L'Ecuyer-CMRG"`. One might use `gather` to generate 1,000 different sets of seeds, and call out the a^{th} set of 6-element seeds for the a^{th} computation for $a \in A$ computations. Specifically, this method of generating L'Ecuyer (1999) seeds and passing them into each parallel computation might be achieved as follows

```
> seed.temp <- gather(a,seed=123)
> Seed <- matrix(nrow=a,ncol=6)
>
> for(i in 1:a){
+       Seed[i,] <- seed.temp[[i]][2:7]
+ }
>
> function <- function(a){
+       seed.run <- Seed[a,]
+       set.seed(seed.run, "L'Ecuyer-CMRG")
+       ...
+ }
```

in which we first gather the list of 6-element L'Ecuyer (1999) seeds, place the gathered seeds into a matrix for ease of access, and set the seed within the parallel function to be the a^{th} row of our seed matrix, being sure to call the `"L'Ecuyer-CMRG"` seed function. In this setup, we are guaranteed that our seeds will be different across each independent computation and will be fully reproducible across any software or parallel package given the L'Ecuyer (1999) method, and as such, will form the foundation of generally reproducible parallel computations. We provide evidence of the reproducibility of a parallel Monte Carlo simulation using this adapted L'Ecuyer (1999) method in Section 4.

3.4. **Parallel Computations Using Sweave.** While the examples we will present here can be performed in relatively short order with just a few processors, many real application procedures will undoubtedly take much longer to run. Given the dominance of LaTeX in economic and econometric writing, Sweave has become a popular platform for combining computational results (that are run in `R`) with the LaTeX code. In the context of the computationally difficult parallel settings that are the subject of this review, the Sweave environment may seem undesirable since the computer code would need to be run for each compilation of the LaTeX document. Meredith & Racine (2009) show how one can cache `R` code within the LaTeX document, although they do not provide a detailed discussion.

Caching `R` code means that the one runs the computationally intensive `R` code a single time, and stores the results for Sweave to locate and weave into the LaTeX document each time the document is compiled. Hence, caching allows one to use Sweave without having to re-compute computationally demanding procedures each time the LaTeX document is compiled.

Prior to discussing the actual Sweave coding involving caching, we mention that when one caches code care must be taken to make sure the intended consequence is clear from the code. Caching is implemented by storing code chunks modification time in a metadata database. This database will then be used to provide results back to Sweave when the code is recompiled, so that unless modifications to the code have been made, the `R` chunks will not be run again. Only code chunks that do not specifically create an object for the global environment should be cached, because upon recompilation the code is not run again. Thus, when caching computational results from a parallel processing environment, it is better for figures and tables to be constructed in later chunks of `R` code so that they are not cached and will be rerun (and included) in the LaTeX document every time it is compiled.

Here we mention several points. First, in the `NOWEB` syntax that underlies Sweave (Ramsey 2008) if we wished to cache we would simply type `cache=TRUE`. Second, when we compile the .Rnw Sweave file we would need to tell the compile code that we wish to engage in caching. Several readily available packages to cache are `weaver` (Falcon 2006), `pgfSweave` (Bracken & Sharpsteen 2011) and `cacheSweave` (Peng 2012). For example, if our Sweave file was named Parallel.Rnw, we would use the following commands

```
> library(cacheSweave)
> Sweave("Parallel.Rnw",driver=cacheSweaveDriver())
```

to compile Parallel.Rnw using caching with the `cacheSweave` package.[6]

4. Examples

We now consider several common examples and illustrate how each can be run very simply in parallel using `multicore`, `snow` and `snowfall`. We use `parallel` to assist in L'Ecuyer (1999) random number generation for `multicore` and `snow`. We compare the performance of a two-processor and four-processor cluster with a single processor sequential counterpart by recording the computation speed for each computational setup. We record computation time using `system.time()`.

4.1. Bootstrap Standard Errors.

We first illustrate `multicore` on a residual bootstrap procedure for estimating the standard errors in a standard multivariate linear regression model. Applied econometricians frequently employ bootstrap procedures in a variety of econometric settings: when estimating standard errors in nonlinear or nonparametric regression analysis, or when approximating the finite sample distribution of a test statistic under the null hypothesis. In this example, we assume a simple regression model, but generate multivariate data with sample size $n = 30,000$ to reflect the large sample size often utilized in micro-level studies. We emphasize that substantial computational improvements that potentially impact research productivity are likely in more complex computational problems, perhaps involving more complex functional forms and relatively large datasets.

We generate our model via:

```
> ## Basic parameters
> n <- 30000
> B <- 1000
> ## Generate multivariate data
> x <- cbind(runif(n,2,8),
+            runif(n,2,3),
+            runif(n,-3,3),
+            runif(n,1,2),
+            runif(n,2,6))
> y <- 12+3.6*x[,1]+4*x[,2]-0.7*x[,3]+1.1*x[,4]-x[,5]+rnorm(n)
```

That is, we generate a multivariate linear regression model consisting of 30,000 observations and intend to estimate standard errors based on 1,000 bootstrap replications from an iid residual bootstrap. We might first run a linear regression and then collect the fitted values and residuals to send to the processors.

[6]An alternative to Sweave that may be useful in this context is the `knitr` package in `R` that performs many of the same functions as Sweave in RStudio.

We can define our bootstrap function, initialize the `multicore` environment, and carry out our parallel computation making sure to record computation time.

```
> ## Run linear model
> lm <- lm(y~x)
> ## Collect residuals and fitted values
> resid <- residuals(lm)
> fit <- fitted(lm)
> ## Bootstrap function
> boot <- function(B){
+
+     ## Sample residuals
+     resid.star <- resid[sample(seq(1:n),replace=TRUE)]
+
+     ## Construct bootstrap y
+     y.star <- fit + resid.star
+
+     ## Run bootstrap regression
+     lm.boot <- lm(y.star~x)
+
+     ## Return bootstrap coefficients
+     return(coefficients(lm.boot))
+
+ }
> ## Load library
> library(multicore)
> ## Determine available processors
> multicore:::detectCores()
> ## Set L'Ecuyer seed (runs from `parallel' package)
> set.seed(123,"L'Ecuyer")
> ## Run parallel computation and record computation time
> time.par <- system.time(mclapply(1:B,boot,mc.cores=2))
> ## Shut down parallel environment
> multicore:::closeAll()
> ## Run sequentially and record computation time
> set.seed(123)
> time.seq <- system.time(lapply(1:B,boot))
```

Here, we have set our list equal to a sequence $\{1 : B\}$, and named our bootstrap function `boot`. We have set the seed to be different across different processors (this step uses `parallel`), and have selected a two processor parallel environment. Implementing the example in the `multicore` environment yielded a computation time of 39.937 seconds.

```
> time.par[3]
elapsed
 39.937
```

A simple comparison to a sequential bootstrap procedure yields a computation time of 68.803 seconds.

```
> time.seq[3]
elapsed
 68.803
```

The single processor (sequential) setup required almost double the computation time compared to using the two processor cluster. One can imagine a more computationally complex problem that may take four weeks to compute sequentially. On the two processor parallel setup, computation time could be reduced to two weeks. Computation time could be reduced to approximately one week in a standard four processor environment. We emphasize that the bootstrap function sent to each processor is *identical* to the bootstrap function used in a sequential processing setup. The only difference is that in the cluster environment, the total number of bootstrap replications is being divided across the number of processors. It should be clear that the parallel environment can greatly reduce computation time.

4.2. Multiple Starting Values For Nonlinear Optimization.

In this example, we consider the problem of nonlinear numerical optimization. Such optimization problems arise frequently in many areas of applied econometrics, such as nonlinear least squares, maximum likelihood estimation, bandwidth selection in nonparametric and semiparametric kernel methods, to name only a few. Practitioners are ever wary of numerical solutions to nonlinear optimization problems, and typically, one must run a local optimizer multiple times to search for a robust solution. (This technique is known as multistarting.) Depending on the complexity of the model, the dimensionality of the optimization problem (e.g., the number of parameters to estimate in a basic nonlinear least squares problem), and the sample size, multistarting a local optimizer to obtain a robust solution is computationally intensive. Fortunately for practitioners, multistarting is also ‘embarrassingly parallel’.

We consider a simple Cobb Douglas model with additive error and wish to estimate the model parameters using nonlinear least squares. We assume a sample size of 2,500 observations, and implement the optimization procedure 10 times (10 multistarts).

```
> library(snow)
> set.seed(123)
> ## Basic parameters
> n <- 2500
> nmulti <- 10
> ## Generate model
> x <- cbind(runif(n,4,12),
+            runif(n,2,5),
+            runif(n,6,7))
> b <- runif(4)  # Coefficients
> y <- b[1]*x[,1]^b[2]*x[,2]^b[3]*x[,3]^b[4]+rnorm(n)
> ## Construct function to send to processors
> nls <- function(nmulti){
+
+     ## Define nls function
+     model <- function(a){
+
+         (1/n)*sum((y - a[1]*x[,1]^a[2]*x[,2]^a[3]*x[,3]^a[4])^2)
+
+     }
+
+     ## Construct solutions vector
+     solve <- numeric(length(b)+2)
+
+     ## Optimize and store results
+     optim <- optim(runif(length(b)),model,method="Nelder-Mead")
+     solve[1] <- optim$value
+     solve[2] <- optim$convergence
+     solve[3:(length(b)+2)] <- optim$par
+
+     ## Return solution
+     return(solve)
+ }
```

Now, using the cluster environment and the optimization function named `nls`, we can send the optimization problem to the processors and record computation time.

```
> ## Setup cluster
> cluster <- makeCluster(2,type="SOCK")
> ## Export objects
> clusterExport(cluster,c("n","x","b","y"))
> ## Set L'Ecuyer seed
> clusterSetupRNGstream(cluster,seed=123)
> ## Run parallel computation and record computation time
> time.par2 <- system.time(solution <- parLapply(cluster,1:nmulti,nls))
> ## Shut down cluster
> stopCluster(cluster)
> ## Obtain optimal solution with lowest function value
> solve.mat <- sapply(1:nmulti,function(i){cbind(solution[[i]])})
> min <- min(solve.mat[1,])
> b.hat <- solve.mat[3:(length(b)+2),solve.mat[1,]==min]
> ## Compare actual and estimated coefficients
> cbind(b,b.hat)
> ## Run sequential computation and record computation time
> set.seed(123)
> time.seq2 <- system.time(lapply(1:nmulti,nls))
```

In our example, the cluster environment yielded a computation time of 1.623 seconds,

```
> time.par2[3]
elapsed
  1.623
```

whereas the sequential setup yielded a computation time of 3.415 seconds.

```
> time.seq2[3]
elapsed
  3.415
```

Computation time was substantially faster in the parallel environment than in the sequential setup, which illustrates the impressive improvements in computational ability realized in the cluster environment. As in the previous example using `multicore`, computation time was approximately double in the sequential setup compared to the parallel setup. It is clear that the advantage of using `snow` increases significantly with the computational complexity of the problem, as well as in the number of processors employed in the parallel setup.

4.3. **A Monte Carlo Simulation.** In our final example, we conduct a Monte Carlo simulation to assess the size of the standard Hausman test, following Hausman (1978) and Amini, Delgado, Henderson & Parmeter (2012). Monte Carlo procedures are commonly used in econometric research to assess the finite sample properties of estimators and tests. Typically, researchers perform multiple simulations using different model specifications, estimators, or sample sizes. Any particular set of simulations can be computationally demanding, and an entire body of research based on simulations can be extremely computationally demanding. As is shown here, Monte Carlo methods are 'embarrassingly parallel' and lend well to these 'embarrassingly easy' parallel environments.

A simple parallel Monte Carlo simulation might be performed as follows. First, define the number of Monte Carlo replications to be 1,000. Then, set parameters to define a panel dataset with the number of individuals equal to 1,000, measured over 3 years. In this example, we employ a four processor parallel environment.

```
> library(snowfall)
> library(rlecuyer)    # for L'Ecuyer random number streams
> set.seed(123)
> ## Basic parameters
> M <- 1000
> n <- 1000
> T <- 3
> nT <- n*T
> ## Define Monte Carlo function
> mc <- function(M){
+
+     ## Library
+     library(plm)    # for fixed and random effects estimation
+
+     ## Generate data
+     tid <- rep(c(1:T),n)   # time id
+     nid <- matrix(t(matrix(rep(c(1:n),T),n,T)),nT,1,byrow=TRUE)   # firm id
+     xit <- runif(nT,-1,1)   # variable
+     mui <- runif(n,-1,1)   # fixed effect
+     vi <- matrix(t(matrix(mui,n,T)),nT,1,byrow=TRUE)   # error
+     yit <- 2*xit+vi+rnorm(nT)   # outcome
+
+     ## Setup panel data frame
+     pdata <- data.frame(y=yit,x=xit,nid=nid,tid=tid)
+     pdata <- pdata.frame(pdata,c("nid","tid"))
+
+     ## Fixed effects estimation
+     model.fe <- plm(y~x,data=pdata,model="within",effect="individual")
+
+     ## Random effects estimation
+     model.re <- plm(y~x,data=pdata,model="random",effect="individual")
+
+     ## Compute test
+     test <- phtest(model.fe,model.re)
+
+     ## Return p-value
+     return(test$p.value)
+
+ }
```

For this type of problem, all data are generated within the Monte Carlo function. Now to run the Monte Carlo simulations we call

```
> ## Setup parallel
> sfInit(parallel=TRUE,cpus=4)
> ## Export objects
> sfExport("n","T","nT")
> ## Set L'Ecuyer random number generation - requires `rlecuyer' package
> sfClusterSetupRNGstream(seed=123)
> ## Run parallel computation and record computation time
> time.par3 <- system.time(solution <- sfLapply(1:M,mc))
> ## Shut down cluster
> sfStop()
> ## Check size of 5 percent test
> length(solution[solution <= 0.05])/M
> ## Run sequential computation and record computation time
> set.seed(123)
> time.seq3 <- system.time(lapply(1:M,mc))
```

In this example, our parallel Monte Carlo took 157.703 seconds to compute

```
> time.par3[3]
```

```
elapsed
157.703
```

while are sequential computation was completed in 534.247 seconds.

```
> time.seq3[3]
```

```
elapsed
534.247
```

The parallel computation is completed approximately 3.4 times faster than the simulation in sequential mode (parallel computation is not exactly 4 times faster because of cluster initialization and shutdown time). Computation time on relatively more complex problems can be substantially reduced using a parallel environment, and the additional advantage of `snowfall` is the user-friendly set up around `snow`.

4.3.1. *Replicability Of The Monte Carlo Across Machines And Clusters.* To demonstrate the ability to replicate results determined in parallel we also ran the exact same Monte Carlo across three different machines each using a different (and non-divisible) number of processors and each using a different embarrassingly parallel package. We used a Mac running OS/X with seven processors and `multicore`, a Linux cluster using 13 processors and `snowfall` and an IBM running Windows using three processors and `snow`.

First, we set up the Monte Carlo code exactly as it was used previously, except to ensure replicability we use the `harvestr` package to construct random seeds for the L'Ecuyer random number generator. These seeds are then drawn *inside* our Monte Carlo function so that we ensure the same M simulations are being run across the three machines.

```
> ## Load packages
> library(harvestr)
> ## Basic parameters
> M <- 1000
> n <- 100
> T <- 3
> nT <- n*T
> ## Define seeds
> seed.temp <- gather(M,seed=123)
> Seed <- matrix(nrow=M,ncol=6)
> for(i in 1:M){
+
+     Seed[i,] <- seed.temp[[i]][2:7]
+
+ }
> ## Define Monte Carlo function
> mc <- function(M){
+
+     seed.run <- Seed[M,]
+     set.seed(seed.run, "L'Ecuyer-CMRG")   ## to make each run fully reproducible
+
+     ## Library
+     library(plm)    # for fixed and random effects estimation
+
+     ## Generate data
+     tid <- rep(c(1:T),n)   # time id
+     nid <- matrix(t(matrix(rep(c(1:n),T),n,T)),nT,1,byrow=TRUE)   # firm id
+     xit <- runif(nT,-1,1)   # variable
+     mui <- runif(n,-1,1)   # fixed effect
+     vi <- matrix(t(matrix(mui,n,T)),nT,1,byrow=TRUE)   # error
+     yit <- 2*xit+vi+rnorm(nT)   # outcome
+
+     ## Run test
+     pdata <- data.frame(y=yit,x=xit,nid=nid,tid=tid)
+     pdata <- pdata.frame(pdata,c("nid","tid"))
+
+     ## Fixed effects estimation
+     model.fe <- plm(y~x,data=pdata,model="within",effect="individual")
+
+     ## Random effects estimation
+     model.re <- plm(y~x,data=pdata,model="random",effect="individual")
+
+     ## Compute test
+     test <- phtest(model.fe,model.re)
+
+     ## Return p-value
+     return(test$p.value)
+
+ }
```

After the code has been appropriately modified we use the same commands as before. First run the Monte Carlo on a Mac using `multicore` with seven processors

```
> ## Load package
> library(multicore)
> seed.temp <- gather(M,seed=123)
> Seed <- matrix(nrow=M,ncol=6)
> for(i in 1:M){
+
+     Seed[i,] <- seed.temp[[i]][2:7]
+
+ }
> ## Run Monte Carlo
> solution <- mclapply(1:M,mc,mc.cores=7)
> ## Shut down cluster
> multicore:::closeAll()
> ## Store results
> result.mac <- unlist(solution)
```

Next run the Monte Carlo on a Windows system using `snow` and three processors

```
> ## Load package
> library(snow)
> ## Setup parallel
> cluster <- makeCluster(3)
> ## Export objects
> clusterExport(cluster,c("n","T","nT","Seed"))
> ## Run Monte Carlo
> solution <- parLapply(cluster,1:M,mc)
> ## Shut down cluster
> stopCluster(cluster)
> ## Store R\results
> result.IBM <- unlist(solution)
```

Finally, run the Monte Carlo on a Linux system using `snowfall` and thirteen processors.

```
> ## Load package
> library(snowfall)
> ## Setup parallel
> sfInit(parallel=TRUE,cpus=13)
> ## Export objects
> sfExport("n","T","nT","Seed")
> ## Run Monte Carlo
> solution <- sfLapply(1:M,mc)
> ## Shut down cluster
> sfStop()
> ## Store results
> result.lin <- unlist(solution)
```

It is easy to see across the three calls that we are producing identical Monte Carlo p-values and that the individual p-values have the same summary statistics.[7]

```
> ## Summary of the p-values from the Mac cluster
> summary(result.mac)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
0.0004364 0.2704000 0.5071000 0.5125000 0.7792000 0.9996000
> ## Size of the test from the Mac
> length(which(result.mac<0.05))/M
[1] 0.053
> ## Summary of the p-values from the IBM cluster
> summary(result.IBM)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
0.0004364 0.2704000 0.5071000 0.5125000 0.7792000 0.9996000
> ## Size of the test from the IBM
> length(which(result.IBM<0.05))/M
[1] 0.053
```

[7]This does not guarantee all 1000 values are identical but reporting all 1000 values for each of the three computers would be infeasible. Our hope is that the summary statistics suffices. Detailed results that verify that all 1,000 Monte Carlo replications have identical values are available upon request.

```
> ## Summary of the p-values from the Linux cluster
> summary(result.lin)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
0.0004364 0.2704000 0.5071000 0.5125000 0.7792000 0.9996000
> ## Size of the test from the Linux
> length(which(result.lin<0.05))/M
[1] 0.053
```

5. Conclusions

In this review, we have focused on several recently developed packages available in `R` with which to implement parallel computations on 'embarrassingly parallel' problems. We have demonstrated that there are essentially zero barriers to implementing many standard econometric procedures in parallel, and have shown that computation time can be drastically reduced on standard desktop/laptop computers without requiring access to large cluster computers, expensive software, or complicated knowledge of parallel algorithms. We note, of course, that users with access to a large cluster environment will also be able to use these packages, and likely reduce competition time by orders of magnitude.

As we have discussed, there are several reproducibility issues that arise within a parallel environment, and reproducibility requires the ability to control the computational starting values for *each* independent calculation. We have provided several alternative approaches that are able to guarantee reproducibility in a parallel environment. Through this discussion, we have extended the important discussions of Koenker & Zeileis (2009) and Meredith & Racine (2009) into the parallel computing environment, and have provided several examples of how applied econometricians can adapt these techniques into their research at very little cost.

We have demonstrated the usefulness and simplicity of each of these three packages using three separate 'embarrassingly parallel', yet commonly implemented econometric procedures - the bootstrap, nonlinear optimization, and a Monte Carlo simulation. We have shown that each of these procedures can be implemented with very few modifications to existing computer codes. We have shown how computation time in a two processor and four processor parallel environment is half (or one fourth in the four processor setup) the standard computation time when using a sequential (single processor) set up. Clearly, the reduction in computation time can lead to very large improvements in efficiency and productivity for many applied econometricians.

Computational Resources

All computations were conducted using `R 2.15.1`, `snow` version 0.3-10, `snowfall` version 1.84, and `multicore` version 0.1-7. Computations were conducted on a Windows 7 desktop with Intel i7-2600 CPU @ 3.40GHz, a dual 8-core Intel Xeon-E5 Linux cluster, and a dual quad-core Mac OS/X desktop with Intel Core i7 @ 2.93Ghz.

References

Amini, S., Delgado, M. S., Henderson, D. J. & Parmeter, C. F. (2012), Fixed vs. random: The Hausman test four decades later, *in* 'Advances in Econometrics: Essays in Honor of Jerry Hausman', Vol. 29, Emerald Group Publishing, pp. 479–513.

Bracken, C. & Sharpsteen, C. (2011), *pgfSweave: Quality speedy graphics compilation and caching with Sweave*. R package version 1.2.1.
URL: *http://CRAN.R-project.org/package=pgfSweave*

Creel, M. (2005), 'User-friendly parallel computations with econometric examples', *Computational Economics* **26**, 107–128.

Creel, M. (2007), 'I ran four million probits last night: HPC clustering with Parallelknoppix', *Journal of Applied Econometrics* **22**, 215–223.

Eugster, M. J. A., Knaus, J., Porzelius, C., Schmidberger, M. & Vicedo, E. (2011), 'Hands-on tutorial for parallel computing with R', *Computational Statistics* **26**, 219–239.

Falcon, S. (2006), *weaver: Tools and extensions for processing Sweave documents*. R package version 1.24.0.

Hausman, J. A. (1978), 'Specification tests in econometrics', *Econometrica* **46**, 1251–1271.

Hayfield, T. & Racine, J. S. (2012), 'Parallel nonparametric kernel smoothing methods for mixed data types', *http://cran.r-project.org/web/packages/npRmpi/npRmpi.pdf* .

Ho, A. T. Y., Huynh, K. P. & Jacho-Chávez, D. T. (2011), 'npRmpi: A package for parallel distributed kernel estimation in R', *Journal of Applied Econometrics* **26**, 344–349.

Knaus, J. (2012), 'Easier cluster computing (based on snow)', *http://cran.r-project.org/web/packages/snowfall/snowfall.pdf* .

Koenker, R. & Zeileis, A. (2009), 'On reproducible econometric research', *Journal of Applied Econometrics* **24**, 833–847.

L'Ecuyer, P. (1999), 'Good parameters and implementations for combined multiple recursive random number generators', *Operations Research* **47**, 159–164.

L'Ecuyer, P. & Simard, R. (2001), 'On the performance of birthday spacings tests for certain families of random number generators', *Mathematics and Computers in Simulation* **55**, 131–137.

L'Ecuyer, P., Simard, R., Chen, E. J. & Kelton, W. D. (2002), 'An object-oriented random-number package with many long streams and substreams', *Operations Research* **50**, 1073–1075.

Matsumoto, M. & Nisimura, T. (1998), 'Mersenne twister: A 623-dimensional equidistributed uniform pseudo-random number generator', *ACM Transactions on Modeling and Computer Simulation* **8**, 3–30.

Meredith, E. & Racine, J. S. (2009), 'Towards reproducible econometric research: The Sweave framework', *Journal of Applied Econometrics* **24**, 366–374.

Peng, R. D. (2012), *cacheSweave: Tools for caching Sweave computations*. R package version 0.6-1.
URL: *http://CRAN.R-project.org/package=cacheSweave*

Ramsey, N. (2008), *Noweb: a simple, extensible tool for literate programming*.
URL: *http://www.eecs.harvard.edu/nr/noweb/*

Redd, A. (2012), 'A parallel simulation framework', *http://cran.r-project.org/web/packages/harvestr/harvestr.pdf* .

Schmidberger, M., Morgan, M., Eddelbuettel, D., Yu, H., Tierney, L. & Mansmann, U. (2009), 'State of the art in parallel computing in R', *Journal of Statistical Software* **31**, 1–27.

Swann, C. A. (2001), 'Software for parallel computing: The LAM implementation of MPI', *Journal of Applied Econometrics* **16**, 185–194.

Swann, C. A. (2002), 'Maximum likelihood estimation using parallel computing: An introduction to MPI', *Computational Economics* **19**, 145–178.

Tierney, L., Rossini, A. J., Li, M. N. & Sevcikova, H. (2012), 'Simple network of workstations', *http://cran.r-project.org/web/packages/snow/snow.pdf* .

Urbanek, S. (2012), 'Parallel processing of R code on machines with multiple cores or CPUs', *http://cran.r-project.org/web/packages/multicore/multicore.pdf* .