3-5-2003

Simple Parallel Statistical Computing in R

Anthony Rossini
University of Washington, blindglobe@gmail.com

Luke Tierney
University of Iowa, luke-tierney@uiowa.edu

Na Li
University of Washington, nali@umn.edu

1 Introduction

Modern computer processors are now sufficiently powerful to make many statistical computations seem instantaneous. However, important situations still exist where a single result can require days to compute. As a result, either less accurate but faster methods are used, or potentially important computations are skipped entirely. Either approach can compromise the veracity of the resulting inference. Many common computationally demanding statistical procedures, such as bootstrapping and cross-validation, involve independent repeated computations with slightly different inputs. Problems of this form are sometimes referred to as *embarrassingly parallel*. If multiple processors or computers, which are often available in organizations where statistical computations are carried out, could be easily harnessed to perform these computations in parallel, then computations that would otherwise take hours or days could be performed in minutes or hours.

Despite some notable early efforts (e. g. Schervish (1988) and see also Appendix B) parallel computing does not appear to have been used extensively by statisticians. Toolkits and libraries for parallel computing on shared memory and distributed memory parallel systems, as exemplified by clusters of workstations, have been available for many years. These include the PVM (Geist et al., 1994) and MPI (Message Passing Interface Forum, 1995, 1997) message passing libraries. These toolkits work in terms of abstract basic computational units which we will call *compute nodes*. Compute nodes can be viewed as the abstraction of computers evaluating serial tasks which are components of a parallel computation. They might all run on a single machine with one or more processors, or on multiple such machines connected by a communications network. The toolkits typically provide functions and tools for message passing, communication management, and administrative tasks such as starting, configuring, and stopping a cluster of compute nodes. They are designed to support sophisticated parallel algorithms with complex communications needs, and were initially developed for use with low level compiled languages such as C and FORTRAN. The infrastructure provided by these toolkits, or even by a basic socket interface, is more than adequate for managing the scatter/compute/gather steps needed for handling most embarrassingly parallel problems. But the details of what needs to be done to manage communications and interface with computer languages in which the statistical algorithms of interest are written, are sufficiently complex to deter many potential users from directly applying these toolkits for particular computations.

This paper presents a set of simple tools for implementing embarrassingly parallel computations using the high level statistical computing language R (R Development Core Team, 2002; Ihaka and Gentleman, 1996). We identify a useful set of primitive building blocks and show, using several realistic examples, how these can be used to parallelize computations. The high level interface can be used with any of three low level communication mechanisms: sockets, MPI, or PVM. The details of which mechanism is used and how it is used are hidden from the high level user. This flexibility and opacity extend to the underlying parallel computational engine, which might be a high-level SMP system, a dedicated cluster (often called a Beowulf cluster), or a collection of heterogeneous workstations. With suitable support software, such as the Globus toolkit (Globus Project, 2002), it should also be possible to use a computational grid (Foster and Kesselman, 1998) as the engine.

2 Basic Tools and Usage

The parallel computing framework described in this paper is implemented in the R package `snow` (Simple Network of Workstations), which is available on the Comprehensive R Archive Network (CRAN). The framework is based on a master/slave design: A master R process, running either interactively or as a batch process, creates a cluster of slave R processes that perform computations on behalf of the master. Communication between master and slaves can use a socket interface, the PVM message passing framework via the `rpvm` package (Li and Rossini, 2002, 2001), or the MPI message passing framework via the `Rmpi` package (Yu, 2002a,b). The appropriate choice of communication mechanism should be based on local considerations; this will be discussed further in Appendix A. The basic routines for using a computational cluster are summarized in Table 1.

[Table 1 about here.]

Evaluating the expression

```
cl <- makeCluster(2)
```

results in a master process which specifies and creates a cluster of two slave processes. The return value is used to communicate with the cluster. The exact specification of the cluster nodes to be used for computation need not be provided. This separates cluster programming and management from the algorithm and allows the statistician to ignore the particular cluster configuration and computational grid management tools deployed. Since they are very important to implementation, but not the primary interest of this paper, we discuss some of the issues involved in the selection of cluster management tools and libraries in Appendix A.

The primary functions for computing on the cluster are `clusterEvalQ`, `clusterCall`, and `clusterApply`. `clusterEvalQ` evaluates a literal expression on each cluster node and returns a list of the results. It is the cluster version of the R function `evalq` and is suitable for evaluating top level expressions on all nodes. For example,

```
clusterEvalQ(cl, library(boot))
```

can be used to load the `boot` package on all compute cluster nodes.

`clusterCall` calls a specified function with identical arguments on each node in the cluster. The arguments are evaluated on the master, their values are transmitted to the slave nodes along with the function, and the slaves execute the function calls. A list of the values obtained on the nodes is returned. Using `clusterCall`, we define a function to obtain information about all nodes as

```
clusterInfo <- function(cl, ...) {
    info <- function(...) Sys.info()[...]
    infoList <- clusterCall(cl, info, ...)
    do.call("rbind", infoList)
}
```

The machine name and type for each node can then be obtained by

```
> clusterInfo(cl,c("nodename","machine"))
     nodename                machine
[1,] "owasso.stat.uiowa.edu" "Power Macintosh"
[2,] "itasca"                "i686"
```

In this example, one node is running on a Power Macintosh computer running Macintosh OS X and the other on an Intel 686 computer running Linux. This definition of `clusterInfo` combines the results of the parallel evaluation into a more usable form than a raw list of values; this is often useful when defining higher level parallel functions.

Finally, `clusterApply` takes a cluster, a list of arguments, and a function, and calls the function with the first element of the list on the first node, with the second element of the list on the second node, and so on. A list of the results is returned. The list of arguments must have at most as many elements as there are nodes in the cluster. A simple example:

```
> clusterApply(cl, list(1,2), rnorm)
[[1]]
```

```
[1] -1.58173

[[2]]
[1]  0.3399294 -1.8955971
```

This function is useful both directly and as a building block for creating higher level functions. For example, we can construct a parallel version of the R function `lapply`, for applying a specified function to each element of a list and returning a list of the results, as

```
parLapply <- function (cl, x, fun, ...) {
    xs <- clusterSplit(cl, x)
    do.call("c", clusterApply(cl, xs, lapply, fun, ...))
}
```

The function `clusterSplit` splits a list into contiguous pieces, one for each node in the cluster. By default, the pieces are as close to equal in size as possible. Other possibilities are discussed in Section 5.2. The `apply` family of R functions are natural candidates for parallelization, and the `snow` package contains parallel versions named `parLapply`, `parSapply`, and `parApply`.

The function `stopCluster` needs to be called at the end in order to cleanly shut down the cluster and clean up any remaining connections between machines. This function and `makeCluster` are the only user-visible connections to the parallel computation engine from R, taking care of initialization of slave R processes as well as shutting down and disconnecting them from the communication mechanism.

3 Parallel Random Number Generation

The generation of streams of random variates is critical for many parallelizable statistical computations. When no explicit seed is supplied, the R random number generators are seeded using the system clock value at the time that the generator is first called. This is quite likely, though not guaranteed, to produce identical random number streams on all cluster nodes. If we do want to use identical random number streams on the nodes then we can set a common seed on all nodes to insure that this will be the case. However, in many cases we would like to use independent random number streams on the nodes.

One approach to generating independent streams on the nodes is to use the Scalable Parallel Random Number Generators library (SPRNG, Mascagni and Srinivasan, 2001). An R interface is provided by the `rsprng` package (Li, 2002). The SPRNG library can initialize a fixed number of streams or it can be used to dynamically spawn streams as needed. The state of the generator can be saved and restored to guarantee the reproducibility of simulations. Several choices for the underlying generators are available. The number of available streams is very large, e.g. 2^{39648} for the default Modified Lagged Fibonacci generator.

The `snow` package provides a simple interface, `clusterSetupSPRNG`, for initializing one SPRNG stream for each node. When `clusterSetupSPRNG` is called with only the cluster reference `cl` as its argument, it initializes SPRNG with a random seed drawn from the current generator on the master.

```
> clusterSetupSPRNG (cl)
> clusterCall (cl, runif, 3)
[[1]]
[1] 0.014266542 0.749391854 0.007316102

[[2]]
[1] 0.8390032 0.8424790 0.8896625
```

We can insure that we obtain the same set of independent streams on a future repetition of the parallel run by specifying an explicit `seed` argument:

```
> clusterSetupSPRNG(cl, seed=1234)
> clusterCall (cl, runif, 3)
[[1]]
[1] 0.59831947 0.09220296 0.56194440

[[2]]
[1] 0.05054199 0.27565455 0.48012810

> clusterSetupSPRNG(cl, seed=1234)
>  clusterCall (cl, runif, 3)
[[1]]
[1] 0.59831947 0.09220296 0.56194440

[[2]]
[1] 0.05054199 0.27565455 0.48012810
```

More extensive control over the SPRNG streams is available through the low-level functions provided in the `rsprng` package. A potential alternative to the SPRNG library, which may be supported by `snow` in the future, is provided by the routines described in L'Ecuyer et al. (2002).

4 Examples

We now present several examples that illustrate the process and benefits of parallelizing suitable sequential R code. The examples are drawn from both help pages to existing R packages and from the book *Modern Applied Statistics with S* (Venables and Ripley, 2002). The computations in this section were carried out using a cluster of identical dual processor 1.5GHz Intel Xeon computers running RedHat Linux 7.3. Timings were computed with the function `wallTime`, which is defined as

```
wallTime <- function(expr) system.time(expr)[3]
```

4.1 A Parallel Bootstrap

Bootstrapping is a classic example of a time-consuming but simple-to-parallelize computation. The `boot` package provides a framework for generating bootstrap replicates of an arbitrary R function applied to data. One of the examples provided in the documentation, based on Example 6.8 of Davison and Hinkley (1997), generates bootstrap replicates of a generalized linear model fit for data on the cost of constructing nuclear power plants. Producing 1,000 bootstrap replicates on a single processor (1.5GHz Xeon) takes approximately 27 seconds:

```
> wallTime(boot(nuke.data, nuke.fun, R = 1000, m = 1,
+     fit.pred = new.fit, x.pred = new.data))
[1] 26.86
```

This computation can be split onto a cluster using `clusterCall`. For a cluster of 10 nodes with 100 replicates generated on each node the result is nearly a ten-fold speed-up:

```
> cl<-makeCluster(10)
> clusterSetupSPRNG(cl)
> clusterEvalQ(cl, library(boot))
> wallTime(clusterCall(cl, boot, nuke.data, nuke.fun, R = 100,
+  m = 1, fit.pred = new.fit, x.pred = new.data))
[1] 2.95
```

Figure 1 shows the speed-up obtained for clusters of several different sizes.

[Figure 1 about here.]

The speed-up is nearly linear in the number of processors; the small deviation from linearity is due to communication costs and slight variability in the run times of the individual jobs.

These examples serve to illustrate the performance gains available by running a bootstrap in parallel. However, they do not yet represent a complete solution. Since `clusterCall` returns a list of separate bootstrap results, a complete implementation of a parallel bootstrap function would merge these into a single bootstrap result structure. The result structure returned by `boot` is somewhat complex, so we will illustrate this using the simpler setting of the next example.

4.2 Parallel Kriging

The package `sgeostat` is one of several R packages that provides a routine for spatial prediction or kriging. The function `krige` provided in this package uses a pure R implementation that loops over the rows of a data frame of points at which predictions are requested. As a result, it is quite easy to construct a parallel version of this function: split the prediction points into one chunk for each node, use `clusterApply` to call `krige` on each chunk on its node, and assemble the results from the nodes into the form of the value returned by `krige`. The resulting parallelized function is

```
parKrige <- function(cl, s, ...) {
    # split the prediction points s
    idx <- clusterSplit(cl, 1: dim(s)[1])
    ssplt <- lapply(idx, function(i) s[i,])

    # compute the predictions in parallel
    v <- clusterApply(cl, ssplt, krige, ...)

    # assemble and return the results
    s.o <- point(s)
    s.o$zhat <- do.call("c", lapply(v, function(y) y$zhat))
    s.o$sigma2hat <- do.call("c", lapply(v, function(y) y$sigma2hat))
    return(s.o)
}
```

The `krige` help page presents an example of predicting zinc concentration in ground water on a 50×50 grid for a flood plane of the river Maas using only observation within 1,000 meters of a prediction point. Using a single process the computation takes a little under 10 seconds:

```
> wallTime(krige(grid$point,maas.point,'zinc',maas.vmod,
+                 maxdist=1000,extrap=FALSE,border=maas.bank))
[1] 9.61
```

The computation takes approximately 1.6 seconds using 10 processors :

```
> wallTime(parKrige(cl,grid$point,maas.point,'zinc',maas.vmod,
+                    maxdist=1000,extrap=FALSE,border=maas.bank))
[1] 1.61
```

Figure 1 shows speed-ups obtained for the problems on clusters of several different sizes.

The speed-up achieved in this example is less than in the bootstrapping example because there is more variation in computation times across nodes. Figure 2 (a) is a Gantt chart which displays CPU usage over time for `parKrige` with a five node cluster; this chart comes from the PVM visualization tool `xpvm`. This display demonstrates that some of the nodes finish their work early and spend time idling while others complete their computations. Better utilization of the processors is possible by using `clusterApplyLB`, a dynamic load balancing version of `clusterApply`. When called with more list elements *n* than cluster nodes *p* the jobs for the initial *p* list elements are placed on the processors, and the remaining jobs are assigned to the first available processor until all *n* jobs are complete. Figure 2 (b) demonstrates a variant of `parKrige` which uses `clusterApplyLB` with four times as many jobs as processors. Here, the workload is evenly distributed and the total computing time is further reduced.

[Figure 2 about here.]

Dynamic load balancing can improve cluster utilization in many problems. It can also be useful for effectively utilizing inhomogeneous clusters in which nodes differ significantly in performance. However, care is needed in choosing the size of the problem to compute. As the units of computation decrease in size, total cluster utilization increases. However, this also increases communications, which penalizes the total compute time. One other issues that arise with dynamic load balancing is that the compute node which executes a particular job is no longer deterministic. Some care is needed in parallel simulations if the simulations are to use dynamic load balancing and remain reproducible. These issues are discussed further in Section 5.2.

Other R-based kriging implementations are based on C code and are thus faster. Nevertheless, for larger problems these may also benefit from parallelization. More elaborate approaches for parallelizing kriging computations are also possible through the use of additional communication between slave node processes (Gebhardt, 2001), which cannot be supported by the master/slave relationship provided by `snow`. Identifying similar approaches to parallelization and researching appropriate APIs is part of the future plans for `snow`.

4.3 Parallel Cross Validation

Venables and Ripley (2002) discuss the use of cross validation to choose the number of hidden units and amount of weight decay for fitting neural networks. They present functions for carrying out the cross validation which consists of several nested loops. The inner-most loop repeats each neural network fit several times and the next inner-most loop, the cross validation loop, iterates over the observations to omit from the fit and then predict. The outer loops iterate over different settings of the number of hidden units and values for weight decay.

One approach for parallelizing this computation is to start with the cross validation loop. From a practical perspective, this is most easily done in two stages. The first step, vectorization, extracts the body of the loop into an auxiliary function and replaces the loop by a call to `lapply`. After testing and debugging this modified sequential function, a cluster argument can be added and the call to `lapply` is replaced by a call to `parLapply`. The timing for the sequential version (Venables and Ripley, 2002, page 249) is 262.6 seconds. The parallel version using ten nodes takes 35.1 seconds.

This particular example uses ten-fold cross validation and so parallelizing the cross validation loop produces only 10 separate jobs. This large granularity results in the fluctuations which were seen in Figure 1. While load balancing might result in more efficient utilization of the nodes, it would be necessary to unroll the two inner loops, distribute the resulting jobs on the cluster, and collect the results. The programming effort would be more substantial than the effort required for the approach outlined here, but may be worth while for producing a general purpose parallel cross validation tool.

4.4 Parallel Profile Likelihood

The final example that we consider uses the `proflik` function in the package `geoR` for computing profile likelihoods of parameters of Gaussian random field models. The profile likelihood for a particular parameter is obtained by fixing that parameter at several different values and, for each of these values, maximizing the likelihood over the remaining parameters. The function `proflik` uses the optimization function `optim` to carry out these optimizations, and iteration over the set of values of the parameter to be profiled is accomplished using the function `apply`.

Profile likelihood computation can be quite time consuming because of the repeated maximizations involved. For the example provided in the `proflik` help page, computing the profile likelihood of the sill parameter at 16 points takes a little over one minute. Since the iterations use `apply` it seems natural to parallelize the computations by replacing the call to `apply` by a call to `parApply`. This is possible, but some complications due to the coding style used in this function need to be addressed first.

The profiling optimizations need to pass certain information to the objective function that is given to the optimizer `optim`. `proflik` uses global variables to pass this information. Unfortunately these global variables are only defined on the master, and therefore an initial attempt to parallelize `proflik` by just using `parApply` in place of `apply` will fail. One solution is to transfer the required global variables to the cluster nodes. A better solution is to take advantage of lexical scope and define the objective function for `optim` in an environment containing the required additional variables as local variables. The resulting function closure contains both the function definition and its defining environment. When a function closure is passed from the master to a slave node, a copy of the entire closure consisting of code and local environment is transferred. Thus all the information needed by the objective function will be available on the slave nodes.

The practical activity of parallelizing the `proflik` function is done in two stages. First, we eliminate the use of global variables by defining the objective functions in suitable local environments that are captured under lexical scoping rules. Second, after testing and debugging the resulting new sequential version of `proflik`, we replace the calls to `apply` by calls to `parApply` and obtain a parallel version. Using eight nodes the parallel version computes the 16-point profile likelihood mentioned above in approximately 9 seconds. The results for other cluster sizes shown in Figure 1 are again rather uneven due to the coarse granularity of the problem: There are only 16 independent jobs, so using 10 processors provides no speed-up over using eight.

5 Performance Considerations

Ideally, one would hope that a parallelized computation on *P* processors would run *P* times faster than if using only one processor. This ideal is only rarely achieved. The primary reasons are communications overhead, task complexity, and uneven node performance. The last is especially true in clusters composed of different types of machines, which can be found in ad-hoc clusters formed from available machines in an academic department.

5.1 Communication

All distributed parallel computations require network communication to distribute data and collect results. Even in dedicated clusters with high speed networking, communication is orders of magnitude slower than computation. Communication is typically even slower in networked workstation environments. In some cases it may be possible to improve overall performance through the use of data compression/uncompression at the communications layer, or by using more complex communication architectures such as a tree of nodes. But the most effective approach to reduce communication costs is to eliminate unnecessary data transfer.

It is necessary to understand the characteristics of the data that is being transfered to optimize communication. One of the costs of the high level view provided by a language like R is that it is possible to lose sight of details that have a significant impact on performance. One good example is an early experience using the `snow` package to compute, in parallel, a Random Forest (Liaw, 2002). Superficially, this computation is completely parallelizable, and thus performance improvements similar to those experienced with parallel bootstrapping were expected. However, the initial attempt resulted in a parallel version running several times slower than the serial version. Close examination revealed that the slowdown was due to the fact that the result value returned by the `randomForest` function contained the full forest that had been produced, a data structure many megabytes in size. The cost of transmitting these structures dominated the parallel computation. Omitting the forests from the result, or performing any computations for which they were needed on the slave nodes before returning the reduced results, produced a parallelized version of the algorithm exhibiting the originally anticipated performance gains.

One source for unexpected data transfer is found in the capturing of data by environments of local functions. When function closures (Gentleman and Ihaka, 2000), which consist of the function definition and the environment in which the function was created, are serialized for transmission the local frames of their environments are included in the serialization. This can be very useful, as illustrated in Section 4.2. However, it can lead to unnecessary communication if these environments contain large amounts of data that are not needed on the remote nodes. Careful design of nested function definitions can insure that only the local variables actually needed for a computation are visible to the nested function. While this can manually alleviate this problem, hopefully code analysis tools will be available in the future to assist with this task.

Visualization tools such as `xpvm` can be very helpful in understanding the amount of communication that occurs in a parallel computation. One example of its application can be found in the charts in Figure 2. Similar tools exist for MPI, though most seem to be commercially licensed.

5.2 Load Balancing

As tasks become more complex, for example by requiring more initialization or communication, optimizing a parallel computation requires some thought on how to balance the load across available processors. Suppose that P processors are available to use for computation. The simple approach for static load balancing divides the computation into $K \leq P$ subprocesses, each of which is run on a single processor; `snow` provides the function `clusterSplit` for assistance. This approach maintains ease of implementation and is easily reproducible by providing initial values for the procedure, such as optimization starting points and random number generator seeds, for each process. Some nodes may be known to be faster than others, and we are examining how to use this information to allow `clusterSplit` to chose the proportion of a vector assigned to each node. We can extend this approach to a more dynamic load balancing solution by dividing the problem into $L > P$ subprocesses. This relaxes pre-specification of the node where the computational unit will be evaluated, but not necessarily the configuration or content to be evaluated. As demonstrated in the kriging example of Section 4.2, this can decrease total computation time by adapting to the variability in the computation time of individual tasks. This approach can also adapt to varying speeds among compute nodes without requiring advance knowledge about performance or load differences. This approach requires more thought in order to obtain reproducibility, for example by using L independent random number streams tied to the L separate tasks instead of one stream for each of the P processors.

Truly adaptive load balancing strategies would be optimal for ensuring maximum decrease in computation time. These solutions relax the requirement to pre-specify the computational units to evaluate. However, maintaining reproducibility, which is critical for load balancing a statistical process, is very difficult to do in this case, and there does not appear to be a solution for which a simple API can be easily developed.

6 Discussion

Through the use of relevant examples, we have shown that interesting statistical computations can be parallelized, often with a nearly linear speed-up rate. We have discussed some of the concerns that must be addressed when parallelizing tasks — communication, load balancing, and the merging of results — and presented a few ways to address them.

A typical strategy for parallelizing computation in R using `snow` involves the following steps: 1) identify the loops; 2) vectorize the loops by isolating the repeated computation in the loop into a workhorse function, and test and debug this function in a sequential setting; 3) write a combine or gather function to piece together the results of these repeated computations; 4) call appropriate `snow` functions to distribute the execution of the workhorse function. Note that once `snow` and its supporting packages have been properly installed on all machines in the cluster, there is nothing else that needs to be done for configuration. The use of a high-level language such as R also minimizes the amount of software that needs to be written.

In contrast, developing a parallel application in a more conventional language such as C or FORTRAN is often much more complicated. Parallelizing an existing sequential application often requires a complete rewrite and associated debugging. Maintaining codebases for both sequential and parallel versions is difficult, and extensions to different parallel APIs adds another layer of complexity. Any change in code requires propagating the modified source code and recompiling and deploying the executables on all nodes, a particularly challenging task if multiple architectures are involved. Finding communication bugs in parallel applications is not easy, whereas `snow` provides much of the parallel communications code, protecting the statistician from needing to know and manage these details.

There are still many extensions to consider and evaluate. The first class of extensions are those to the application programmers interface (API). We have focused on simplicity for the initial API, while maintaining sufficient generality to be applicable to a wide range of statistical computations. Future work will address generic approaches to more complex parallelization strategies, including divide-and-conquer (branch-and-bound) heuristics as exemplified by the quick-sort algorithm, and complete communication, as exemplified by by finite element analysis. Newer grid computing toolkits such as HARNESS, and GLOBUS should be supported. Finally, more statistics-related functionality, such as tighter integration with the bootstrap package, approaches for MCMC sampling, and optimization will be important; design of the API to implement and integrate these generic statistical tools will be critical to insuring widespread adoption.

Cluster computation may not reduce the computational time for every algorithm. Some computational problems cannot be broken down easily, or, when broken down, do not scale well for reasons including data transfer rates and variation in the times for individual processes to complete. Communication protocols and hardware on one computational cluster may adversely affect the results of a computation which was successfully deployed on a different cluster. Cluster computing is most efficient when the task can be divided into small, long-running sub-tasks which require little data to be communicated between nodes. For these tasks, cluster computing can noticeably reduce computation time.

`snow` is deployable on nearly any network of machines running Unix-like operating systems which share common user accounts. Perhaps the most important result obtained is that that the examples have been run on three different clusters (one using two configurations) at two universities, with no modification other than one-time local installation and configuration of the compute cluster libraries. There is no requirement for specialized hardware, and in fact, we have deployed this system in heterogeneous Unix-based environments, using different processors (Intel, AMD, PowerPC) as well as operating system kernels (Intel-Linux, PPC-Linux, Mac OS X), and operating systems (Debian Linux, Red Hat Linux, Mac OS X 10.1 and 10.2). If PVM or MPI are used, then all systems must of course use compatible versions of the same communications library. Fortunately many Linux distributions, including Debian and Red Hat, have pre-compiled packages for both PVM and LAM-MPI, making installation and configuration of `snow` reasonably simple. We are currently exploring several options for using computers running Microsoft operating systems. A master running on a Windows system can easily use the socket version of `snow` to run slave nodes on Unix-based systems. Using Windows systems for slave nodes in a socket cluster requires installation and configuration of remote shell support, such as the `sshd` daemon for Cygwin (Cygwin). A version of PVM is also available for Microsoft operating systems. We are currently exploring how this can be used in pure Windows environments as well as mixed Windows/Unix settings.

In summary, we have presented a simple system for parallelization which many statisticians will be able to use immediately to decrease computational processing time for many computationally intensive problems. We consciously do not provide a solution flexible enough to express all classes of parallel algorithms. Instead we focus on a system that is very easy to learn to use, yet is powerful enough to express a large class of important parallel computations. Parallelizing a computation reduces computing time, but the effort required to develop and deploy a parallel solution must be considered as well. The simplicity of the `snow` system helps to reduce this effort. It is therefore an effective framework for statistical research projects which create and evaluate computationally intensive statistical procedures, and for distributing the resulting tools to other users.

Acknowledgments

We wish to thank Andy Liaw for bringing the Random Forest example to our attention.

References

Moshe Bar. OpenMosix www page. WWW, 2002. URL `http://www.openmosix.org/`.

A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13:361–372, 1998.

Jim Basney and Miron Livny. Deploying a high throughput computing cluster. In Rajkumar Buyya, editor, *High performance cluster computing: Architectures and Systems*, volume 1, pages 116–134. Prentice Hall, 1999.

Luca Cardelli. A language with distributed scope. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 286–297, 1995.

Cygwin. Cygwin home page. WWW, 2003. URL `http://www.cygwin.com/`.

A.C. Davison and D.V. Hinkley. *Bootstrap Methods and Their Application*. Cambridge University Press, 1997.

Sandhya Dwarkadas, Alejandro A. Schäffer, Robert W. Cottingham Jr., Alan L. Cox, P. Keleher, and Willy Zwaenepoel. Parallelization of general-linkage analysis problems. *Hum. Hered.*, 44:127–141, 1994.

Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

Albrect Gebhardt. pvmkrige. WWW, 2001. URL `ftp://ftp-stat.uni-klu.ac.at/pub/R/contrib`.

Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam, editors. *PVM: Parallel Virtual Machine. A user's guide and tutorial for networked parallel computing.* MIT Press, Cambridge, Massachusetts, 1994.

Robert Gentleman and Ross Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9:491–508, 2000.

Charles J. Geyer and Elizabeth A. Thompson. Annealing Markov chain Monte Carlo with applications to ancestral inference. *J. Amer. Statist. Assoc.*, 90:909–920, 1995.

Globus Project, 2002. The Globus Project. WWW, 2002. URL `http://www.globus.org/`.

Sandeep K. Gupta, Alejandro A. Schäffer, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Integrating parallelization strategies for linkage analysis. *Comput. Biomed. Res.*, 28:116–139, 1995.

G. Hains and F. Loulergue. Functional Bulk Synchronous Parallel Programming using the BSMLlib Library. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice, pages 165–178. Nova Science Publishers, 2002.

Erik A. Hendriks. BProc: Beowulf distributed process space. WWW, 2002. URL `http://bproc.sourceforge.net/`.

Konrad Hinsen. High-level scientific programming in python. In P. M.A. Sloot, C.J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002*, volume 3 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

Patricia D. Hough, Tamara G. Kolda, and Virginia J. Torczon. Asynchronous parallel pattern search for nonlinear optimization. *SIAM J. Sci. Comput.*, 23:134–156, 2001.

Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5:299–314, 1996.

Pierre L'Ecuyer, Richard Simard, E. Jack Chen, and W. David Kelton. An object-oriented random-number package with many long streams and substreams. *OR*, 50:1073–1075, 2002.

Na Li. rsprng. WWW, 2002. URL `http://cran.r-project.org/`.

Na Li and A.J. Rossini. RPVM: Cluster statistical computing in R. *R News*, 1(3):4–7, September 2001. URL `http://CRAN.R-project.org/doc/Rnews/`.

Na Li and A.J. Rossini. rpvm. WWW, 2002. URL http://cran.r-project.org/.

Andy Liaw. randomForest. WWW, 2002. URL http://cran.r-project.org/.

Michael Mascagni and Ashok Srinivasan. Sprng. WWW, 2001. URL http://sprng.cs.fsu.edu.

Xiao-Li Meng and Wing Hung Wong. Simulating ratios of normalizing constants via a simple identity: A theoretical exploration. *Stat. Sinica*, 6:831–860, 1996.

A. Merlin, G. Hains, and F. Loulergue. A SPMD Environment Machine for Functional BSP Programs. In *Proceedings of the Third Scottish Functional Programming Workshop*, 2001.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard (version 1.1).*, 1995. URL http://www.mpi-forum.org. Technical Report, http://www.mpi-forum.org.

Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface.*, 1997. URL http://www.mpi-forum.org. Technical Report, http://www.mpi-forum.org.

P. L. Miller, P. Nadkarni, J. E. Gelernter, N. Carriero, A. J. Pakstis, and K. K. Kidd. Parallelizing genetic linkage analysis: a case study for applying parallel computation in molecular biology. *Comput. Biomed. Res.*, 24:234–248, 1991.

R Development Core Team. 1.6.2. WWW, October 2002.

M. J. Schervish. Applications of parallel computation to statistical inference. *J. Amer. Statist. Assoc.*, 83: 976–983, 1988.

SciPy. Scientific python. WWW, 2003. URL http://www.scipy.org/.

W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S.* Springer, fourth edition edition, 2002.

Wolfram Research. gridMathematica. WWW, 2003. URL http://www.wolfram.com/products/gridmathematica/.

Hao Yu. Rmpi. WWW, 2002a. URL http://cran.r-project.org/.

Hao Yu. Rmpi: Parallel statistical computing in R. *R News*, 2(2):10–14, June 2002b. URL http://CRAN.R-project.org/doc/Rnews/.

A Configuration Issues

In addition to installing the snow package and optional support packages, some one-time configuration is needed. This appendix outlines the issues that need to be addressed; details are given in the package documentation.

A.1 Choosing a Communication Layer

The snow package can use one of three communication layers: sockets, PVM or MPI. The sockets interface is the most basic. It does not require any additional software. On the other hand, it does not provide tools for monitoring performance and for cleaning up stray processes in the event of failures. Some of the more sophisticated features of snow may not be supported under sockets. For example, at the time of writing the load balancing function clusterApplyLB is not available under sockets since R does not yet include a facility for waiting for the first of several sockets to become available for reading.

The PVM layer requires that the rpvm package and the PVM library be installed. (PVM stands for Parallel Virtual Machine.) This library can be installed at the system level or by individual users. PVM provides several tools for examining and controlling the state of the virtual machine, including the graphical interface xpvm, which was used to produce the graphical visualizations in Figure 2.

The MPI layer requires the Rmpi package and an MPI system supported by Rmpi. At present only LAM-MPI is directly supported. It would be easy to modify Rmpi to support other MPI systems that include the MPI 2.0 process spawning facility. MPI systems that do not provide process spawning require a different approach to creating a cluster. snow support for such systems is under development. Like PVM, LAM-MPI can be installed at the system level or by individual users.

A.2 Configuring the Communication Layer

The communication layer is responsible for starting any additional processes that are needed, possibly on remote machines. To do this they require information about the machines to use for slave processes. Both PVM and LAM-MPI can be started with configuration files that specify the machines to use and, optionally, the user name on the machine. A separate mechanism is provided for the socket layer.

Once the machines to use have been identified, a remote process execution mechanism is needed to start the required processes. In a secure environment, such as the internal network in a dedicated cluster, the remote shell facility rcmd can be used; it would typically be configured to not require authentication.

In a less secure environment, such as using workstations within an organization or perhaps across organizations, the secure shell `ssh` can be used. PVM and LAM-MPI use environment variables to specify the remote shell mechanism to use. The socket layer allows the remote shell command to be specified within R.

The commands used to start the R slave processes need to be able to locate the R program, the `snow` package and supporting packages, and a shell script within the `snow` package that is used to run the slave loop that starts up communication with the master and waits for commands. In a homogeneous environment in which the machines used to run the master and slave processes have the same architecture and file system layout the required information can be determined on the master. In a heterogeneous environment with multiple architectures and different file system layouts the current approach is to require, on each machine used to run slave processes, the placement of a small shell script in the execution path and the definition of an environment variable that identifies the location of the directory containing the `snow` package and supporting packages. Other approaches are being explored.

A.3 Initializing Compute Clusters

When using the PVM or MPI message passing toolkits, the compute clusters should be initialized before loading `snow` into R. For reasons of security, we will use `ssh` for initializing the parallel toolkits, and assume that it has been configured and works for the user. We assume that PVM (with `rpvm`) or LAM-MPI (with `Rmpi`) has been installed and verified by the systems administrator, and that SPRNG, `rsprng`, and `snow` have also been installed. Construct a file containing the names of each machine, one per line. For PVM, this file is traditionally called `pvmhosts`, and for LAM-MPI, this is called `LAMhosts`. Construction of the PVM-enabled cluster can be done by:

```
env PVM_RSH=ssh pvm pvmhosts
```

and for an MPI-enabled cluster,

```
env LAMRSH="ssh -x" lamboot -v LAMhosts
```

At this point, R can be started, `snow` can be loaded, and the examples should work.

B Related Work

B.1 Statistical Applications

This section provides some examples of the use parallel computing in statistics. It is intended to be illustrative rather than exhaustive.

In pedigree analysis, computing the likelihood for a given set of parameters can be very computationally intensive. Since the likelihoods for different sets of parameters and for different pedigrees are independent, it is very natural to distribute their computation to several processors (Miller et al., 1991). One popular package, FASTLINK, takes this further by carefully partitioning the missing data space to allow parallel computation for even a single pedigree and set of parameters (Dwarkadas et al., 1994; Gupta et al., 1995). FASTLINK implements this approach using a distributed shared memory system.

Pattern search is a class of direct search methods for solving unconstrained nonlinear optimization problems. Since it does not require the calculation of derivatives it is very popular when the evaluation of the objective function is very expensive. This approach works by sampling the objective function over a predefined pattern of points. Once those points are identified the corresponding function values can be computed independently and hence concurrently (Hough et al., 2001).

Importance sampling is a Monte Carlo integration technique that is useful when the target distribution is difficult to simulate from, for example when there is an unknown normalizing constant. Instead, we sample from another known distribution and weight the samples by their importance sample weights. Importance sampling is very inefficient when sampling and target distributions are far apart. Bridge sampling (Meng and Wong, 1996) is one extension of this technique that uses multiple sampling distributions and can thus be parallelized.

Markov chain Monte Carlo methods have been very widely studied over the last decade. Basic MCMC is inherently sequential, but there are still opportunities for parallelization. In high-dimensional problems, single updates may be sufficiently complex to benefit from parallelization. It can also be useful to run several parallel chains, either for diagnostic purposes or as part of a larger algorithm. Simulated tempering (Geyer and Thompson, 1995) draws realizations from a sequence of distributions which may change randomly over time. Only one of the distributions may be of interest, but the use of multiple chains at different "temperatures" can significantly improve mixing. Originally developed as a sequential algorithm, this approach can be adapted to run in parallel, where it is known as parallel tempering. Methods that combine MCMC with importance weighting, such as particle filters, also have components that can be run in parallel.

B.2 Parallel Computational Frameworks

Interfaces to MPI and PVM analogous to the R packages `Rmpi` and `rpvm` have been developed for several high level languages. Examples include `MatlabMPI` for Matlab, `Parallel::MPI` for Perl, and `Pypvm` and `Parpy` for Python. Some language and runtime systems provide other communication layers that can be used for distributed computing. Java provides the remote method invocation interface; Modula 3 provides network objects. Modula 3 network objects were used as the basis for the distributed language Obliq (Cardelli, 1995).

Simple higher level parallel toolkits include COW (Cluster of Workstations) from Scientific Python (SciPy) and the Mathematica Parallel Computing Toolkit, recently renamed gridMathematica (Wolfram Research, 2003). Parts of the design of `snow` were based on the design of Python COW.

More advanced parallel computing frameworks based on the Bulk Synchronous Processing (BSP) model for parallel computation have also been developed. A version for Python is described in Hinsen (2002). The design of the Python approach is based on the design of a BSP programming framework for ML (Merlin et al., 2001; Hains and Loulergue, 2002)

While message passing remains the most commonly used approach to parallel computing on networks of workstations, approaches based on manipulation of distributed shared memory rather than explicit message passing are also possible. These include the tuple space approach of LINDA as well as approaches that create a common shared address space.

Supporting facilities that can facilitate the management of distributed computing include operating system support for transparent process migration (Barak and La'adan, 1998; Bar, 2002) and systems such as `BProc` for presenting a single shared process ID space (Hendriks, 2002). Both of these are kernel-level distributed computing tools, and are orthogonal in effect to user supplied parallelized programs. Cluster management is a related problem. Tools such as Sun Microsystems' Sun Grid Engine (SGE, Foster and Kesselman, 1998) and Condor (Basney and Livny, 1999) provide explicit management of processes across a cluster, assisting with load balancing. These tools make highly optimized load-balancing less of an issue.

List of Figures

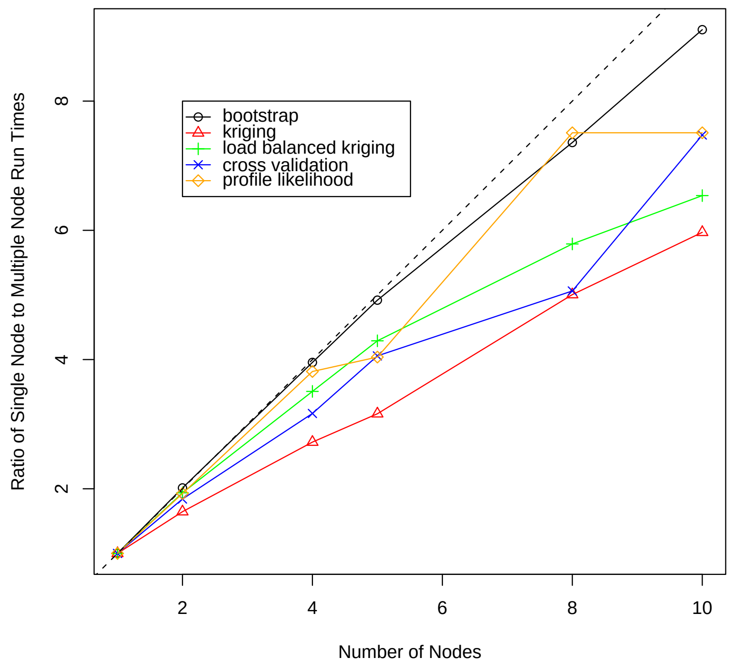
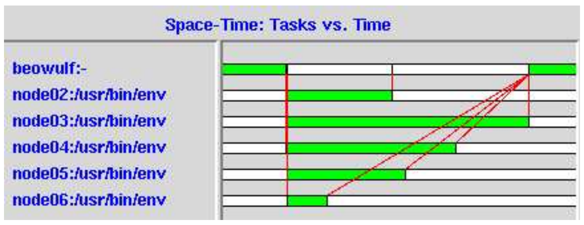
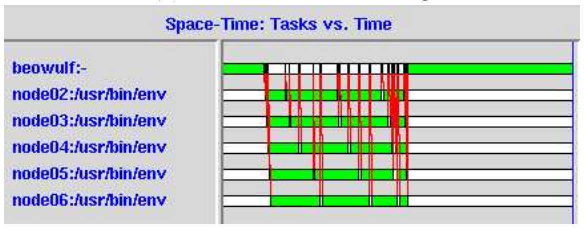


Figure 1: Speed-ups (ratio of single node to multiple node execution times) for all of the examples on clusters of several sizes. The dashed line is the 45 degree line.



(a) without load balancing



(b) with load balancing

Figure 2: Cluster utilization for the kriging example on 5 compute nodes with and without load balancing. The top node in each graph is the master. Green represents active computation; red lines represent communications. The time-scales are not the same; load balancing is faster but not 2-fold.

List of Tables

High Level Routines	
`parLapply`	parallel `lapply`
`parSapply`	parallel `sapply`
`parApply`	parallel `apply`
Basic Routines	
`clusterExport`	export variables to nodes
`clusterCall`	call function on each node
`clusterApply`	apply function to arguments on nodes
`clusterApplyLB`	load balanced `clusterApply`
`clusterEvalQ`	evaluate explicit expression on nodes
`clusterSplit`	split vector into pieces for nodes
Administrative Routines	
`makeCluster`	create a new cluster of nodes
`stopCluster`	shut down a cluster
`clusterSetupSPRNG`	initialize random number streams

Table 1: Major `snow` API functions.