Pragmatic parallel computing

Hana Ševčíková and A. J. Rossini
University of Washington
{hanas, rossini}@u.washington.edu

1 Introduction

Two of the biggest practical problems facing parallel statistical computing are random number generation and fault tolerance. This paper describes some recent work from our research group which addresses these issues. This research has resulted in the packages **rlecuyer**, **snowFT**, and in contributions to the **snow** package.

The introduction of **rpvm**, **Rmpi**, **snow** and related R packages has lowered the bar for implementing parallel algorithms in R. **rpvm** (Li and Rossini 2001a;b) was among the first to enable parallel R applications by providing an interface and infrastructure for using the PVM message-passing library for pure R code (Geist et al. 1994). **Rmpi** was publicly released shortly after and it provided an interface to MPI (Dongarra et al. 1994, Yu 2002). Parallel random number generation was addressed with **rsprng**, an R package which provides an interface to the distributed random number generator SPRNG (Srinivasan and Mascagni 2000, Li 2002). This simplified the implementation of stochastic parallel applications and algorithms as exemplified by numerical studies and bootstrapping (see e.g. Carson et al. (2003)).

snow (Tierney 2002, Rossini et al. 2003) provides a generic high level interface to message-passing libraries (PVM, MPI, raw socket communication), making parallel programming in R communication-layer independent. Although **snow** is a simple and powerful tool for developing cluster applications, its initial focus on simplicity avoided several important pragmatic issues including fault tolerance and guaranteed reproducibility.

For us, parallel statistical computing is about getting appropriate final results in a timely manner. Part of the "timely manner" is to make parallel computing easily accessible; **snow** lets one start computing in parallel in a general manner, but does not necessary ensure that the computational results will successfully complete or be reproducible. The power of a computational cluster is proportional to the number of compute nodes (and corresponding network interconnections) which form the system. To keep costs low, they are often constructed using commodity rather than specialized hardware. Since they are far more complex than a single computer, they suffer from the increased probability of hardware failure, either at the level of the individual compute-nodes or in the interconnecting network system. In addition, as simulations become larger in scale, the corresponding statistical models more complex, and the inferential computations of point and interval estimators more challenging, the likelihood increases that pathological inputs which are ill-handled by the program code will be generated.

This article describes some of our research into reproducibility for parallel computing. Practical real-world requirements for parallel applications include fault tolerance, computational reproducibility, dynamic adjustment of the cluster configuration, and computational transparency (Ševčíková 2003). We now proceed to describe the basic usage of **snow** and the practical issues which led to our extensions. Next, we describe the features of **snowFT** and explain their usage. Finally, we conclude with a practical example illustrating the investigation of a fractal dimension estimation procedure.

2 Basic Usage

snow facilitates the evaluation of a function `myfun(x,par)` that must be executed r times, with the results collated and summarized. Assume that p processors are available. Suppose also, each call of `myfun` requires an independent stream of random numbers. To keep the example from being trivial, let `myfun` require a specific R package **mylib** and the evaluation of an initialization function `myinitfun` prior to computation.

Then, one can proceed as follows:

```
cl <- makeCluster(p)
clusterEvalQ(cl, library(mylib))
clusterCall(cl, myinitfun)
clusterSetupSPRNG(cl)
results <- clusterApplyLB(cl,par1[1:r],myfun,par=par2)
stopCluster(cl)
evaluate(results)
```

Intuitively, the first expression creates and returns a link to the virtual cluster, creating a cluster of p slave processes. We will occassionally refer to these "slave processes" as *nodes* or *computational nodes*. The second expression loads **mylib** on each slave. Then, `myinitfun` is called on each slave. `clusterSetupSPRNG` initializes one SPRNG stream for each slave node. The function `clusterApplyLB` calls `myfun` on first slave with arguments (`par1[1], par2`), on second slave with (`par1[2],par2`) and so on up to the pth slave. After one of the p slaves returns a result, `myfun` is called on that slave with (`par1[p+1],par2`) etc. This is repeated until `myfun` is called r times and the routine waits until all r results are received. `stopCluster` cleans up any remaining connections and shuts down the cluster. The desired statistics would be then computed in the last user-defined function.

3 Problems

The described scenario uses a simple algorithm for `clusterApplyLB` which can be simply described (assuming that $p \leq r$) by the following pseudo-code:

```
start myfun on p slaves
for (i in 1:r) {
    wait for a result
    if (i+p <= r) start myfun on the free slave
    save result
}
```

This load-balances the process. Unlike the alternative **snow** function, `clusterApply`, the faster slaves process more computational tasks than slower slaves. Note that the scenario does not require synchronizing computation between slaves. The results are potentially non-reproducible since each stream of random numbers is allocated to a particular slave, but the assignment of replicates to slaves is non-deterministic. Further, any hardware or software failure of a slave will result in an infinite loop while the main process waits for completion of the subprocess. This is true for any other **snow** function that receives results from slaves, since during program execution, there currently is no updating of the status of the computation.

Finally, the size p of the cluster is fixed by initial specification and there is no way to change it later. As a starting point to explore solutions to these issues, we constructed **snowFT** with a function `clusterApplyFT`.

4 Random Numbers

Two practical problems for parallel stochastic simulation are the appropriate use of distributed random number generators, and more importantly, the exact reproduction of results. Evaluating the quality of univariate random number streams is quite difficult, and more critically to our situation, extremely difficult for parallel streams. As part of our initial research into the quality of current implementations of parallel streams, we constructed an R package, **rlecuyer** (Ševčíková and Rossini 2004), which provides an R interface to RNGstream (L'Ecuyer et al. 2002). We also contributed modifications to **snow** to make this new parallel random generator transparently accessible, as an alternative to SPRNG.

Exact reproduction of stochastic computational results in the parallel setting requires strong control over the allocation of random number streams to slave processes. This was originally done in **snow** by considering 'one stream per node'. Guaranteed reproducibility across heterogeneous and adaptive settings requires a 'one stream per replicate or subprocess' paradigm. With this change, starting from a certain seed, the ith generated stream is assigned to ith replicate. This is implemented for RNGstream by creating an identical stream table on each node and making corresponding assignments prior to each computation call using `.lec.CurrentStream`. For SPRNG, `initSprngNode(i,r-1,...)` is called prior to the computation of ith replicate. Thus, the replicates produce the same results regardless of the node on which they are computed.

5 Features of snowFT

The master node's primary task is to distribute pieces of the computation and collect/collate the final results. Assuming a computation that is much more time consuming than the overhead caused by managing the slaves, the master spends most of the time waiting for the slave nodes to finish. Hence, the master can spend a portion of the waiting time to check the existence of its slave nodes and similar additional administration, without significantly affecting the execution time. This is accomplished by replacing the original blocking receive command, *wait for a result*, with the following scheme:

```
look for a result
if (no result arrived) {
    loop {
        failure detection
        if (failure) {recovery; break}
        administration
        if (p increased) {add nodes; break}
        wait t time units for a result
        if (result arrived) break
    }
}
```

The first line is implemented by a non-blocking receive and the last receive call in the loop is implemented by a time-out receive function. Hence, the management loop is entered every t time units until a result is obtained. The loop exits either when some results arrive, a failure is recovered (see next section) or when the size of the cluster has been increased. The two latter cases imply that free nodes are available for starting new computations. However, since no results are returned, the results storing procedure is omitted.

5.1 Failure Detection and Recovery

Failure detection uses the **rpvm** function `.PVM.pstats`; this returns process status and allows us to easily identify failed nodes.

If a node fails, the recovery procedure replaces it by a new created node. Then any calls and initializations made on the node prior to the computation have to be made. Going back to the introductory example, these would be evaluating the `library` expression, calling `myinitfun` and initializing the random number generator (RNG). For the usage in `clusterApplyFT` the user defines a function containing all calls that are made prior to the computation and passes it to `clusterApplyFT` as argument.

Since failure detection can be done faster than message transfer, results may arrive after the generating computational node has been detected as failed and the appropriate recovery has been made. Therefore in order to identify the arrived result, it is useful to keep a list of the failed nodes.

An alternative way of implementing failure detection is using the function `.PVM.notify`. In this case, the master node is automatically notified by the pvm daemon upon process failure. However, this notification happens after an unspecified delay. In contrast, our solution gives the master full control over the timing of failure detection.

Reproducibility is ensured by repetitions of failed computations. Our implementation runs the main computation loop of r replicates up to three times. If there are missing results in the first run, then a second run is performed on the corresponding failed replicates. The third run is performed for any second-run failures. If there are any failures in the last run, the user is notified that reproducibility can not be guaranteed. The use of three runs as sufficient is subjective and can be easily modified by the user.

5.2 Administration

The master process writes the state of the simulation along with any failed replicates to a file to assist with the clarifying the behavior of the process. By default, the replicates being currently proceeded are written to a file '.proc' and the failed replicates to a file '.proc_fail'. Since the administration procedure is performed only during the master's waiting time, updating '.proc' may be delayed, depending on replicate computational times.

5.2.1 Dynamic Cluster Resizing

Dynamically changing the number of involved processors is useful for optimal adaptation to current system load conditions; total run times are a function of both CPU and network load, and addition and completion of other processes can affect both parameters. This is enabled in **snowFT** by providing a management file (by default '.clustersize'). At the beginning of the computation the master stores here the initial size of the cluster. This can be changed by the user any time. The master reads the file within the administration procedure. If the size is to be increased, new nodes are created and the appropriate initialization of the nodes (mentioned in the recovery paragraph) are performed. If the size is to be decreased, nodes are successively discarded after they finish their current computation.

Extensions, such as automatic modification of '.clustersize' by a load monitoring procedure, or a dynamic addition/deletion of nodes by an extraneous process are potentially a topic of future research.

5.3 Intermediate Results

We have added the capacity to call a predefined function which is evaluated after a certain number of finished replicates. This function is called with the list of results and the number of finished results as arguments, where the non-finished results are set to NULL. A list of other arguments can be passed to the print function as well. This feature is useful especially for computing and printing intermediate results.

6 Fault-tolerant Load-balancing

The function `clusterApplyFT` requires more information in its arguments than `clusterApplyLB`. This is due to the recovery procedure, dynamic node addition, and improved random number stream handling. Since the composition of the cluster may change during the function call, the current cluster is returned together with the computation results. We simplified this using a wrapper function, called `performParallel`, that creates a cluster of a given size, performs the initialization of the nodes including the RNG, calls `clusterApplyFT`, performs aftermath functions if needed, stops the cluster and returns the computation results.

Consider the example in Section 2 and suppose additionally we have a function `myprintfun` for computing intermediate results that we wish to run after each 10 replicates. We also wish to use the RNGstream generator with default seed. Then, we could proceed as follows:

```
initfun <- function() {
  library(mylib)
  myinitfun()
}

performParallel(p, par1[1:r], myfun, initfun=initfun, printfun=myprintfun,
  gentype="RNGstream", printrepl=10, par=par2)
```

The optional `exitfun` argument can be used for any concluding operations. The argument `mngtfiles` is used to set the default management file name mentioned in the administration section. Also, by setting `ft_verbose=TRUE` the function prints out messages about the execution status.

6.1 Limitations

Although our design guarantees detection and recovery of any computational failure, fault tolerance, while improved, is still imperfect. Functions passed in the arguments `initfun` and `exitfun` are performed on nodes using `clusterCall`. Since this **snow** function uses blocking receive and thus is not a fault tolerant function, any failure that happens during the execution of `initfun` and `exitfun` would not be detected and would result in an infinite waiting of the master node. Moreover, if a failure occurs on the master node, the computation fails and the slave nodes should be shut up manually. Our choice of the current design is a compromise in the tradeoff between the benefit one would gain by solving these problems and the resulting program complexity.

The **snow** package is built on PVM, MPI, or a local socket interface. Unfortunately, MPI in its current specification (MPI standard 2.0) does not provide tools for implementing a simple fault tolerance algorithm. Changes which will facilitate this are in progress (see e.g. Fagg et al. (2003)) but at the time of writing this article none of these has been included in the common MPI implementations. Our fault tolerance implementation in **snowFT** thus requires PVM.

6.2 Example

We now demonstrate our extensions with a practical example. We wish to analyze statistical properties of an estimation procedure of fractal dimension. The estimate is computed on a two dimensional random field of a size $n \times n$ with a known fractal dimension D. Repeating the estimation r times where r is sufficiently large should provide a sufficient statistics about the estimator. For large n and large r, this can be a very time consuming task. Hence, parallelization can help to solve it in a reasonable time.

Our estimation procedure `Variogram(x)` takes a two dimensional random field x as an argument and returns an estimator of fractal dimension as a single value. For the simulation of random fields we use the function `GaussRF` of the **RandomFields** package.

Our main computation function is defined as:

```
EstimateFD <- function(n, rfp, model, method) {
  grid <- c(0,(n-1)/n,1/n)
  rf <- GaussRF(x=grid,y=grid,grid=TRUE,n=1,model=model,param=rfp,
    gridtriple=TRUE,method=method)
  return(Variogram(rf))
}
```

The init function is simply:

```
initfct <- function() library(RandomFields)
```

The main program can be implemented as follows:

```
n<-256; r<-10000 # field size, replicates
p<-5 # cluster size
output <- 'FD.out'; D<-2.5
model<-"2dfractalB"; method <- "local CE"
rfp <- c(mean=0,variance=1,nugget=0,scale=1,kappa=6-2*D)
write(paste(' r', ' mean',' MSE', ' RMSE', 'variance', ' bias',
 sep=" "), file=output)
```

```
res <- performParallel(p,rep(n,r),EstimateFD,initfun=initfct,printfun=printfct,
 printargs=list(fd=D,file=output),printrepl=500,cltype="PVM",rfp = rfp,
 model=model,method=method)
write("Results:",file=output, append=TRUE)
evaluateresult(res,D,output)
```

The functions for intermediate results and for evaluating results are defined as:

```
printfct <- function(res,n,args) {
 evaluateresult(res,args$fd,args$file)
}

evaluateresult <- function(res, fd, file) {
 stat <- statistics(unlist(res),fd)
 write.table(data.frame(stat), file=file, append=TRUE, col.names=FALSE,
   row.names=FALSE, quote=FALSE)
}

statistics <- function(res, fd) {
 res <- res[!is.null(res)]
 mean<-mean(res); mse <- mean((res-fd)^2)
 return(list(n=length(res), mean=round(mean,4), mse=round(mse,6),
   rmse=round(sqrt(mse),6), var=round(var(res),6), bias=round(mean-fd,6)))
}
```

7 Discussion

Computational clusters are rapidly becoming widely available in both commercial and academic settings as researchers look for inexpensive means to increase the efficiency and throughput of statistical research and collaborative activities. We have described our activities studying the real-world use of computational clusters for statistical research. The results of our theoretical work have been implemented in the **snowFT** and **rlecuyer** R packages which augment **snow**. These packages provide robustness and increased confidence in the reproducibility of the results from computationally intensive procedures evaluated on computational clusters.

Acknowledgement

We thank Michael (Na) Li and Luke Tierney for helpful comments which have improved this article.

References

Carson, B., Murison, R., Mason, I., *Computational Gains Using RPVM on a Beowulf Cluster*, R News **3** (2003)(1), 21–26.

Dongarra, J., Hempel, R., Hey, A., Walker, D., *MPI: A Message-Passing Interface Standard*, International Journal of Supercomputer Applications **8** (1994), 159–416.

Fagg, G., Gabriel, E., Chen, Z., Angskun, T., Bosilca, G., Bukovsky, A., Dongarra, J., *Fault Tolerant Communication Library and Applications for High Performance Computing*, Los Alamos Computer Science Institute Symposium (2003), to appear.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V., *PVM: Parallel Virtual Machine*, The MIT Press (1994).

L'Ecuyer, P., Simard, R., Chen, E. J., Kelton, W. D., *An Object-Oriented Random-Number Package With Many Long Streams and Substreams*, Operations Research **50** (2002)(6).

Li, M. N., *rsprng*, R package (2002), http://cran.r-project.org.

Li, M. N., Rossini, A. J., *RPVM: Cluster statistical computing in R*, R News **1** (2001a)(3), 4–7.

Li, M. N., Rossini, A. J., *RPVM: Cluster Statistical Computing in R*, Technical report at University of Washington (2001b), http://www.analytics.washington.edu/statcomp/projects/rhpc/rpvm/.

Rossini, A. J., Tierney, L., Li, M. N., *Simple Parallel Statistical Computing in R*, UW Biostatistics Working Paper Series (2003)(193).

Srinivasan, A., Mascagni, M., *SPRNG: A Scalable Library for Pseudorandom Number Generation*, ACM Transactions and Mathematical Software **26** (2000), 436–461.

Tierney, L., *Simple Network of Workstations for R*, Technical report, School of Statistics, University of Minnesota (2002), http://www.stat.uiowa.edu/luke/R/cluster/cluster.html.

Ševčíková, H., *Statistical Simulations on Parallel Computers*, Journal of Computational and Graphical Statistics (2004), to appear.

Ševčíková, H., Rossini, A., *rlecuyer*, R package (2004), http://cran.r-project.org.

Yu, H., *Rmpi: Parallel statistical computing in R*, R News **2** (2002)(2), 10–14.