JOIM
www.joim.com

JOURNAL OF INVESTMENT MANAGEMENT, Vol. 7, No. 4, (2009), pp. 1–12

SURVEYS AND CROSSOVERS

FINANCIAL APPLICATIONS WITH PARALLEL R

Sanjiv R. Das[a,*] *and Brian Granger*[b]

The use of statistical packages in finance has two functions. One, econometric analysis of large volumes of data, and two, programming financial models. A popular package for these purposes is R. In this article we will examine two canonical applications of parallel programming for option pricing. We use the ParallelR package developed by REvolution Computing. We price options using trees and Monte Carlo simulation. Both these approaches are commonly used for option pricing and are amenable to parallelization and grid computing. In this paper we demonstrate the application using the widely used mathematical/statistical R package.

1 Introduction

R is an open-source programming language for statistics and econometrics. R evolved at Bell Labs and is similar to the S language that is also widely known and used in statistical applications. But it is more than a mere statistical language. It is a full-blown programming environment for numerical and scientific computing. In this article we demonstrate the use of a parallel version of R for option pricing applications.

*Corresponding author. Finance Department, Leavey School of Business, Santa Clara University, Santa Clara, CA 95053, USA.

[a]Leavey School of Business, Santa Clara University, Santa Clara, CA 95053, USA. E-mail: srdas@scu.edu

[b]Physics Department, California Polytechnic State University, San Luis Obispo, CA 93407, USA. E-mail: bgranger@calpoly.edu

R is already widely used in academic research. There are many financial packages that have been developed and are becoming increasingly popular. The `Rmetrics` suite of packages comprises `fArma` for autoregressive time series modeling, `fOptions` for pricing options, `fBonds` for fixed-income models, `fCopulae` for credit risk modeling, etc. There are several other packages provided by `Rmetrics` and these are being extended to additional models on an on-going basis.[1]

R is an open source project and is supported on all hardware platforms. For readers unfamiliar with R, the software, tutorials and full documentation are available at the project website on Source Forge (`http://www.r-project.org/`). R is an excellent platform for scientific computing because it combines many features. First, it is an industrial strength platform for statistics and

econometrics, and contains all the required tools, applicable in a canned manner. Therefore, the user can easily run many statistical analyses with just a few commands. In this mode, R offers all the tools that come with pure statistical packages such as SAS, SPSS, and Stata. In finance, for example, one may run a GARCH analysis with only a few lines of program code. Second, R is interactive. Analyses may be run from the command line with immediate response. Therefore, while R is a full-blown programming language, it also supports interactive computation making it suitable for a wide range of user styles. Third, R is a matrix or array-based language, and therefore subsumes the functionality of mathematical packages directed at scientific computing, such as Matlab and Gauss. Fourth, R comes with several libraries and toolboxes that may be put to specialized purpose. As of March 2009, the R repository, known as CRAN (Comprehensive R Archive Network) contained over 1700 packages. All software in CRAN is free. Hence, the adoption rate for R has accelerated, enhancing the network benefits of a growing scientific community.

There are several other R programming packages for finance. The `RQuantLib` package supports option-pricing and fixed-income functions. The `quantmod` package is (quite obviously) for quantitative modeling in finance. One could go on and on, but the list would not be exhaustive, and it's best to refer to internet sources that keep these packages up to date and have current listings.[2]

Despite this proliferation of R tools for finance, there are very few free tools for financial applications that exploit the newest hardware, i.e. multicore processor machines, GPUs and computing clusters. The paucity of parallel programming solutions using R is symptomatic of these early days of parallel programming packages, as is the absence of documentation that enables ordinary application programmers to get up and running in short order. This article seeks to provide a gentle introduction to the basics of a newly developed solution to parallel computing in R, namely the `ParallelR` package. This is developed by REvolution Computing, a company that styles itself as the "Red Hat of R computing".[3] The `ParallelR` package is very easy to install and use. This package needs to be used in conjunction with the Network Spaces (NWS) Server. Network Spaces is a system that is widely used to allow different scripts and programs to communicate with each other. These programs may reside on the same machine or on multiple machines. Appendix A details how to install these tools (NWS and `ParallelR`), and Appendix B explains how to deploy them.

Parallel implementations of programs are best elucidated with examples. The rest of the article is directed towards finance academics who may be interested in getting up and running with the system with a short learning curve.

2 Parallel Monte Carlo

The simplest examples of parallel computing are cases where an algorithm can be broken down into distinct subprocesses, each of which operates completely separately. What this means is that there is no interprocess communication nor is there any shared data. These problems are known as "embarrasingly parallel" and the procedure only involves a final aggregation step after all subprocesses have returned the results of their computations.

The pricing of European options is an exemplar of such problems and offers a good starting point for demonstrating the use of `ParallelR`. We price European calls in the Black and Scholes (1973) and Merton (1973) models. In this model the risk-neutral stock price evolution is given by

$$S(t+h) = S(t)\exp\left[\left(r - \frac{1}{2}\sigma^2\right)h + \sigma\epsilon(t)\sqrt{h}\right], \quad \epsilon(t) \sim_{iid} N(0,1) \qquad (1)$$

where $S(t)$ is the stock price at time t, h is a small time interval (measured in fractions of years), r is the risk free rate of interest, and σ is the volatility of the stock. The initial stock price is $S(0)$. The standard normal distribution is denoted $N(0,1)$ above. Equation (1) represents the price path of the stock $S(t)$ as driven by repeated draws of the random variable $\epsilon(t)$.

The Monte Carlo simulation proceeds by generating m paths of stock prices, where each path is denoted by $S_j(t), t = 1 \ldots T$ for the j-th path. Each path is assumed to have n steps, such that an option of maturity T will have a time-step of $h = T/n$. The terminal stock price on each path is denoted $S_j(T)$. The final call price will be given by

$$\text{Call Price} = \exp(-rT)\frac{1}{m}\sum_{j=1}^{m}\max[0, S_j(T) - K] \quad (2)$$

Since this involves a simple sum over the outcome of m paths, the work of each path can be farmed out to different "workers", i.e. processor cores on the same machine or CPUs on different computers. This simple divide-and-conquer approach is a hallmark of embarrassingly parallel problems. In this article we focus only on parallelism across multiple cores on the same computer, and leave parallelism on a computing grid for future exposition.

The program code for this model is presented in Figure 1.

The program contains a pricing function `mcoption` (lines 2–14) that computes the call price with no parallelization. (We have eschewed vectorization of the program in line 9 because we want to emphasize the parallelization features). In order to run this function across multiple cores of a dual-core machine, we invoke function `mcparallel` (lines 16–19) which breaks down the problem into equal parts and sends each part to a separate core or

```
1 #Parallel R Monte Carlo
2 mcoption = function(s0,k,t,v,r,n,m) {
3   h = t/n
4   s = matrix(0,m,n+1)
5   payoff = matrix(0,m,1)
6   for (j in 1:m) {
7       s[j,1] = s0
8       for (i in 2:(n+1)) {
9           s[j,i] = s[j,i-1]*exp((r-0.5*v^2)*h + rnorm(1)*v*sqrt(h))
10      }
11      payoff[j] = max(0,s[j,n+1]-k)
12  }
13  callprice = mean(payoff)*exp(-r*t)
14 }
15
16 mcparallel = function(s0,k,t,v,r,n,m,numw) {
17  sol = eachWorker(sl,mcoption,s0,k,t,v,r,n,m/numw)
18  print(sol)
19 }
20
21 ##### MAIN SEGMENT #####
22 library(nws)
23 numw = 2
24 sl = sleigh(workerCount=numw)
25 syst = system.time(mcparallel(100,100,1,0.3,0.03,52,100000,numw))
26 print(syst)
27 close(sl)
```

Figure 1 Program code for Monte Carlo call option pricing with `ParallelR`.

"worker". The function `mcparallel` calls the function `mcoption`. (The first two letters "mc" in these functions stands for Monte Carlo).

Line 22 of the program initializes the Network Spaces (NWS) library (we assume that the NWS server has already been activated as shown in Appendix B). The number of worker cores is defined in line 23. Line 24 initializes the "sleigh" that brings all cores in the CPU into operation. Line 25 calls the function `mcparallel` and also starts the system timer to analyze the run time performance of the program. Lines 26–27 print results and close the sleigh that shackles the cores together in parallel.

Line 17 of the program that contains the critical call to each worker core of the CPU, sending out simulation path requests. Each core is assigned an equal share of the total number of paths m.

Starting up and running ParallelR

Before running the R program, we need to make sure that the NWS server is up and running. To do this, invoke from the command line:

```
/opt/REvolution/ParallelR/bin/
nwsserver start
```

To check that the server is running, open a browser and go to the following URL:
`http://localhost:8766`

In order to assess progressive performance, we run the Monte Carlo simulation for $m = 100,000$ paths and $n = 52$ steps over a one-year horizon in three different ways:

(1) We run the model *without* using the parallelR package. Hence, we make a direct call to the `mcoption` function and the results are as follows.

```
> system.time(mcoption(100,100,1,
  0.3,0.03,52,100000))
   user  system elapsed
145.605   0.855 145.286
```

The elapsed time is 145 seconds, i.e. over 2 minutes.

(2) Next, we run the model using parallelR but use only one worker (core) of the CPU. The timing results are as follows.

```
> library(nws)
> numw = 1
> sl = sleigh(workerCount=numw)
> syst = system.time(mcparallel
  (100,100,1,0.3,0.03,52,100000,
  numw))
[[1]]
[1] 13.3047

> print(syst)
   user  system elapsed
  1.065   0.246 145.767
> close(sl)
```

We see that the elapsed time is essentially the same as before when we did not use the parallel environment.

(3) Finally, we run the model with two workers in a parallel environment. The results are:

```
> numw = 2
> sl = sleigh(workerCount=numw)
> syst = system.time(mcparallel
  (100,100,1,0.3,0.03,52,100000,
  numw))
[[1]]
[1] 13.20356

[[2]]
[1] 13.17537

> print(syst)
   user  system elapsed
  0.600   0.142  78.577
> close(sl)
```

We see that the elapsed time (78 seconds) is now half of that taken when only one worker was invoked for the task. The final price will be the average of the prices generated by the two workers. We note here that the goal of this exercise is to demonstrate the syntax for parallelizing Monte Carlo programs in R, and is by no means a fast approach for option pricing. The example here shows how the use of both cores of the processor halves the time taken. Were the serial version of the model to run much faster, then it is also likely that the overhead required in managing the two cores might be a significant portion of the run time, and the gains from parallelization would be less striking. Nonetheless, even with these caveats, the ease with which parallelized Monte Carlo models may be implemented is evident from the simple syntax involved.

3 Option pricing on trees

In this section we provide an example of how parallelism may be applied to call option pricing on a binomial tree. We employ the well known approach for call pricing on trees developed by Cox, Ross and Rubinstein (1979), known as the CRR model. In our implementation we use the variant proposed by Jarrow and Rudd (1983). The basic details of the scheme are provided below, in order to introduce the notation. It is assumed here that the reader has taken a basic course in finance, and is thereby familiar with these models. If not, the original papers offer simple expositions that are easily followed, and there are several text books on option pricing that also exposit the methodology.

We assume an initial stock price s and that the stock pays no dividends. The strike price of the call option is denoted k, and the time to maturity is t years. The annualized standard deviation of the stock's return is denoted v and the risk free rate is r. The stock price is assumed to follow a binomial branching process over n periods, where each period period is of length $h = t/n$. In each period the stock may proceed from its current value to two values, an "up" value or a "down" one. The stock moves up by a multiplicative factor $u = \exp[(r - 0.5v^2)h + v\sqrt{h}]$. Similarly, it moves down by a multiplicative factor $d = \exp[(r - 0.5v^2)h - v\sqrt{h}]$. These two factors $\{u, d\}$ define the Jarrow-Rudd version of the CRR model. The probability of moving up is denoted q and correspondingly, the probability of moving down is denoted $(1-q)$. In these models, the probability $q = (R - d)/(u - d)$, where $R = \exp(rh)$ is the "drift" of the stock price, equal to the amount that \$1 would grow to in one period at the risk free rate of interest. It is well-established that for there to be no arbitrage in the markets, the following condition should be satisfied: $d \leq R \leq u$. Under these assumptions for u and d, it can easily be shown that the tree recombines, i.e., beginning from a given node, an up-move followed by a down-move in the stock price ends up at the same node on the tree two periods hence.

Once the underlying stock price tree is put in place, the call option price today (denoted c_0) is priced by the method of "backward recursion." What this means is that at the final maturity of the option on the tree, i.e., at the leaves, the payoff of the option is calculated, based on the stock prices at each leaf, i.e., $c_t = \max[0, s_t - k]$, where c_t is the call option value at time t (maturity). This done for all $(n + 1)$ terminal leaves on the stock tree. Then, one period prior to maturity, the option prices at each node on the tree are based on the expected present value of the ensuing two nodes, i.e., $c_{t-h} = [qc_t^u + (1-q)c_t^d]/R$, for all nodes at time $t - h$. The variable c^u is the value of the option at the up-node from the current node, and the value c^d is that of the down-node on the lower branch emanating from the current node. Then, using the option values at time $(t-h)$, the values at time $(t-2h)$ are computed for all nodes at this time. In general at time (jh), the option values for $(j + 1)$ nodes are computed using the formula: $c_{jh} = [qc^u_{(j+1),h} + (1 - q)c^d_{(j+1),h}]/R$.

```
1 #Binomial tree in R
2
3 backrec = function(Cu,Cd,q,R) {
4   res = (q*Cu + (1-q)*Cd)/R
5 }
6
7 bincall = function(s,k,t,v,r,n) {
8   #BASIC SET UP
9   dt = t/n
10  u = exp((r-0.5*v^2)*dt + v*sqrt(dt))
11  d = exp((r-0.5*v^2)*dt - v*sqrt(dt))
12  RR = exp(r*dt)
13  q = (RR-d)/(u-d)
14
15  #STOCK TREE
16  stkp = array(0,dim=c(n+1,n+1))
17  stkp[1,1] = s
18  for (i in 2:(n+1)) {
19      stkp[1,i] = stkp[1,i-1]*u;
20      for (j in 2:i) {
21          stkp[j,i] = stkp[j-1,i-1]*d
22      }
23  }
24
25  #CALL OPTION TREE
26  optval = array(0,n+1)
27  for (j in 1:(n+1)) {
28      optval[j] = max(0,stkp[j,n+1]-k)
29  }
30  for (i in n:1) {
31      res = eachElem(sl,backrec,elementArgs=list(optval[1:i],optval[2:(i+1)],q,RR) )
32      optval[1:i] = as.numeric(rbind(res))
33  }
34  print(optval[1])
35  optval[1]
36 }
37
38 ##### MAIN SEGMENT #####
39 library(nws)
40 numw = 2
41 sl = sleigh(workerCount = numw)
42
43 syst = system.time(bincall(100,100,1.0,0.3,0.03,100))
44 print(syst)
45
46 close(sl)
```

Figure 2 Program code for binomial tree call option pricing with `ParallelR`.

In this recursive fashion we arrive at today's price of the option, i.e., c_0. The program to implement this algorithm is presented in Figure 2.

The program details are as follows. Lines 3–5 contain a function `backrec` that implements the backward recursion described in the previous paragraph. Lines 7–36 contain the main function `bincall` to which the parameters of the call option are passed. Lines 8–13 perform the basic set up of values for the JR model. Lines 15–23 contain the code to generate the entire stock price tree `stkp` for n periods. Lines 26–29 calculate the terminal values of the call option.

Backward recursion is implemented in lines 30–33. Note that for each period i, there are i nodes to be computed based on $(i + 1)$ nodes in the following period. The calculation of each node can be sent to any of the cores of the CPU. On a dual-core machine, each core of the CPU will handle roughly $i/2$ node calculations. For each calculation the function `backrec` is called. The optimal loading of the cores and distribution of work is handled by the `ParallelR` package through the function `eachElem`. The key line to pay attention to here is line 31. The `backrec` function is used, and the parameters that are passed are transmitted in a list. Each item in the list is processed by one of the

cores of the CPU and eventually, the individual results are returned to the variable we called `res` (short for result, but it could be given any name as desired). This list is then collapsed into an array in line 32.

The main program code that loads the NWS library, defines the number of cores to be used, initializes the sleigh and then calls the option pricing function is presented in lines 39–43. Results are printed by line 44, and the sleigh is closed in line 46.

We ran the program for the following parameter values: $s = k = 100$, $t = 1$ year, $v = 0.30$, $r = 0.03$. The number of periods is taken to be $n = 100$. First we used one core of the CPU, and then we used both. Run times for these experiments are as follows:

```
> source("jr_par.R")
[1] 13.29692
   user  system elapsed
  6.767   0.679 106.709

> source("jr_par.R")
[1] 13.29692
   user  system elapsed
  7.470   0.802  80.519
```

We see that the single core run took 107 seconds of elapsed time and for the dual-core elapsed time is 80 seconds. The dual core version is not twice as fast simply because not all aspects of the program have been parallelized, only line 31 of the program. We have also not undertaken several other optimizations (such as vectorization) of the code because we wanted to demonstrate only the ease of use of the `eachElem` function in `ParallelR`. It is seen that we have replaced a loop over all i nodes in the i-th period with a single line of parallelized code.

Not doing vectorization violates one of the golden rules in parallel computing: "fully optimize the serial version before even thinking about a parallel version." It is important to note that the reason we see parallel speedup in these examples is that the serial version is deliberately slow. If the serial version were optimized, the parallel versions may not be much faster and might even be slower than the serial version if the serial version is very fast and the parallel version has too much overhead to result in an overall speedup. The most appropriate way to use `ParallelR` is to price a set of options that have different strike prices/dates. Then we would use an optimized/vectorized CRR/JR function and have `ParallelR` call it many times for the different parameters. In this way we would use these tools and obtain great speedup as well.

4 Discussion

Users of mathematical/statistical packages for scientific computing are always looking for improvements in computing efficiency. The advent of multi-core CPUs has ushered in a new age of high-performance computing by enabling parallelism on a single machine, versus that on clusters of machines. Software has been struggling to keep up with advances in hardware, and scientists find that writing parallel programs is not an easy task. Financial engineers would prefer that their mathematical tools make the transformation of their code to parallel versions as easy as possible.

In this article, we explored how parallelism is implementable with a widely used package, the open source programming language R. We showed how two simple commands, `eachElem` and `eachWorker`, may be used to take advantage of multiple cores with minimal changes to the programming code. We showed how `eachElem` makes running loops in option pricing trees easier to code as well as run faster. We showed how `eachWorker` could be used to speed up a Monte Carlo pricing algorithm. It is clearly the approach of choice for embarrassingly parallel problems like those encountered in Monte Carlo models. We used

R with two add-ons. One, a Network Spaces (NWS) library `nws` and two, a parallel programming package known as `ParallelR`. The entire environment is mostly free and easy to install and use. We hope the examples provided here of simple canonical problems in finance will lead to widespread use of these tools.

Acknowledgments

We thank REvolution Computing for making `ParallelR` available to us for the experiments in this paper.

A Installing R and ParallelR

R is available for free from the CRAN repository. CRAN stands for Comprehensive R Archive Network. The website is `http://cran.r-project.org/`. It is extremely easy to download the entire program and set it up on Windows, Mac or Linux systems. Additional packages are easy to download and install as needed.

The R package with further optimizations is available for free from REvolution Computing as well (`http://www.revolution-computing.com/`). They provide a version of R called REvolution R that is fully compatible with all R packages. It is free, open-source, and may be downloaded from their website.

In order to get `ParallelR`, you need to purchase it separately for a license fee. This version is called REvolution R Enterprise and is essentially REvolution R plus `ParallelR`. All installs are very simple and require no special system administration by the user at all.

For the user interested in starting from scratch and learning R, the documentation on the R project page is of very high quality and any one of the tutorials and basic manuals there will get you started and reasonably proficient in a few hours. See for example the web document `http://cran.r-project.org/doc/manuals/R-intro.html`. This provides a comprehensive introduction to R.

B Using ParallelR

B.1 Starting the Network Spaces Server

To start the NetWorkSpaces server, use the `nwsserver` command. The nwsserver script is in the bin subdirectory of the ParallelR installation directory. Start the server by going to the subdirectory and issuing the following command:

```
/opt/REvolution/ParallelR/bin/
nwsserver start
```

To shut down the server, use the following command:

```
/opt/REvolution/ParallelR/bin/
nwsserver stop
```

B.2 Starting the R parallel environment

Start R (the Revolution R version) by clicking on the application icon.

Load the NetWorkSpaces package, `nws`. You do this by calling the library:

```
library(nws)
```

If the `nws` package is not installed then you will need to install the binary first in the usual manner in which R packages are installed. To do this, click on "Packages & Data" and then run "Package Installer." Search for "nws" in the usual repositories and then go ahead and ask R to install the binary of the package. The entire process is obvious from the various screens that come up.

Verify that the NetWorkSpaces server is running. Any command that uses the server will do, for example:

```
sl <- sleigh()
```

If you do not get an error, the server is running. See the following start-up sequence for an example:

```
Welcome to REvolution R Version 1.3.0 - 100% R and more...
REvolution R Copyright (C) 2009 REvolution Computing, Inc.
-----------------------------------------------------------
Includes R version 2.7.2 (2008-08-25)
Copyright (C) 2008 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'revo()' to visit www.revolution-computing.com for the latest REvolution
R news, 'forum()' for the community forum, or 'readme()' for release notes.
Type 'demo()' for some demos, 'help()' for on-line help, or 'help.start()'
for an HTML browser interface to help. Type 'q()' to quit REvolution R.

> library(nws)
> sl = sleigh()
Error in socketConnection(serverHost, port = port, open = "a+b",
  blocking = TRUE) : cannot open the connection
In addition: Warning message:
In socketConnection(serverHost, port = port, open = "a+b", blocking = TRUE) :
  sanjiv-dass-imac.local:8765 cannot be opened
> sl = sleigh()
```

The command worked the second time because the first time it was run without starting up the `nws` server. After seeing the error, the user starts the NWS server (by issuing the command: `/opt/REvolution/ParallelR/bin/nwsserver start`). Once that is done, the "sleigh()" command works just fine.

Next, we will look at two simple ways in which parallelism may be implemented in the `ParallelR` framework: (i) using the `eachElem` command, and (ii) using the `eachWorker` command.

B.3 Example: `sleigh()` *with* `eachElem`

Here is a simple example of testing the use of R in parallel mode with a system timer.

- The `sleigh` function creates the workers and returns the Sleigh object that is used to maintain the state of the computation.
- The `eachElem` function executes a specified function multiple times in parallel with a varying set of arguments, returning a list of length equal to the number of

executions containing the results of those computations.

- The `eachWorker` function executes a specified function exactly once on each worker in the sleigh. A common use of eachWorker is to provide an initial state for a computation, for example, by loading packages or data. It can be used to run simulations in parallel.

In this example, we use the `eachElem` command.

```
01 #TEST_EACHELEM.R
02 #PROGRAM TO TEST THE EACHELEM
   OPERATION
03 workerdo = function(nsim) {
04     rtot = 0.0
05     for (i1 in 1:nsim) {
06         for (i2 in 1:1000) {
07             rtot = rtot + rnorm(1)
08         }
09     }
10 }
11
12 test1 = function(n) {
13  s = sleigh(workerCount = n)
14  res = eachElem(s,workerdo,
    rep(100,100))
15  close(s)
16 }
17
18 for (n in 1:4) {
19  print(system.time(test1(n)))
20 }
```

Lines 3–10 contain the function `workerdo` that is being executed. Lines 12–16 contain the calling function `test1` that breaks up the total task into subcalls and invokes the function `workerdo`. In lines 18–20, the program is run with the number of elements (workers) increasing from 1 to 4. The key command to note is in line 14. The command `eachElem` is called using the sleigh `s`, a collection of workers, in this case, CPU cores. The function that is being called by the sleigh is `workerdo`. And the `workerdo` function parameter `nsim` is being passed through a variable `rep(100,100)`. What is `rep(100,100)`? It is a 100 element long vector, each element of which is of value 100. Hence, `workerdo` will be called a 100 times, each time with parameter `nsim` equal to a value of 100. What `ParallelR` does through the function `eachElem` is to distribute each of these 100 function calls to `workerdo` to each core of the CPU in an optimal manner. Hence, every one of the 100 calls is assigned to each core when it is free and ready to receive the job.

In this program we keep increasing the number of workers in the sleigh and see if it improves the run time. Note that we run this experiment on a dual-core computer, and so, after $n = 2$, we will not expect to see any further improvement in performance. The run time result is:

```
> source("test_eachelem.R")
   user  system elapsed
  0.511   0.143 121.645
   user  system elapsed
  0.331   0.115  71.109
   user  system elapsed
  0.372   0.143  73.661
   user  system elapsed
  0.507   0.273  75.623
```

Each row of the results above corresponds to $n = 1, 2, 3, 4$ assigned workers (cores). When the worker count is 1, both cores of the dual-core machine are not used, and the program takes longer to run (about 122 seconds of elapsed time) than when the worker count is raised to 2 (an elapsed time of about 71 seconds). Increasing the worker count to greater than 2 does not make much difference as the number of cores is limited to 2. In fact there is a slight increase in elapsed time, because of the additional overhead of managing virtually the third and fourth workers.

Next we implement the same model using the `eachWorker` command.

B.4 Example: `sleigh()` *with* `eachWorker`

```
01 #TEST_EACHWORKER.R
02 #PROGRAM TO TEST EACHWORKER
03 workerdo = function(nsim) {
04     rtot = 0.0
05     for (i1 in 1:nsim) {
06         for (i2 in 1:1000) {
07             rtot = rtot + rnorm(1)
08         }
09     }
10 }
11
12 test2 = function(n) {
13     s = sleigh(workerCount=n)
14     res = eachWorker(s,workerdo,
       floor(10000/n))
15     close(s)
16 }
17
18 for (n in 1:4) {
19  print(system.time(test2(n)))
20 }
```

Lines 3–10 contain the function `workerdo` that is being executed. Lines 12–16 contain the calling function `test2` that breaks up the total task into subcalls and invokes the function `workerdo`. In lines 18–20, the program is run with the number of elements (workers) increasing from 1 to 4. The key command to note is in line 14. The command `eachWorker` is called using the sleigh `s`, a collection of CPU cores. The function being called by the sleigh is `workerdo`. And the `workerdo` function parameter `nsim` is being passed through a variable `floor(10000/n)`. What is `floor(10000/n)`? It is the `nsim` parameter that will be passed to `workerdo` for each worker that is invoked. The total number we want is `nsim` equal to 10000. If there is only one worker then a single call to one core will be made with `nsim` equal to 10000. If there are two workers (cores) to be used, then `nsim` will be 5000 for each worker. Likewise for three and four workers.

The program was run with $n = 1, 2, 3, 4$ and the results are shown below for increasing n. The run times are as follows:

```
> source("test_eachworker.R")
   user  system elapsed
  0.068   0.059 118.499
   user  system elapsed
  0.092   0.073  68.915
   user  system elapsed
  0.120   0.096  71.217
   user  system elapsed
  0.251   0.219  73.617
```

The program is run on a dual-core machine. When only one worker is active, a single core is used and one core remains idle. The elapsed run time is 118 seconds. When we go from one to two workers, the elapsed time drops dramatically to 69 seconds. Thereafter increasing the number of workers does not help much as the number of cores is fully used.

We see that the elapsed time is similar for this example irrespective of whether we use the `eachElem` or the `eachWorker` command.

Notes

1 See `www.rmetrics.org`.

2 See `http://cran.r-project.org/web/views/Finance.html`.

3 See `www.revolution-computing.com`. Why any company would want to be associated with a red hat beats me!

References

Black, F. and Scholes, M. (1973). "The Pricing of Options and Corporate Liabilities." *Journal of Political Economy* **81**, 637–654.

Cox, J., Ross, S. and Rubinstein, M. "Option Pricing: A Simplified Approach." *Journal of Financial Economics* 7, 229–263.

Jarrow, R. and Rudd, A. (1983). Option Pricing, Irwin, Homewood Illinois.

Merton, R. (1973). "Theory of Rational Option Pricing." *Bell Journal of Economics and Management Science* 4(1), 141–183.