Parallel Processing in R
09/07/2025

The Monte Carlo simulation (see the article with the same title) could take a quite long processing time to complete. Parallel processing is to take the advantage of the modern multicore processor of a computer, or a networked cluster that hosts a large number of processors. The computations are distributed across multiple processors and the results are then combined before further analysis. R has several libraries that support parallel processing. This article is to show several examples.

Computer Hardware

The simulation is performed on a MacBook Pro with a 1.4GHz Quad-Core Intel Core i5 processor. There is also an Intel Iris Plus Graphics chip and an 8GB DDR3 Memory chip.

Simulation Script, Single Core

The R studio simulation code is the same as that has been extensively discussed in "Monte Carlo Simulation." For parallel processing, the entire simulation is now captured in a function "do_work(d)" where d is a vector of option contract expiration dates. This function returns the expected trading gain for each d value as a list. The script below shows the procedure using a single processor. "QQQ" is the model name, the contract expiration dates (1, 5, and 20) are assigned to the vector d. The start time for the simulation is stored in a variable st. The list-apply function, lapply(), is employed to carry out the simulation by calling the "do_work" function for each value in d. The results are assigned to a list cagr. The processing time is recorded as the difference between the current time and the start time (st). The average processing time with a single core is approximately 72.9 seconds.

```
tkr="QQQ"

d=c(1,5,20)
st=proc.time()
cagr=lapply(d,do_work)
proc.time()-st
```

Parallel Processing, parLapply()

The parLapply() from the parallel library can be used to replace the lapply() function for parallel processing. Before using this function, the number of cores must be specified (see below). First, the parallel library must be loaded. The variable n_cores specifies the number of cores to be used for the simulation, which in turn becomes the input parameter for the function makeCluster(). The cluster is then "stopped" after the simulation. The function parLapply() is employed to replace the function lapply(), except that the cluster name is specified as the first parameter.

```
library(parallel)
n_cores=1
CL=makeCluster(n_cores)
tkr="QQQ"

d=c(1,5,20)
st=proc.time()
cagr=parLapply(CL,d,do_work)
proc.time()-st
stopCluster(CL)
```

The processing time using the parLapply() parallel processing is summarized below. The improvement from a single core to double cores is quite significant. However, the number of cores larger than two does not bring additional acceleration.

Number of Cores	Processing Time (sec)
1	71.7
2	56.1
3	56.8

Parallel Processing, foreach()

The foreach() function from the foreach library can be carried out as parallel processing when it is "registered" via the doParallel library to the parallel processing cluster. The foreach() function is normally used to replace an iteration loop in an R script. In the example below, however, the vector, d, is used as the argument of the foreach() function.

```
library(parallel)
library(doParallel)
n_cores=1
CL=makeCluster(n_cores)
registerDoParallel(CL)
tkr="QQQ"

d=c(1,5,20)
st=proc.time()
cagr=foreach(d=d) %dopar% {do_work(d)}
proc.time()-st
stopCluster(CL)
```

The processing times using the foreach() parallel processing are shown below. The improvement from a single core to double cores is appreciable. However, the number of cores larger than two does not make additional acceleration.

Number of Cores	Processing Time (sec)
1	74.7
2	61.0
3	61.0

Parallel Processing, mcapply()

The mclapply() function from the parallel library provides an easy replacement for the lapply() function for parallel processing. The parallelism is created via the forking mechanism but it is not available for the Windows operating system. If the mclapply() function is called under a Windows system, the number of cores is forced to be set to one. All the simulations carried out in this article are conducted using MacOS Sequoia version 15.6.1. There are two places where the number of cores needs to be specified. One is the makeCluster() function and the other is the third argument of the mclapply() function.

```
library(parallel)
n_cores=1
CL=makeCluster(n_cores)
tkr="QQQ"

d=c(1,5,20)
st=proc.time()
cagr=mclapply(d,do_work, mc.cores=n_cores)
proc.time()-st
stopCluster(CL)
```

The processing times using the mclapply() function are listed below. There is no improvement from a single core to a multicore processor. Rather, processing time increased drastically when a multicore option is selected.

Number of Cores	Processing Time (sec)
1	72.6
2	270.0
3	257.8

Parallel Processing, free_apply()

The free_apply() function from the free.apply library is supposed to greatly simplify the replacement for the lapply() function for parallel processing.

```
library(future)
library(future.apply)
tkr="QQQ"
d=c(1,5,20)
#plan(multisession, workers = 1)
#plan(multicore, workers = 1)
plan(sequential)
cagr=future_lapply(d, do_work, future.seed=TRUE)
```

However, as a great disappointment, it did not work at all in our case.

Concluding Remarks

Parallel processing promises computational performance improvement especially for Monte Carlo simulations of a large sample size. The original R script has been modified so that the lapply() function is employed for the heavy-duty computation; therefore the script becomes ready for parallel processing. Several R functions for parallel processing have been experimented for their effectiveness. Some of them are designed specifically to replace the lapply() function, and others are to replace the iteration loops.

Among those functions experimented, the parLapply() function offered the greatest performance improvement and the foreach(), although intended for iterations, is the second best. The mclapply() function and free_apply() function do not work at all for the computational case investigated in this article.

Appendix: Modified Script Ready for Parallel Processing

```
do_work=function(d){
retn=function(x){len=length(x);retn=vector(length=len); for (i in 2:len)
retn[i]=x[i]/x[i-1]-1; return(retn)}
prz=function(r){len=length(r);prz=vector(length=len); prz[1]=1;for (i in 2:len)
prz[i]=prz[i-1]*(1+r[i-1]); return(prz)}
c_cagr=function(r,c00,x,d){g=r; for (i in 2:length(r)) {if (r[i]<x) {g[i]=c00+r[i]} else
{g[i]=c00+min(0,x)} }; return(mean(g)*252/d)}
p_cagr=function(r,p00,x,d){fia=r; for (i in 2:length(r)) {if (r[i]<x) {fia[i]=p00+r[i]-
max(0,x)} else {fia[i]=p00} }; return(mean(fia)*252/d)}
CS=function(t,r,sigma,x){w1=(r*t-log(1+x))/(sigma*sqrt(t)); w2=sigma*sqrt(t)/2
return((pnorm(w1+w2)-(1+x)*pnorm(w1-w2)/exp(r*t)))}
PS=function(t,r,sigma,x){w1=(r*t-log(1+x))/(sigma*sqrt(t)); w2=sigma*sqrt(t)/2
return((1+x)*pnorm(w2-w1)/exp(r*t)-pnorm(-w1-w2))}

CC=function(d){
  c1=c0[seq_along(c0) %% d == 0]; r1=retn(c1)
  for (i in 1:length(x)){
    c00[i]=CS(d/252,r,csigma,x[i])
    cagr[i]=c_cagr(r1,c00[i],x[i],d)}
  return(cagr)
}

CSP=function(d){
  c1=c0[seq_along(c0) %% d == 0]; r1=retn(c1)
  for (i in 1:length(x)){
    p00[i]=PS(d/252,r,psigma,x[i])
    cagr[i]=p_cagr(r1,p00[i],x[i],d)}
  return(cagr)
}

#tkr="SPY Model"; n=500000; mu=0.0004472206; std=0.01210494
tkr="QQQ Model"; n=500000; mu=0.0005332306; std=0.01710912
r0=rnorm(n,mu,std); c0=prz(r0); csigma=std*sqrt(252); psigma=csigma;cagr=numeric()
x=seq(from=-0.2,to=0.2, by=0.001);cagr=x; c00=x; p00=x; r=0.04;
return(data.frame(x=x,cagr=CC(d)))
}

dec=function(y){return(sprintf("%.3f",y))}

tkr0="QQQ"; d=c(1,5,20)
st=proc.time()
cagr=lapply(d,do_work)
proc.time()-st

x=cagr[[1]]$x; y1 = cagr[[1]]$cagr; y2= cagr[[2]]$cagr; y3= cagr[[3]]$cagr
plot(x,y1, type="l", main=paste(tkr0,": Max Gain = ", dec(max(y1,y2,y3))),
        xlab=expression(x[K]),ylab="Returns"); grid(lty=2)
lines(x,y2, col="blue"); lines(x, y3, col="red")
```

The statement `cagr=lapply(d,do_work)` can readily be replaced by a parallel processing function parLapply(), foreach(), or mclapply() as described in the text.