**The ‘parallel’ package**

Many computations in R can be made faster by the use of parallel computation. Generally, parallel computation is the simultaneous execution of different pieces of a larger computation across multiple computing processors or cores.

The parallel package can be used to send tasks (encoded as function calls) to each of the processing cores on your machine in parallel.

The mclapply() function essentially parallelizes calls to lapply(). The first two arguments to mclapply() are exactly the same as they are for lapply(). However, mclapply() has further arguments (that must be named), the most important of which is the mc.cores argument which you can use to specify the number of processors/cores you want to split the computation across. For example, if your machine has 4 cores on it, you might specify mc.cores = 4 to break your parallelize your operation across 4 cores (although this may not be the best idea if you are running other operations in the background besides R).

The first thing you might want to check with the parallel package is if your computer in fact has multiple cores that you can take advantage of.

require(parallel)

cores <- detectCores()

cores

## [1] 8

The mclapply() function (and related mc\* functions) works via the fork mechanism on Unix-style operating systems. Because of the use of the fork mechanism, the mc\* functions are generally not available to users of the Windows operating system.

mclapply(1:7, FUN = function(x) return(x), mc.cores = cores-1)

## Error in mclapply(1:7, FUN = function(x) return(x), mc.cores = cores - : 'mc.cores' > 1 is not supported on Windows

Using the forking mechanism on your computer is one way to execute parallel computation but it’s not the only way that the parallel package offers. Another way to build a “cluster” using the multiple cores on your computer is via *sockets*.

Building a socket cluster is simple to do in R with the makeCluster() function.

cl <- makeCluster(cores-1)

The cl object is an abstraction of the entire cluster and is what we’ll use to indicate to the various cluster functions that we want to do parallel computation.

To do a lapply() operation over a socket cluster we can use the parLapply() function.

# sample function

test <- function(){

Sys.sleep(2)

return(TRUE)

}

# call "test" in parallel apply

parLapply(cl = cl, 1:7, fun = function(x) {

test()

})

## Error in checkForRemoteErrors(val): 7 nodes produced errors; first error: could not find function "test"

You’ll notice, unfortunately, that there’s an error in running this code. The reason is that while we have loaded the sulfate data into our R session, the data is not available to the independent child processes that have been spawned by the makeCluster() function. The data, and any other information that the child process will need to execute your code, needs to be exported to the child process from the parent process via the clusterExport() function. The need to export data is a key difference in behavior between the “multicore” approach and the “socket” approach.

# export "test" to the cluster nodes

clusterExport(cl, "test")

# call "test" in parallel apply

parLapply(cl = cl, 1:7, fun = function(x) {

test()

})

## [[1]]

## [1] TRUE

##

## [[2]]

## [1] TRUE

##

## [[3]]

## [1] TRUE

##

## [[4]]

## [1] TRUE

##

## [[5]]

## [1] TRUE

##

## [[6]]

## [1] TRUE

##

## [[7]]

## [1] TRUE

How long does it take?

# parallel

t0 <- proc.time()

xx <- parLapply(cl = cl, 1:7, fun = function(x) {

test()

})

t1 <- proc.time()

t1-t0

## user system elapsed

## 0.00 0.00 2.03

# serial

t0 <- proc.time()

xx <- lapply(1:7, FUN = function(x) {

test()

})

t1 <- proc.time()

t1-t0

## user system elapsed

## 0.00 0.00 14.19

clusterEvalQ() evaluates a literal expression on each cluster node. It can be used to load packages into each node.

# load the zoo package in each node

clusterEvalQ(cl = cl, require(zoo))

## [[1]]

## [1] TRUE

##

## [[2]]

## [1] TRUE

##

## [[3]]

## [1] TRUE

##

## [[4]]

## [1] TRUE

##

## [[5]]

## [1] TRUE

##

## [[6]]

## [1] TRUE

##

## [[7]]

## [1] TRUE

# call zoo functions in parallel apply

parLapply(cl = cl, 1:7, fun = function(x) {

is.zoo(zoo())

})

## [[1]]

## [1] TRUE

##

## [[2]]

## [1] TRUE

##

## [[3]]

## [1] TRUE

##

## [[4]]

## [1] TRUE

##

## [[5]]

## [1] TRUE

##

## [[6]]

## [1] TRUE

##

## [[7]]

## [1] TRUE

Once you’ve finished working with your cluster, it’s good to clean up and stop the cluster child processes (quitting R will also stop all of the child processes).

stopCluster(cl)

**The ‘Rcpp’ package**

Reference:  
<http://heather.cs.ucdavis.edu/~matloff/158/RcppTutorial.pdf>

The Rcpp package provides C++ classes that greatly facilitate interfacing C or C++ code in R packages using the .Call() interface provided by R. It provides a powerful API on top of R, permitting direct interchange of rich R objects (including S3, S4 or Reference Class objects) between  
R and C++.

Maintaining C++ code in it’s own source file provides several benefits (recommended). However, it’s also possible to do inline declaration and execution of C++ code, which will be used in the following example.

Let’s implement the Fibonacci sequence both in R and C++:

\[F\_n = F\_{n-1}+F\_{n-2}\]  
with \(F\_0 = 0\) and \(F\_1=1\).

fibR <- function(n){

if(n==0) return(0)

if(n==1) return(1)

return(fibR(n-1) + fibR(n-2))

}

Rcpp::cppFunction("

int fibC(const int n){

if(n==0) return(0);

if(n==1) return(1);

return(fibC(n-1) + fibC(n-2));

}")

Compare the performance:

require(microbenchmark)

microbenchmark(fibR(20), fibC(20))

## Unit: microseconds

## expr min lq mean median uq max neval

## fibR(20) 8501.602 9666.352 11144.27108 10452.601 11692.950 20956.601 100

## fibC(20) 26.800 29.801 48.61603 38.251 43.801 986.201 100