HPC-R Exercises: Parallelism

Drew Schmidt

11/12/2016

Parallelism

1. Create a vector containing the square root of the numbers 1 to 10000 using 2 cores using:
   - `mclapply()` (skip this if you are using Windows).
   - `parSapply()`
   - `foreach()` with the backend(s) of your choice.
2. Benchmark your solutions above against your best serial implementation from the Section 2 exercises.
3. The Monte Hall game is a well known "paradox" from elementary probability. From Wikipedia:

   ```
   Suppose you're on a game show, and you're given the choice of three doors:
   Behind one door is a car; behind the others, goats. You pick a door, say No. 1,
   and the host, who knows what's behind the doors, opens another door, say No. 3,
   which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it
   to your advantage to switch your choice?
   ```

Simulate one million trials of the Monte Hall game on 2 cores, switching doors every time, to computationally verify the elementary probability result. Compare the run time against the 1 core run time.

Answers

1. Possible solutions are:

```
library(parallel)

n <- 10 # For demonstration purposes
ncores <- 2

### mclapply() --- will not work on Windows!
simplify2array(mclapply(1:n, sqrt, mc.cores=2))
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

```
### parSapply()
cl <- makeCluster(2)
parSapply(cl, 1:n, sqrt)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

```
stopCluster(cl)

### foreach() with snow-like parallel package backend
library(doParallel)
cl <- makeCluster(2)
registerDoParallel(cl)
```

```r
# This is the wrong way to use foreach for a small, quick function executed many times
foreach(i=1:n, .final=simplify2array) %dopar% sqrt(i)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

```r
# This will perform much better, even though it looks gross
foreach(i=1:ncores, .combine=c) %dopar%
{
  roots <- numeric(n/ncores)
  for (j in 1:(n/ncores))
    roots[j] <- sqrt(i*j)

  roots
}
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 1.414214 2.000000
##  [8] 2.449490 2.828427 3.162278
```

```r
stopCluster(cl)
```

2. Using the above implementations:

```r
library(parallel)
library(doParallel)
library(rbenchmark)

n <- 10
ncores <- 2

cl <- makeCluster(ncores)
registerDoParallel(cl)

f <- function(n) simplify2array(mclapply(1:n, sqrt, mc.cores=ncores))
g <- function(n) parSapply(cl, 1:n, sqrt)
h <- function(n, ncores)
{
  foreach(i=1:ncores, .combine=c) %dopar%
  {
    roots <- numeric(n/ncores)
    for (j in 1:(n/ncores))
      roots[j] <- sqrt(i*j)

    roots
  }
}

benchmark(mclapply=f(n), parSapply=g(n), foreach=h(n, ncores),
                    columns=c("test", "replications", "elapsed", "relative"))
```

```
##        test replications elapsed relative
## 3   foreach          100   5.209  100.173
## 1  mclapply          100   0.518    9.962
## 2 parSapply          100   0.052    1.000
```

```r
stopCluster(cl)
```

3. Possible solutions are:

```r
lets_make_a_deal <- function(.)
{
  prize_door <- sample(1:3, size=1)
  first_selection <- sample(1:3, size=1)

  ### Assume we always switch; in that case, we return
  if (prize_door == first_selection)
    return("lose")
  else
    return("win")
}

wincount <- function(winlosevec) sum(winlosevec=="win")



library(parallel)
n <- 10 # For demonstration purposes

# 2 cores
system.time({
  winlose <- simplify2array(mclapply(1:n, lets_make_a_deal, mc.cores=2))
  wincount(winlose) / n
})
```

```
##    user  system elapsed
##   0.004   0.004   0.007
```

```r
# 1 core
system.time({
  winlose <- sapply(1:n, lets_make_a_deal)
  wincount(winlose) / n
})
```

```
##    user  system elapsed
##       0       0       0
```