GitHub This repository Search Explore Features Enterprise Pricing Sign up Sign in

Parallelized text file reading in R and other bits and pieces

15 commits 1 branch 0 releases Fetching contributors

Branch: master Parallel-R-File-Read / +

Fetching latest commit…

LICENSE

README.md

Code

Issues

Pull requests

Pulse

Graphs

HTTPS clone URL

https://github.com

You can clone with HTTPS or Subversion.

Download ZIP

README.md

Parallel R file reads with `parallel` and `data.table` packages

Introduction

There will be times when you will want to read multiple text files in parallel. Data ingest is one of the arduous steps of analysis. For some time now the `parallel` package has existed in R for ... well parallelizing tasks either on the cores of your computer, or executing jobs on a cluster. The `parallel` package brings together functionality from the `snow` package and the `multicore` package. If you are on a Windows OS, you will not be able to take advantage of the `multicore` or `mc` type functionality in the `parallel` package. My advise is to switch to using a proper OS ;-). If you use R for analysis, I would also strongly recommend that you read the book Parallel R *: Data Analysis in the Distributed World By Q. Ethan McCallum , Stephen Weston*. There is a clear difference between using `snow`-type function and `multicore` or `mc`-type functions. The `snow`-type functions are essentially for clusters but can be used for parallel processing on your local machine, whereas the `mc` type function run locally on your machine using the system `fork` command. This is why `mc`-type functions will not run on Windows systems.

The `data.table` package is phenominal for handling large datasets in memory and for carrying out analysis on those datasets. It is truly an invaluable tool. The `read.table` or `read.csv` functions come out of the box with R, and all R programmers have at one time or another used these functions, they are good and convenient but can sometimes be a little on the slow side when text files get large. The `fread` function from the `data.table` package is an equivalent tool to `read.table` but is much faster.

The laptop used has a quad core i7 processor showing 8 threads by hyper-threading and 30GB of memory.

Using the `mclapply` function

The `mclapply` function in the `parallel` package is the multicore equivalent of often used `lapply` function. Very often you are working locally and simply need to read lots of files on a computer or a laptop. In these cases `mclapply` is your friend. It is convenient and with just a few changes you can make best use of your local computing resources.

First we read in the file names with the `list.files` function. This is another very convenient function it is such a task saver. The data used are the Aquisition data sets from Fannie Mae and there are 49 such file in the folder ...

```
fNames <- list.files("raw/", full.names = TRUE)
```

Then we compare times of `lapply` and `mclapply` functions with `read.csv`, and then look at the effect of using the `fread` function on the time taken. The items below are not benchmarks but gives a rough comparison if the amount of time that is spent using the different methods. Each method was run a couple of times and the best time taken.

```
 system.time(x1 <- lapply(fNames, read.csv, sep = "|", header = FALSE))
#user      system elapsed
#209.846   0.851  210.748

system.time(x2 <- mclapply(fNames, read.csv, sep = "|", mc.cores = 8, header = FALSE))
#user      system elapsed
#341.713  10.654  62.682

system.time(x3 <- mclapply(fNames, fread, sep = "|", mc.cores = 8))
#user     system elapsed
#85.441   2.148  23.012
```

The R programmer will know that the time of importance is the elapsed time, notice that the `fread` function doesn't have to be told whether the file has a header or not, it is clever enough to work it out for itself. But this isn't all, the other nice thing about using the `data.table` format is that binding the list of tables together to form one large table is much faster than using data frames:

```
 system.time(x1 <- do.call(rbind, x1))
#user     system elapsed
#38.816   3.310  42.171

dim(x1) # check number of rows and columns
#[1] 19761893       22

system.time(x3 <- do.call(rbind, x3))
#user    system  elapsed
#0.958   0.124   1.083

dim(x3) # check number of rows and columns
#[1] 19761893       22
```

Most R programmers have been in the situation where you are waiting for ages for lists of data frames to bind together to give the final table. The `rbind` of `data.table` objects makes this problem go away.

Logistic Map

Couldn't resist an example simulating from the chaotic Logistic Map. Here is the map function that takes in the r value and number of points desired ...

```
logMap <- function(.r, .n){
  .x <- vector(mode = "numeric", length = .n)
  .x[1] <- .5
  for(i in 1:(.n - 1)){
    .x[i + 1] <- .r*.x[i]*(1 - .x[i])
  }
  return(.x)
}
```

Here are the outputs obtained comparing `lapply` and `mclapply`:

```
.r <- seq(2.99, 3.99, by = 0.001)

system.time(x1 <- mclapply(.r, logMap, .n = 1E4, mc.cores = 8))
#user  system elapsed
#26.789   0.868   3.922

system.time(x2 <- lapply(.r, logMap, .n = 1E4))
#user  system elapsed
#16.017   0.004  16.031
```

The 'secret sauce' here is in the `logMap` function where the output vector `.x` is generated first rather than growing the vector iteratively - basically allowing memory to be allocated first before the analysis.

More parallel programming in R will follow in good time.

© 2015 GitHub, Inc. Terms Privacy Security Contact Help

Status API Training Shop Blog About Pricing