Nicholas Lytal

991834259

STA 250

2/13/14

**Assignment 2: Data Parallelization**

**Introduction**

Reading in and processing enormous data files has a heavy time cost, but we can reduce this cost by taking measures to parallelize our operations. Rather than placing the burden of computation on a single core on a machine, we can instead divide the work among several cores, depending on the steps involved. To gauge the improvements of parallelization, we consider the same problem as in the previous assignment—measuring arrival delays in a series of airline data sets—and compare the times needed to perform the same operation of constructing a delay frequency table to calculate various statistics. Two approaches to parallelization that we take include using R’s parallel package to create clusters, and employing threads in C code to read in data through low-level procedures.

**R Clustering**

Through R’s parallel() package, we can create a cluster with several nodes—each node being a separate instance of R. Each node has access to the functions and data present in the original or “master” node without having to copy it over, assuming the nodes are all on the same machine. Furthermore, certain built-in functions like clusterApply perform operations across nodes, again without requiring multiple copies of the data. A multiple core machine can run separate tasks on separate cores, thus maximizing computational speed for any easily parallelizable procedure. The machine used for this task has four cores, so we will construct nodes keeping that fact in mind—we can make more than four nodes, but they will each have diminishing returns, since multiple nodes will have to exist on the same core.

When clustering, each node must receive a distinct list of arguments (in this case, files) to run to avoid redundancy. However, since each file is not necessarily the same length, it is likely one core will finish processing its files before the others, and it must unfortunately remain idle until the other cores finish their jobs. To circumvent this we use the load-balancing version of clusterApply, clusterApplyLB, which take as an argument the entire list of files and distributes them to open nodes as they become free. This removes a considerable portion of otherwise wasted time, but it does require that all relevant files be on the same machine. Though this is the case here, in other situations the time spent moving data to all relevant machines could outweigh the time saved by using load balancing.

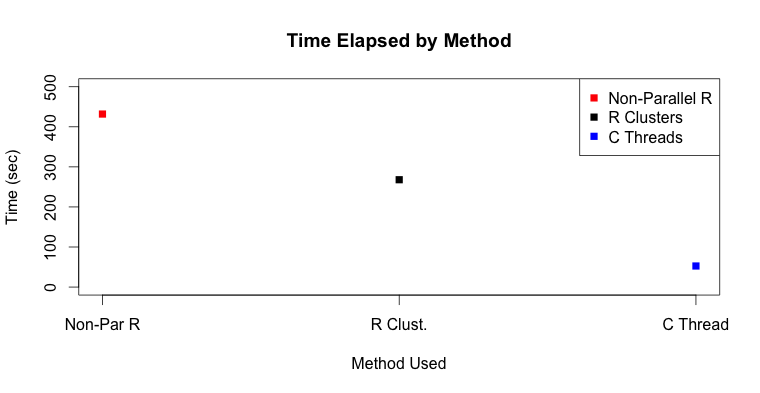
In this particular instance, we read in all the data in parallel before constructing a delay table on the collected values back in the master node.

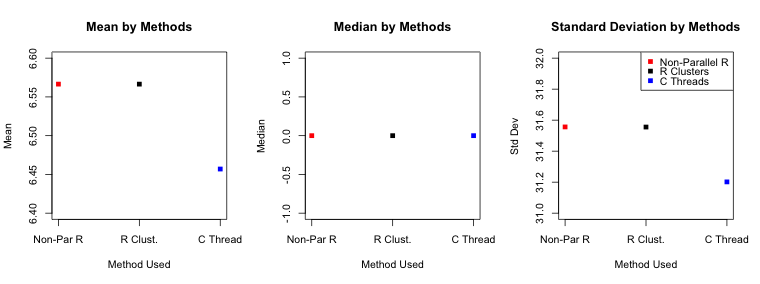
**C Threads**

Despite the usefulness of R clusters, R is still a high-level language and will take more time than we would prefer to perform certain operations. This is largely due to R’s need to search for functions every time they are called, which when applied over millions of data points leads to noticeable slowdown. A low-level language like C has pre-defined definitions for its functions, and they do not need to be searched, so we can considerably speed up the process of reading in data. As with R clustering, using C code to create separate threads lets us process multiple files at once by distributing the computational load among all available cores.

For this method, we use AirlineDelays, an R package whose code makes extensive use of .Call to run C scripts. After using regular expressions to identify the differences between file formats, we then use the function “getDelayTable\_thread”, using as arguments a list object of filenames and an appropriate vector of column numbers to read depending on the file type. This function makes a number of calls to separate scripts that read individual entries, create Table objects to store them in, convert the tables to a format R understands, and combine the results of the separate threads into a single massive delay table containing the values of all files.

**Plots**





**Results**

The initial act of reading data into R makes up the majority of computation time, so it isn’t surprising that we observe a significant increase in speed compared to previous, unparallelized methods. The following plots show the improvements in time compared to the original, non-parallel R method, as well as the actual values acquired.

Due to the fact that parallelization only occurs in the data input portion of the R clustering method, the amount of time saved, while significant, is only about 40%. Ideally, splitting the load over four cores would result in a little under 75% reduction in time. By using Rprof() to profile the code involved, we see that “table” comprises the majority of the computational time. Ideally, we would prefer to parallelize the construction of the tables as well, but we still notice a marked improvement compared to non-parallelized methods.

C threading is definitely the better approach to parallelization, even after considering a 75% drop in time from an ideal case of R clustering. Profiling the code here reveals that almost all the computational work is done in .Call, which is contributing to the low overall time since C does not need to make function calls of its own. There appears to be a slight difference in the values generated from the C threading process compared to those produced when working exclusively in R. It is possible that some entries may be misclassified, but further analysis is needed to indicate where this may be occurring.

**Additional Comments**

Although both of these methods are effective at reducing our overall computation time, there are a few drawbacks to each. One of the most important ones they share is a lack of fault tolerance. Other frameworks such as Hadoop have built-in redundancy—there is an overlap regarding which nodes have which files, so if a job fails for whatever reason, the work that node has accomplished is not immediately lost. With parallel R or C threads, once a node is killed its work dies with it, which could lead to a huge time cost depending on the size of the data involved.

C threads are the best option of the methods featured by a large margin, but there is a lot more setup involved in constructing the code to use this method. The time spent defining all the scripts required may offset the added flexibility and saved time, depending on the circumstances. Using R clustering or another high-level option like Hadoop may offer lesser benefits of parallelization that are nevertheless much faster to execute in their entirety.

**R Code Appendix**

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# STA 250 HW 2 - METHOD 1: Clustering in R

# Basic setup

setwd("~/Desktop/STA\_250\_HW1")

library(parallel)

# stopCluster(cl) # Stop the cluster & free up memory

# First identify the files to subject to clustering

files = list.files(pattern="csv$")

# We will use clusterSplit later to divide the files

# among the clusters

# Once the files are selected, we must create a function

# that reads their contents and outputs the individual

# delay values. It must account for the two file formats.

delays = function(files){

# Splits the nodes files into old and new formats

files.old = files[grep('1[0-9]{3}.csv|200[0-7]{1}.csv', files)]

files.new = files[grep('[a-z].csv', files)]

# Subjects each type of file to the appropriate regex

# commands, using paste to insert filenames to cat

# If a mix of old and new files exists

if(length(files.old) != 0 & length(files.new) != 0)

{

con = pipe(paste("cat", paste(files.old, collapse = " "),

"| cut -f 15 -d, | egrep -v '^$' |

egrep -v 'ArrDelay' \

cat", paste(files.new, collapse = " "),

"| cut -f 45 -d, | egrep -v '^$' |

egrep -v 'ARR\_DEL15'"))

}

# If only old files exist

else if(length(files.old) != 0 & length(files.new) == 0)

{

con = pipe(paste("cat", paste(files.old, collapse = " "),

"| cut -f 15 -d, | egrep -v '^$' |

egrep -v 'ArrDelay'"))

}

# If only new files exist

else if(length(files.old) == 0 & length(files.new) != 0)

{

con = pipe(paste("cat", paste(files.new, collapse = " "),

"| cut -f 45 -d, | egrep -v '^$' |

egrep -v 'ARR\_DEL15'"))

}

# The above distinctions must be made, or else shell will

# attempt to operate on empty space and hang

open(con, open="r") # Opens the defined connection to read

delays = readLines(con) # contains all arrival delays

close(con) # Closes defined connection

delays = as.numeric(delays) # removes parentheses

delays # outputs delay values

}

# Start R profiling

Rprof("/tmp/readSelectedLines.prof") # R profiling

# Start time calculation

start = proc.time()

cl = makeCluster(4, "FORK") # Create cluster of 4 nodes

# filesplit = clusterSplit(cl, files) # split files among nodes

# clusterApply generates a list of size 4, with each one containing

# many files' worth of delay times in numeric form

# delays.list = clusterApply(cl, filesplit, delays) (non-LB way)

# clusterApplyLB generates a list with as many items as there are files,

# with each item containing a single files delay times. By load

# balancing, files are allocated more efficiently, and less time passes.

delays.list = clusterApplyLB(cl, files, delays)

# Condenses the entire list, with all delays, into a single vector

delays.all = rapply(delays.list,c)

# Forms frequency table

delays.all = data.frame(table(delays.all))

# List all possible delay times, removing NAs

delays.all = delays.all[1:nrow(delays.all)-1,]

delays.all # returns delay table

stopCluster(cl) # Cluster computation is over

# All possible delay times

d.time = as.numeric(as.matrix(delays.all[1]))

# Frequency of each time

d.count = as.numeric(as.matrix(delays.all[2]))

n = sum(d.count) # Number of entries

sum.prod = sum(d.time\*d.count) # sum of products

sum.prod2 = sum((d.time^2)\*d.count) # sum of counts by time squared

# Mean of the values

# Takes sum of all products and divides by total # entries

mu = mean((sum.prod)/n)

# Median of the values

# Orders all values and takes middle one

med = sort(rep(d.time,d.count))[n/2]

# Std. dev. of the values

# Uses formula for variance (with n-1 correction),

# then takes square root

sd = sqrt((sum.prod2 - (sum.prod^2)/n)/(n-1))

# End time calculation

time = proc.time()-start

# End R profiling

Rprof(NULL)

delay.lb.prof = summaryRprof("/tmp/readSelectedLines.prof")$by.self

results.calb = list(time = time, results = c(mean = mu, median = med, sd = sd),

system = Sys.info(), session = sessionInfo(),

computer = c(RAM = "16 GB 1600 MHz DDR3",

CPU = "2.6 GHz Intel Core i7",

Software = "OS X 10.8.5 (12F45)"))

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# STA 250 HW 2 - Method 2

# Using R & C code with AirlineDelays package

# \*\*\*\*\*\*\*\*\* CODE TWEAKS \*\*\*\*\*\*\*\*\*

# For the most part, the code utilized is the same as that

# present in the git repository, with the following

# adjustments made inside the files themselves:

# In readrecords.h, expand the range of the tables

# to account for values just beyond the default ranges:

#define MAX\_NUM\_CHARS 5000

#define MAX\_NUM\_VALUES 10000

# In getDelayFreqTable.R, adjust the function defintions

# to include a PACKAGE argument to properly search for the

# required .dll files

getDelayTable =

function(filename, fieldNum = getFieldNum(filename))

{

# Must add PACKAGE="AirlineDelays" for this to function

tt = .Call("R\_getFileDelayTable",

path.expand(filename),

TRUE, as.integer(fieldNum),

PACKAGE="AirlineDelays")

tt[tt > 0]

}

getDelayTable\_thread =

function(files, fieldNum = sapply(files, getFieldNum), numThreads = 4L)

{

# Must add PACKAGE="AirlineDelays" for this to function

tt = .Call("R\_threaded\_multiReadDelays",

files, as.integer(numThreads),

TRUE, as.integer(fieldNum),

PACKAGE="AirlineDelays")

tt[tt > 0]

}

# \*\*\*\*\*\*\*\*\*\*\*\*

# Basic setup

# Set working directory to a folder with all CSV files

setwd("~/Desktop/STA\_250\_HW1")

# First identify the files to read in

# This is divided into old (pre-2008) and new (2008 and on)

# This only gives the initial filenames--need full paths too

files = list.files(pattern="csv$")

files.old = files[grep('1[0-9]{3}.csv|200[0-7]{1}.csv', files)]

files.new = files[grep('[a-z].csv', files)]

# Properly gives files their entire pathnames

files.old = paste(getwd(),"/",files.old, sep = "")

files.new = paste(getwd(),"/",files.new, sep = "")

# Organizes files into a list that the C code can read

filelist = as.list(c(files.old,files.new))

library(AirlineDelays) # First load the package

# This is a vector indicating which column number to read

# for each file. Old & new files use columns 15 and 45

# respectively, but in C we start counting at 0, so we use

# 14 and 44 instead.

fieldNums = c(rep(14L, length(files.old)),

rep(44L, length(files.new)))

Rprof("/tmp/readSelectedLines.prof") # R profiling

start = proc.time()

# This R function calls C code to read entries line by line.

# filelist is a list of all files to read

# fieldNum is a vector of column numbers to read, as above

# numThreads MUST equal the number of files being entered,

# or else the R session will abort.

delays2 = getDelayTable\_thread(filelist, fieldNum = fieldNums,

numThreads = length(filelist))

names(delays2) # These are the factor levels

as.numeric(delays2) # These are the actual values

# This puts the data frame in the same form as Method 1

# for ease of comparison

delays2.df = data.frame(names(delays2), as.numeric(delays2))

# GETTING RELEVANT VALUES

# All possible delay times

d.time = as.numeric(as.matrix(delays2.df[1]))

# Frequency of each time

d.count = as.numeric(as.matrix(delays2.df[2]))

n = sum(d.count) # Number of entries

sum.prod = sum(d.time\*d.count) # sum of products

sum.prod2 = sum((d.time^2)\*d.count) # sum of counts by time squared

# Mean of the values

# Takes sum of all products and divides by total # entries

mu = mean((sum.prod)/n)

# Median of the values

# Orders all values and takes middle one

med = sort(rep(d.time,d.count))[n/2]

# Std. dev. of the values

# Uses formula for variance (with n-1 correction),

# then takes square root

sd = sqrt((sum.prod2 - (sum.prod^2)/n)/(n-1))

time = proc.time() - start

Rprof(NULL)

delay.cr.prof = summaryRprof("/tmp/readSelectedLines.prof")$by.self

# Results for C and R code

results.cr = list(time = time, results = c(mean = mu, median = med, sd = sd),

system = Sys.info(), session = sessionInfo(),

computer = c(RAM = "16 GB 1600 MHz DDR3",

CPU = "2.6 GHz Intel Core i7",

Software = "OS X 10.8.5 (12F45)"))

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# \*\*\* Plots \*\*\*

# With results files from each of the three methods, we can

# plot the data with these commands:

methods = c("Non-Par R","R Clust.", "C Thread")

par(mfrow=c(1,1))

# Time Elapsed

plot(c(results.nonpar$time[3],results.calb$time[3],results.cr$time[3]),

col = c('red','black','blue'), xlab = "Method Used", ylim = c(0,500),

ylab = "Time (sec)", main = "Time Elapsed by Method", pch = 15,

xaxt="n")

axis(1,at=1:3,labels=methods)

legend("topright", c("Non-Parallel R", "R Clusters", "C Threads"),

col = c('red','black','blue'), pch = 15)

par(mfrow=c(1,3))

# Mean Results

plot(c(results.nonpar$results[1],results.calb$results[1],results.cr$results[1]),

col = c('red','black','blue'), xlab = "Method Used", ylab = "Mean",

main = "Mean by Methods", ylim = c(6.4,6.6), pch = 15,

xaxt="n")

axis(1,at=1:3,labels=methods)

# Median Results

plot(c(results.nonpar$results[2],results.calb$results[2],results.cr$results[2]),

col = c('red','black','blue'), xlab = "Method Used", ylab = "Median",

main = "Median by Methods", pch = 15,

xaxt="n")

axis(1,at=1:3,labels=methods)

# Standard Deviation Results

plot(c(results.nonpar$results[3],results.calb$results[3],results.cr$results[3]),

col = c('red','black','blue'), xlab = "Method Used", ylab = "Std Dev",

main = "Standard Deviation by Methods", ylim = c(31,32),

pch = 15, xaxt="n")

axis(1,at=1:3,labels=methods)

legend("topright", c("Non-Parallel R", "R Clusters", "C Threads"),

col = c('red','black','blue'), pch = 15)