High Performance Computing - 3

Contents

Compile Code

R is an interpreted language which makes interactive programming possible. On the other hand, this places a greater burden on the computer which needs to translate R code into a machine understandable format. Lower level programming languages achieve better performance by compiling the language into bytecode before execution. One can take advantage of the efficiency of compiled code by compiling R code or using compiled languages such as C++ within R.

library(compile)

compile library offers handy functions to compile R code. Using different levels of Optimize (3 is highest level of compilation, while 0 is not compiled), one can decide on the extent to compile the code. cmpfun() compiles functions while compile() works on lines of code.

Example

```
library(microbenchmark)
model = function(x,y){
  data = data.frame(x,y)
  model = lm(y~x,data)
```

```
}
a = sample(1:10,1e6,replace=T)
b = sample(1:10,1e6,replace=T)
library(compiler)
model_compiled = cmpfun(model,options=list(optimize=3))
microbenchmark(model(a,b),model_compiled(a,b),times=5)
```

```
## Unit: milliseconds
##                  expr      min       lq     mean   median       uq      max
##           model(a, b) 143.9032 146.4033 152.8056 150.8452 152.7769 170.0993
##  model_compiled(a, b) 145.8085 150.2596 164.7655 164.6689 180.1826 182.9081
##  neval
##      5
##      5
```

Another Example

```
mov.avg = function(x, n=20) {
  total = numeric(length(x) - n + 1)
  for (i in 1:n) {
    total = total + x[i:(length(x) - n + i)]
  }
  total/n
}

library(compiler)
# different levels of optimization
mov.avg.compiled0 = cmpfun(mov.avg, options=list(optimize=0))
mov.avg.compiled1 = cmpfun(mov.avg, options=list(optimize=1))
mov.avg.compiled2 = cmpfun(mov.avg, options=list(optimize=2))
mov.avg.compiled3 = cmpfun(mov.avg, options=list(optimize=3))

x = runif(100)
microbenchmark(mov.avg(x),mov.avg.compiled0(x),
               mov.avg.compiled1(x),
               mov.avg.compiled2(x),
               mov.avg.compiled3(x))
```

```
## Unit: microseconds
##                  expr    min      lq     mean  median      uq      max neval
##            mov.avg(x) 13.001 14.0505 67.62800 14.6010 21.0510 5056.900   100
##  mov.avg.compiled0(x) 27.401 29.2510 33.32803 30.1510 37.9515   54.200   100
##  mov.avg.compiled1(x) 14.801 15.7005 19.26206 16.1010 22.7510   63.701   100
##  mov.avg.compiled2(x) 12.901 13.9010 17.16595 14.4505 21.3515   32.300   100
##  mov.avg.compiled3(x) 12.801 13.6010 16.73803 14.1510 20.7010   28.901   100
```

One last example here will illustrate that gains from cmpfun() are modest to sometimes none.

```
permutations = function(x){
  product = 1
  for (i in 1:x){
    product = product*i
    i = i + 1
  }
  product
}
permutations_compiled = cmpfun(permutations,list(optimize=3))
```

```
microbenchmark(permutations(1e7),permutations_compiled(1e7),times=5)
```

```
## Unit: milliseconds
##                          expr      min       lq     mean   median       uq
##           permutations(1e+07) 402.3763 436.9298 435.8577 444.1847 446.5674
##  permutations_compiled(1e+07) 396.9430 405.7589 450.2805 422.7520 505.6998
##       max neval
##  449.2302     5
##  520.2490     5
```

JIT Compiler

compile library also features enableJIT which will compile sections of code between enableJIT tags
Example

```
library(compiler)
enableJIT(level=3)
```

```
## [1] 3
```

```
# 0: It disables JIT.
# 1: It compiles functions before their first use.
# 2: In addition, it compiles functions before they are duplicated. This is useful for some packages li
# 3: It compiles loops before they are executed.
microbenchmark(mov.avg(x))
```

```
## Unit: microseconds
##        expr    min     lq     mean  median     uq    max neval
##  mov.avg(x) 13.201 13.601 14.67109 14.0015 14.501 42.601   100
```

```
enableJIT(level=0)
```

```
## [1] 3
```

Another example

```
model = function(){
  x = sample(1:10,size = 10000,replace = T)
  y = rnorm(10000,mean = 100,sd = 5)
  data = data.frame(x,y)
  reg = lm(y~x,data)
  nonLinear = lm(y~poly(x,5),data)
  library(splines)
  spline = lm(y~ns(x,5),data)
  library(mgcv)
  gam_model = gam(y~s(x),data=data)
  library(rpart)
  tree = rpart(y~x,data)
  library(randomForest)
  forest = randomForest(y~x,data,ntree=1000)
  library(gbm)
  boost = gbm(y~x,data,distribution = 'gaussian',n.trees = 500,interaction.depth = 1,n.minobsinnode = 5
}

model_compiled3 = cmpfun(model,options = list(optimize=3))
microbenchmark(model(),model_compiled3(),times=5)
```

```
## Unit: seconds
##               expr      min       lq     mean   median       uq      max neval
```

```
##            model() 3.363716 3.464190 3.774443 3.639127 3.726024 4.679161     5
##  model_compiled3() 3.380135 3.532492 3.663735 3.613817 3.724894 4.067337     5
```

```
enableJIT(level = 3)
```

```
## [1] 0
```

```
microbenchmark(model(),times=5)
```

```
## Unit: seconds
##     expr      min       lq     mean   median       uq     max neval
##  model() 3.412427 3.495484 3.528647 3.497621 3.572963 3.66474     5
```

```
enableJIT(level = 0)
```

```
## [1] 3
```

Another example

```
s3 = function(n) {
  sum = numeric(length(n))
  for (i in 1:n){
    sum = sum+i
  }
  sum
}
s3_compiled0 = cmpfun(s3,options = list(optimize=0))
s3_compiled3 = cmpfun(s3,options = list(optimize=3))
microbenchmark(s3(1e8),s3_compiled0(1e8),s3_compiled3(1e8),times=5)
```

```
## Unit: seconds
##                  expr       min        lq      mean    median        uq
##            s3(1e+08) 29.903256 30.478680 31.283789 31.665612 31.906859
##  s3_compiled0(1e+08) 30.334827 30.581942 31.260040 31.013369 31.094167
##  s3_compiled3(1e+08)  2.439767  2.502171  2.590283  2.506463  2.593899
##        max neval
##  32.464536     5
##  33.275894     5
##   2.909116     5
```

```
microbenchmark(s3(1e8),times=5)
```

```
## Unit: seconds
##       expr     min       lq     mean   median       uq      max neval
##  s3(1e+08) 30.2413 30.36424 30.85677 30.50813 31.34495 31.82523     5
```

```
enableJIT(level=3)
```

```
## [1] 0
```

```
microbenchmark(s3(1e8),times=5)
```

```
## Unit: seconds
##       expr      min       lq     mean   median       uq     max neval
##  s3(1e+08) 2.477851 2.494915 2.523469 2.528859 2.549452 2.56627     5
```

```
enableJIT(level=0)
```

```
## [1] 3
```

Compiled Languages

The ultimate way to benefit from the power of compiled languages is to write code in C++. But, keep in mind you need to install development tools for this.

For Windows install Rtools and for Mac install the Xcode Command Line Tools. Two R packages that support writing C++ code are `library(inline)` and `library(Rcpp)`. The benefit of the latter is that it integrates well with RStudio.

GPU

In order take advantage of GPUs, one must have a GPU on the machine. Thanks to GPU programming in CUDA done on nVidia chips and in OpenCL on non-nVidia chips, we have a number of handy GPU packages including

- gputools
- gmatrix
- RCUDA
- OpenCL

Addressing Constraint of RAM

By default R loads all the data into memory and also keeps objects created during the session in memory rather than writing to disk. This places a logical limit on the data that can be handled. There are three approaches to address the constraint of RAM.

Use RAM Wisely

When instructed to create an identical copy of an object, R uses a pointer to a memory location rather than actually creating another object. This natural behavior makes efficient use of RAM. The following example creates two objects with the same sequence of numbers. Since they are instantiated as different objects, they take up different addresses in memory, and therefore their own quota of memory.

```
library(pryr)
```

```
##
## Attaching package: 'pryr'
```

```
## The following object is masked from 'package:mgcv':
##
##     %.%
```

```
x = 1:1e6
object_size(x)
```

```
## 680 B
```

```
y = 1:1e6
object_size(y)
```

```
## 680 B
```

```
object_size(x,y)
```

```
## 832 B
```

```
address(x)
```

```
## [1] "0x2305d318"
```

```
address(y)
```

```
## [1] "0x2649b268"
```

Instead, if we create the second object (y) from the first object (x), R will only keep one copy - so same address and less memory used.

```
x = 1:1e6
object_size(x)
```

```
## 680 B
```

```
y = x
object_size(y)
```

```
## 680 B
```

```
object_size(x,y)
```

```
## 680 B
```

```
address(x)
```

```
## [1] "0x1c6f3e30"
```

```
address(y)
```

```
## [1] "0x1c6f3e30"
```

However, as soon as y is edited, R will be forced to create a copy, thus giving it a new address and independent memory.

```
x = 1:1e6
object_size(x)
```

```
## 680 B
```

```
y = x
y[1] = 1L
object_size(y)
```

```
## 4,000,048 B
```

```
object_size(x,y)
```

```
## 4,000,728 B
```

```
address(x)
```

```
## [1] "0x21d5fe40"
```

```
address(y)
```

```
## [1] "0x8600010"
```

Take out the trash

Remove intermediate data when not needed

```
ls()
```

```
##  [1] "a"                     "b"                     "model"
##  [4] "model_compiled"        "model_compiled3"       "mov.avg"
##  [7] "mov.avg.compiled0"     "mov.avg.compiled1"     "mov.avg.compiled2"
## [10] "mov.avg.compiled3"     "permutations"          "permutations_compiled"
## [13] "s3"                    "s3_compiled0"          "s3_compiled3"
## [16] "x"                     "y"
```

```
rm(x,y)
ls()
```

```
##  [1] "a"                     "b"                     "model"
##  [4] "model_compiled"        "model_compiled3"       "mov.avg"
##  [7] "mov.avg.compiled0"     "mov.avg.compiled1"     "mov.avg.compiled2"
## [10] "mov.avg.compiled3"     "permutations"          "permutations_compiled"
## [13] "s3"                    "s3_compiled0"          "s3_compiled3"
```

Calculate on the fly instead of storing

Consider a cluster analysis example

```
library(Matrix); library(pryr)
A = matrix(rnorm(1E5), nrow = 1E4, ncol = 10)
d = dist(A,method = 'euclidean')
clust = hclust(d,method = 'ward.D2')
clusterGroups = cutree(clust,k = 3)
object_size(A,d,clust,clusterGroups) # we created a lot of objects along the way
```

```
## 401,003,368 B
```

```
rm(list = c('A','d','clust'))
```

Move inactive data to disk

```
library(ggplot2)
data = diamonds[sample(1:nrow(diamonds),size = 1e6,replace=T),]
saveRDS(data,'data.rds')
rm(data)
data = readRDS('data.rds')
```

Use memory-efficient data structures

Data Type

Data type can have a tremendous influence on size of data, so when possible, use smaller data types. To illustrate, examine the size of the following vectors.

```
object.size(logical(100))
```

```
## 448 bytes
```

```
object.size(integer(100))
```

```
## 448 bytes
```

```
object.size(numeric(100))
```

```
## 848 bytes
```

```
object.size(complex(100))
```

```
## 1648 bytes
```

```
object.size(character(100))
## 904 bytes
object.size(sample(c('M','F'),size = 100,T))
## 960 bytes
object.size(factor(sample(c('M','F'),size = 100,T)))
## 960 bytes
object.size(sample(c('M','F'),size = 1e6,T))
## 8000160 bytes
object.size(factor(sample(c('M','F'),size = 1e6,T)))
## 4000560 bytes
```

Sparse Matrices

Where possible, use sparse matrices. Logical sparse matrices are even more compact

```
data = rnorm(1e6)
data[sample(length(data),size = 0.7*length(data))] = 0
object.size(data)
## 8000048 bytes
library(Matrix)
m.dense = Matrix(data,nrow=1000,ncol=1000,sparse=F)
object.size(m.dense)
## 8001176 bytes
m.sparse = Matrix(data,nrow=1000,ncol=1000,sparse=T)
object.size(m.sparse)
## 3605504 bytes
m.sparse.logical = Matrix(as.logical(data),nrow=1000,ncol=1000,sparse=T)
object.size(m.sparse.logical)
## 2405504 bytes
```

Symmetric Matrices

Symmetric matrices such as distance and correlation matrices contain the same information above and below the diagonal. These can be condensed by converting to a dspMatrix class.

```
library(Matrix)
data = matrix(rnorm(1e6), 1e4, 1E2)
x = cor(data)
isSymmetric(x)
## [1] TRUE
y = as(x, "dspMatrix")
object.size(x)
## 80216 bytes
object.size(y)
## 41800 bytes
```

Bit Vectors

Binary data can be represented in an even more efficient way, using bit vectors.

```
library(bit)
l = sample(c(TRUE, FALSE), 1e6, TRUE)
b = as.bit(l)
object.size(l) ## 4000040 bytes
```

```
## 4000048 bytes
```

```
object.size(b) ## 126344 bytes
```

```
## 126512 bytes
```

Memory-Mapped files

For datasets that are to large to load into memory, one solution is to store them on a disk in the form of memory-mapped files and load the data into the memory for processing one small chunk at a time. There are two packages that provide memory-mapped files to work with large datasets: `bigmemory` and `ff`

Parallel Processing

In the last couple of years, computers have gotten better but not so much because of faster processors but owing to multi-core processors that do parallel processing. Unfortunately, R is single threaded. Fortunately, there are now ways to do parallel computing, thereby leveraging the power of multiple cores.

Many R programs can be written in order to run in parallel but the extent of parallelism depends on the computing task involved. On one end, there are embarassingly parallel tasks where there are no dependencies betwen parallel subtasks. On the other hand, there are tasks where one model relies on inputs from a previous model. Such processes cannot be parallelized.

Broadly speaking, there are two classes of parallel processing, data parallelism where the dataset is divided and the subsets distributed to different processors, and task parallelism where tasks are distributed to and excuted on different processors in parallel. There are several R packages that allow code to be executed in parallel, one of them is a package called `parallel`.

Here is an example

```
library(parallel)
detectCores()
```

```
## [1] 8
```

```
data = sample(1:10,size = 1e8,replace=T)
dim(data) = c(100000,1000)

# method 1
apply(data,2,mean)
```

```
##    [1] 5.50139 5.49644 5.49223 5.50354 5.50497 5.49419 5.49896 5.49887 5.50030
##   [10] 5.49632 5.50148 5.49827 5.50739 5.51482 5.50910 5.47891 5.49284 5.50199
##   [19] 5.50040 5.50040 5.48435 5.48717 5.50112 5.50383 5.50051 5.50986 5.49165
##   [28] 5.48578 5.50033 5.48732 5.49437 5.50369 5.49665 5.50861 5.49852 5.50472
##   [37] 5.48712 5.50517 5.49725 5.49924 5.52572 5.49756 5.49737 5.48810 5.49279
##   [46] 5.51386 5.50325 5.49385 5.50458 5.50684 5.49719 5.49883 5.50945 5.49529
##   [55] 5.49349 5.50318 5.49789 5.49964 5.50170 5.50132 5.50856 5.50592 5.50240
##   [64] 5.50213 5.51782 5.51014 5.49841 5.48654 5.49701 5.49452 5.50663 5.48961
##   [73] 5.49999 5.48751 5.50086 5.49364 5.49332 5.48972 5.49564 5.48508 5.49671
```

```
##  [82] 5.49946 5.50007 5.51849 5.49967 5.50030 5.50310 5.50136 5.50219 5.48971
##  [91] 5.50166 5.50184 5.50422 5.50034 5.50573 5.48151 5.51020 5.50269 5.49921
## [100] 5.49262 5.49317 5.50116 5.49411 5.50041 5.50918 5.51749 5.50730 5.50032
## [109] 5.49316 5.50822 5.50185 5.49501 5.49428 5.49759 5.49411 5.49272 5.50388
## [118] 5.50050 5.48087 5.50664 5.49727 5.48134 5.49005 5.49343 5.50093 5.49703
## [127] 5.51095 5.50684 5.49284 5.51334 5.50253 5.50509 5.49268 5.50127 5.49592
## [136] 5.51608 5.50400 5.50527 5.50610 5.50085 5.50049 5.49391 5.50571 5.50144
## [145] 5.48931 5.50174 5.49950 5.50502 5.51437 5.48240 5.51002 5.51706 5.49242
## [154] 5.49994 5.50699 5.50832 5.51104 5.49785 5.50391 5.50001 5.50580 5.51862
## [163] 5.48837 5.49290 5.50756 5.50737 5.51101 5.49070 5.51095 5.50364 5.51226
## [172] 5.48589 5.49591 5.48435 5.48793 5.49998 5.49810 5.49720 5.50870 5.49707
## [181] 5.51280 5.50229 5.48806 5.51292 5.49435 5.50349 5.48819 5.49743 5.50641
## [190] 5.49700 5.49625 5.50463 5.51452 5.50080 5.51776 5.50507 5.50285 5.50614
## [199] 5.50552 5.49370 5.50801 5.49644 5.50192 5.49154 5.51353 5.50764 5.48819
## [208] 5.48345 5.50199 5.51266 5.50519 5.49450 5.51916 5.50578 5.50298 5.48630
## [217] 5.49798 5.50117 5.49852 5.48886 5.50580 5.51761 5.50190 5.49580 5.49776
## [226] 5.49105 5.51619 5.49908 5.49798 5.49726 5.48501 5.49440 5.49414 5.50845
## [235] 5.50704 5.50419 5.50960 5.49744 5.51134 5.49072 5.51226 5.49665 5.48997
## [244] 5.50703 5.49178 5.50037 5.48009 5.50259 5.53126 5.50889 5.49540 5.49767
## [253] 5.49298 5.49119 5.50515 5.49922 5.50797 5.49373 5.50466 5.46957 5.51359
## [262] 5.48453 5.50197 5.49947 5.51665 5.49335 5.51763 5.49112 5.49942 5.50183
## [271] 5.50876 5.49668 5.49491 5.50470 5.51197 5.51943 5.49938 5.50269 5.50262
## [280] 5.49941 5.48969 5.49865 5.50097 5.48689 5.48736 5.49456 5.50712 5.51390
## [289] 5.50687 5.50860 5.48404 5.50243 5.49542 5.50583 5.50178 5.50622 5.48991
## [298] 5.49929 5.50383 5.49073 5.50628 5.50178 5.49764 5.51884 5.49677 5.49423
## [307] 5.49343 5.51361 5.48611 5.50173 5.48418 5.50341 5.50166 5.50113 5.51318
## [316] 5.49773 5.50292 5.49926 5.49530 5.49358 5.49473 5.50954 5.50871 5.49909
## [325] 5.50097 5.49419 5.50313 5.51273 5.48568 5.51260 5.49001 5.49619 5.50853
## [334] 5.51046 5.48709 5.50467 5.50866 5.50679 5.49148 5.50428 5.49094 5.51188
## [343] 5.50404 5.49822 5.51218 5.50671 5.48448 5.49411 5.49964 5.51466 5.49921
## [352] 5.49321 5.50679 5.49712 5.48904 5.49924 5.50674 5.48850 5.49970 5.50397
## [361] 5.51782 5.50483 5.49233 5.50499 5.48986 5.51314 5.49958 5.49074 5.49917
## [370] 5.48761 5.49996 5.49961 5.51676 5.49874 5.49962 5.48495 5.49788 5.48061
## [379] 5.49240 5.48367 5.48318 5.51377 5.49869 5.49893 5.48916 5.51130 5.50982
## [388] 5.49684 5.49095 5.50535 5.51020 5.50526 5.48424 5.51150 5.49525 5.48896
## [397] 5.51657 5.48787 5.49761 5.49953 5.51054 5.50232 5.50572 5.48969 5.49954
## [406] 5.49944 5.50068 5.51304 5.50016 5.50077 5.49120 5.50059 5.50438 5.49735
## [415] 5.47650 5.49399 5.50648 5.51693 5.49416 5.49172 5.48631 5.50294 5.50524
## [424] 5.49343 5.49779 5.49927 5.50441 5.50225 5.50321 5.49516 5.50229 5.50082
## [433] 5.49514 5.48809 5.50000 5.50060 5.51057 5.49658 5.49785 5.49269 5.50058
## [442] 5.49841 5.50155 5.51226 5.50328 5.49184 5.48800 5.51811 5.50749 5.50570
## [451] 5.48873 5.50006 5.50593 5.50013 5.50479 5.49798 5.49296 5.49678 5.49760
## [460] 5.49805 5.49593 5.49518 5.50585 5.48709 5.50256 5.49971 5.51109 5.51270
## [469] 5.50018 5.50773 5.47976 5.49977 5.49045 5.51893 5.50200 5.50205 5.51041
## [478] 5.50472 5.49079 5.47196 5.49074 5.51853 5.49193 5.48833 5.50420 5.49946
## [487] 5.49945 5.50679 5.49010 5.49756 5.48700 5.51355 5.51461 5.49430 5.51089
## [496] 5.49543 5.50367 5.50924 5.51202 5.50439 5.50860 5.49912 5.49932 5.51752
## [505] 5.48438 5.49180 5.48975 5.51214 5.49319 5.50884 5.50904 5.48727 5.51037
## [514] 5.50524 5.50744 5.50918 5.48743 5.49786 5.50511 5.49349 5.48985 5.50641
## [523] 5.50342 5.49637 5.50836 5.50258 5.48597 5.50697 5.50538 5.49986 5.50256
## [532] 5.49451 5.49432 5.49983 5.48654 5.50463 5.47240 5.50132 5.50657 5.49586
## [541] 5.48624 5.51235 5.49316 5.50631 5.50124 5.49307 5.48379 5.50063 5.48521
## [550] 5.48690 5.48507 5.48507 5.48490 5.49728 5.50605 5.50911 5.50442 5.49177
## [559] 5.48415 5.47924 5.48995 5.50237 5.49921 5.47829 5.49911 5.49481 5.50222
```

```
##  [568] 5.50665 5.49260 5.50700 5.50372 5.50459 5.50752 5.49781 5.48874 5.50159
##  [577] 5.49367 5.49005 5.50778 5.50932 5.50016 5.50766 5.49887 5.49210 5.49802
##  [586] 5.50094 5.49307 5.48789 5.50564 5.49584 5.50530 5.49040 5.50033 5.47495
##  [595] 5.51241 5.50639 5.50585 5.48855 5.50298 5.51053 5.50474 5.49951 5.50612
##  [604] 5.49408 5.50656 5.50791 5.49987 5.50462 5.49516 5.49504 5.49862 5.49667
##  [613] 5.51654 5.49370 5.49372 5.50018 5.49167 5.49514 5.48592 5.50095 5.47882
##  [622] 5.49203 5.49659 5.49257 5.51279 5.49988 5.49166 5.50501 5.49682 5.49688
##  [631] 5.49512 5.50200 5.47752 5.51204 5.50285 5.51786 5.49250 5.51202 5.47769
##  [640] 5.50647 5.49623 5.50153 5.49741 5.51103 5.49566 5.50271 5.49808 5.49874
##  [649] 5.48762 5.51230 5.50479 5.50703 5.51074 5.49195 5.49579 5.50177 5.50624
##  [658] 5.48646 5.52193 5.49569 5.49540 5.50692 5.51192 5.50883 5.49094 5.51663
##  [667] 5.49912 5.48226 5.49606 5.48455 5.48582 5.49905 5.49563 5.48751 5.50433
##  [676] 5.49400 5.49886 5.50410 5.51160 5.50584 5.49676 5.49783 5.49228 5.51488
##  [685] 5.49998 5.49730 5.50530 5.49755 5.50974 5.49502 5.51053 5.48746 5.49675
##  [694] 5.49933 5.50430 5.50447 5.50266 5.49553 5.51106 5.50413 5.50123 5.48561
##  [703] 5.49255 5.49888 5.51044 5.49770 5.50499 5.50116 5.48881 5.49867 5.48948
##  [712] 5.50543 5.50936 5.48597 5.49772 5.50543 5.49061 5.48882 5.50901 5.50492
##  [721] 5.48534 5.50072 5.51141 5.50846 5.48892 5.50409 5.50788 5.49615 5.50857
##  [730] 5.49485 5.48836 5.50668 5.50921 5.49101 5.48762 5.50033 5.49285 5.51418
##  [739] 5.49377 5.50267 5.48646 5.49312 5.50713 5.50963 5.49931 5.49398 5.50028
##  [748] 5.49418 5.49765 5.49779 5.49899 5.49880 5.50580 5.50012 5.49458 5.50186
##  [757] 5.49357 5.49999 5.51330 5.52821 5.51218 5.49938 5.49032 5.49624 5.52020
##  [766] 5.47446 5.50760 5.48835 5.49280 5.49215 5.50284 5.49613 5.49519 5.50466
##  [775] 5.48548 5.49144 5.48632 5.50659 5.48819 5.50244 5.50294 5.48341 5.48417
##  [784] 5.50358 5.50294 5.49577 5.51184 5.50114 5.50454 5.51459 5.50000 5.48752
##  [793] 5.50729 5.49348 5.49370 5.49396 5.49207 5.49153 5.51022 5.48449 5.48234
##  [802] 5.50194 5.50507 5.49364 5.49446 5.49785 5.49155 5.50156 5.49752 5.50805
##  [811] 5.51357 5.50507 5.49747 5.51376 5.50971 5.49444 5.48705 5.49716 5.48724
##  [820] 5.50780 5.50351 5.49105 5.50622 5.50525 5.51433 5.51506 5.50189 5.49805
##  [829] 5.49485 5.51992 5.49602 5.49665 5.50501 5.50986 5.49498 5.51076 5.47495
##  [838] 5.49865 5.49548 5.50420 5.49889 5.51307 5.49040 5.50059 5.48405 5.50999
##  [847] 5.50685 5.50472 5.52420 5.49274 5.50145 5.51182 5.49689 5.49268 5.49412
##  [856] 5.49995 5.49444 5.49427 5.49246 5.49253 5.49854 5.49446 5.49074 5.51859
##  [865] 5.50765 5.50781 5.49765 5.51536 5.51881 5.48993 5.50266 5.50133 5.50457
##  [874] 5.50417 5.48210 5.50383 5.50356 5.50233 5.50707 5.49825 5.50157 5.49668
##  [883] 5.50498 5.50201 5.50739 5.50964 5.50004 5.50181 5.52160 5.49416 5.49606
##  [892] 5.50013 5.50046 5.49626 5.48588 5.50796 5.48895 5.50389 5.50306 5.49803
##  [901] 5.48907 5.50859 5.49231 5.51024 5.50556 5.50162 5.50701 5.49760 5.51048
##  [910] 5.49240 5.49691 5.50315 5.49767 5.50023 5.50289 5.50124 5.48354 5.49636
##  [919] 5.49673 5.50204 5.47578 5.49530 5.50514 5.50297 5.51278 5.50287 5.49413
##  [928] 5.49430 5.50542 5.50165 5.48943 5.49744 5.48867 5.50992 5.48924 5.49095
##  [937] 5.48405 5.51309 5.49510 5.48961 5.47885 5.50446 5.51571 5.50316 5.51266
##  [946] 5.50687 5.50258 5.50127 5.49277 5.49012 5.49510 5.50329 5.50389 5.49395
##  [955] 5.49270 5.50190 5.49598 5.50538 5.49595 5.49838 5.50787 5.51125 5.50968
##  [964] 5.49332 5.50376 5.50177 5.51040 5.49770 5.52082 5.51301 5.48756 5.49821
##  [973] 5.50683 5.50796 5.49985 5.50261 5.52459 5.48210 5.49559 5.50985 5.50083
##  [982] 5.49762 5.49440 5.48690 5.49239 5.49497 5.51449 5.50674 5.48948 5.48816
##  [991] 5.49419 5.49661 5.49654 5.50681 5.49732 5.49600 5.50649 5.49396 5.50013
## [1000] 5.50743
```

```
# method 2 - parallelize
cl = makeCluster(4)
parApply(cl,data,2,mean)
```

```
##    [1] 5.50139 5.49644 5.49223 5.50354 5.50497 5.49419 5.49896 5.49887 5.50030
```

```
##   [10] 5.49632 5.50148 5.49827 5.50739 5.51482 5.50910 5.47891 5.49284 5.50199
##   [19] 5.50040 5.50040 5.48435 5.48717 5.50112 5.50383 5.50051 5.50986 5.49165
##   [28] 5.48578 5.50033 5.48732 5.49437 5.50369 5.49665 5.50861 5.49852 5.50472
##   [37] 5.48712 5.50517 5.49725 5.49924 5.52572 5.49756 5.49737 5.48810 5.49279
##   [46] 5.51386 5.50325 5.49385 5.50458 5.50684 5.49719 5.49883 5.50945 5.49529
##   [55] 5.49349 5.50318 5.49789 5.49964 5.50170 5.50132 5.50856 5.50592 5.50240
##   [64] 5.50213 5.51782 5.51014 5.49841 5.48654 5.49701 5.49452 5.50663 5.48961
##   [73] 5.49999 5.48751 5.50086 5.49364 5.49332 5.48972 5.49564 5.48508 5.49671
##   [82] 5.49946 5.50007 5.51849 5.49967 5.50030 5.50310 5.50136 5.50219 5.48971
##   [91] 5.50166 5.50184 5.50422 5.50034 5.50573 5.48151 5.51020 5.50269 5.49921
##  [100] 5.49262 5.49317 5.50116 5.49411 5.50041 5.50918 5.51749 5.50730 5.50032
##  [109] 5.49316 5.50822 5.50185 5.49501 5.49428 5.49759 5.49411 5.49272 5.50388
##  [118] 5.50050 5.48087 5.50664 5.49727 5.48134 5.49005 5.49343 5.50093 5.49703
##  [127] 5.51095 5.50684 5.49284 5.51334 5.50253 5.50509 5.49268 5.50127 5.49592
##  [136] 5.51608 5.50400 5.50527 5.50610 5.50085 5.50049 5.49391 5.50571 5.50144
##  [145] 5.48931 5.50174 5.49950 5.50502 5.51437 5.48240 5.51002 5.51706 5.49242
##  [154] 5.49994 5.50699 5.50832 5.51104 5.49785 5.50391 5.50001 5.50580 5.51862
##  [163] 5.48837 5.49290 5.50756 5.50737 5.51101 5.49070 5.51095 5.50364 5.51226
##  [172] 5.48589 5.49591 5.48435 5.48793 5.49998 5.49810 5.49720 5.50870 5.49707
##  [181] 5.51280 5.50229 5.48806 5.51292 5.49435 5.50349 5.48819 5.49743 5.50641
##  [190] 5.49700 5.49625 5.50463 5.51452 5.50080 5.51776 5.50507 5.50285 5.50614
##  [199] 5.50552 5.49370 5.50801 5.49644 5.50192 5.49154 5.51353 5.50764 5.48819
##  [208] 5.48345 5.50199 5.51266 5.50519 5.49450 5.51916 5.50578 5.50298 5.48630
##  [217] 5.49798 5.50117 5.49852 5.48886 5.50580 5.51761 5.50190 5.49580 5.49776
##  [226] 5.49105 5.51619 5.49908 5.49798 5.49726 5.48501 5.49440 5.49414 5.50845
##  [235] 5.50704 5.50419 5.50960 5.49744 5.51134 5.49072 5.51226 5.49665 5.48997
##  [244] 5.50703 5.49178 5.50037 5.48009 5.50259 5.53126 5.50889 5.49540 5.49767
##  [253] 5.49298 5.49119 5.50515 5.49922 5.50797 5.49373 5.50466 5.46957 5.51359
##  [262] 5.48453 5.50197 5.49947 5.51665 5.49335 5.51763 5.49112 5.49942 5.50183
##  [271] 5.50876 5.49668 5.49491 5.50470 5.51197 5.51943 5.49938 5.50269 5.50262
##  [280] 5.49941 5.48969 5.49865 5.50097 5.48689 5.48736 5.49456 5.50712 5.51390
##  [289] 5.50687 5.50860 5.48404 5.50243 5.49542 5.50583 5.50178 5.50622 5.48991
##  [298] 5.49929 5.50383 5.49073 5.50628 5.50178 5.49764 5.51884 5.49677 5.49423
##  [307] 5.49343 5.51361 5.48611 5.50173 5.48418 5.50341 5.50166 5.50113 5.51318
##  [316] 5.49773 5.50292 5.49926 5.49530 5.49358 5.49473 5.50954 5.50871 5.49909
##  [325] 5.50097 5.49419 5.50313 5.51273 5.48568 5.51260 5.49001 5.49619 5.50853
##  [334] 5.51046 5.48709 5.50467 5.50866 5.50679 5.49148 5.50428 5.49094 5.51188
##  [343] 5.50404 5.49822 5.51218 5.50671 5.48448 5.49411 5.49964 5.51466 5.49921
##  [352] 5.49321 5.50679 5.49712 5.48904 5.49924 5.50674 5.48850 5.49970 5.50397
##  [361] 5.51782 5.50483 5.49233 5.50499 5.48986 5.51314 5.49958 5.49074 5.49917
##  [370] 5.48761 5.49996 5.49961 5.51676 5.49874 5.49962 5.48495 5.49788 5.48061
##  [379] 5.49240 5.48367 5.48318 5.51377 5.49869 5.49893 5.48916 5.51130 5.50982
##  [388] 5.49684 5.49095 5.50535 5.51020 5.50526 5.48424 5.51150 5.49525 5.48896
##  [397] 5.51657 5.48787 5.49761 5.49953 5.51054 5.50232 5.50572 5.48969 5.49954
##  [406] 5.49944 5.50068 5.51304 5.50016 5.50077 5.49120 5.50059 5.50438 5.49735
##  [415] 5.47650 5.49399 5.50648 5.51693 5.49416 5.49172 5.48631 5.50294 5.50524
##  [424] 5.49343 5.49779 5.49927 5.50441 5.50225 5.50321 5.49516 5.50229 5.50082
##  [433] 5.49514 5.48809 5.50000 5.50060 5.51057 5.49658 5.49785 5.49269 5.50058
##  [442] 5.49841 5.50155 5.51226 5.50328 5.49184 5.48800 5.51811 5.50749 5.50570
##  [451] 5.48873 5.50006 5.50593 5.50013 5.50479 5.49798 5.49296 5.49678 5.49760
##  [460] 5.49805 5.49593 5.49518 5.50585 5.48709 5.50256 5.49971 5.51109 5.51270
##  [469] 5.50018 5.50773 5.47976 5.49977 5.49045 5.51893 5.50200 5.50205 5.51041
##  [478] 5.50472 5.49079 5.47196 5.49074 5.51853 5.49193 5.48833 5.50420 5.49946
##  [487] 5.49945 5.50679 5.49010 5.49756 5.48700 5.51355 5.51461 5.49430 5.51089
```

```
## [496] 5.49543 5.50367 5.50924 5.51202 5.50439 5.50860 5.49912 5.49932 5.51752
## [505] 5.48438 5.49180 5.48975 5.51214 5.49319 5.50884 5.50904 5.48727 5.51037
## [514] 5.50524 5.50744 5.50918 5.48743 5.49786 5.50511 5.49349 5.48985 5.50641
## [523] 5.50342 5.49637 5.50836 5.50258 5.48597 5.50697 5.50538 5.49986 5.50256
## [532] 5.49451 5.49432 5.49983 5.48654 5.50463 5.47240 5.50132 5.50657 5.49586
## [541] 5.48624 5.51235 5.49316 5.50631 5.50124 5.49307 5.48379 5.50063 5.48521
## [550] 5.48690 5.48507 5.48507 5.48490 5.49728 5.50605 5.50911 5.50442 5.49177
## [559] 5.48415 5.47924 5.48995 5.50237 5.49921 5.47829 5.49911 5.49481 5.50222
## [568] 5.50665 5.49260 5.50700 5.50372 5.50459 5.50752 5.49781 5.48874 5.50159
## [577] 5.49367 5.49005 5.50778 5.50932 5.50016 5.50766 5.49887 5.49210 5.49802
## [586] 5.50094 5.49307 5.48789 5.50564 5.49584 5.50530 5.49040 5.50033 5.47495
## [595] 5.51241 5.50639 5.50585 5.48855 5.50298 5.51053 5.50474 5.49951 5.50612
## [604] 5.49408 5.50656 5.50791 5.49987 5.50462 5.49516 5.49504 5.49862 5.49667
## [613] 5.51654 5.49370 5.49372 5.50018 5.49167 5.49514 5.48592 5.50095 5.47882
## [622] 5.49203 5.49659 5.49257 5.51279 5.49988 5.49166 5.50501 5.49682 5.49688
## [631] 5.49512 5.50200 5.47752 5.51204 5.50285 5.51786 5.49250 5.51202 5.47769
## [640] 5.50647 5.49623 5.50153 5.49741 5.51103 5.49566 5.50271 5.49808 5.49874
## [649] 5.48762 5.51230 5.50479 5.50703 5.51074 5.49195 5.49579 5.50177 5.50624
## [658] 5.48646 5.52193 5.49569 5.49540 5.50692 5.51192 5.50883 5.49094 5.51663
## [667] 5.49912 5.48226 5.49606 5.48455 5.48582 5.49905 5.49563 5.48751 5.50433
## [676] 5.49400 5.49886 5.50410 5.51160 5.50584 5.49676 5.49783 5.49228 5.51488
## [685] 5.49998 5.49730 5.50530 5.49755 5.50974 5.49502 5.51053 5.48746 5.49675
## [694] 5.49933 5.50430 5.50447 5.50266 5.49553 5.51106 5.50413 5.50123 5.48561
## [703] 5.49255 5.49888 5.51044 5.49770 5.50499 5.50116 5.48881 5.49867 5.48948
## [712] 5.50543 5.50936 5.48597 5.49772 5.50543 5.49061 5.48882 5.50901 5.50492
## [721] 5.48534 5.50072 5.51141 5.50846 5.48892 5.50409 5.50788 5.49615 5.50857
## [730] 5.49485 5.48836 5.50668 5.50921 5.49101 5.48762 5.50033 5.49285 5.51418
## [739] 5.49377 5.50267 5.48646 5.49312 5.50713 5.50963 5.49931 5.49398 5.50028
## [748] 5.49418 5.49765 5.49779 5.49899 5.49880 5.50580 5.50012 5.49458 5.50186
## [757] 5.49357 5.49999 5.51330 5.52821 5.51218 5.49938 5.49032 5.49624 5.52020
## [766] 5.47446 5.50760 5.48835 5.49280 5.49215 5.50284 5.49613 5.49519 5.50466
## [775] 5.48548 5.49144 5.48632 5.50659 5.48819 5.50244 5.50294 5.48341 5.48417
## [784] 5.50358 5.50294 5.49577 5.51184 5.50114 5.50454 5.51459 5.50000 5.48752
## [793] 5.50729 5.49348 5.49370 5.49396 5.49207 5.49153 5.51022 5.48449 5.48234
## [802] 5.50194 5.50507 5.49364 5.49446 5.49785 5.49155 5.50156 5.49752 5.50805
## [811] 5.51357 5.50507 5.49747 5.51376 5.50971 5.49444 5.48705 5.49716 5.48724
## [820] 5.50780 5.50351 5.49105 5.50622 5.50525 5.51433 5.51506 5.50189 5.49805
## [829] 5.49485 5.51992 5.49602 5.49665 5.50501 5.50986 5.49498 5.51076 5.47495
## [838] 5.49865 5.49548 5.50420 5.49889 5.51307 5.49040 5.50059 5.48405 5.50999
## [847] 5.50685 5.50472 5.52420 5.49274 5.50145 5.51182 5.49689 5.49268 5.49412
## [856] 5.49995 5.49444 5.49427 5.49246 5.49253 5.49854 5.49446 5.49074 5.51859
## [865] 5.50765 5.50781 5.49765 5.51536 5.51881 5.48993 5.50266 5.50133 5.50457
## [874] 5.50417 5.48210 5.50383 5.50356 5.50233 5.50707 5.49825 5.50157 5.49668
## [883] 5.50498 5.50201 5.50739 5.50964 5.50004 5.50181 5.52160 5.49416 5.49606
## [892] 5.50013 5.50046 5.49626 5.48588 5.50796 5.48895 5.50389 5.50306 5.49803
## [901] 5.48907 5.50859 5.49231 5.51024 5.50556 5.50162 5.50701 5.49760 5.51048
## [910] 5.49240 5.49691 5.50315 5.49767 5.50023 5.50289 5.50124 5.48354 5.49636
## [919] 5.49673 5.50204 5.47578 5.49530 5.50514 5.50297 5.51278 5.50287 5.49413
## [928] 5.49430 5.50542 5.50165 5.48943 5.49744 5.48867 5.50992 5.48924 5.49095
## [937] 5.48405 5.51309 5.49510 5.48961 5.47885 5.50446 5.51571 5.50316 5.51266
## [946] 5.50687 5.50258 5.50127 5.49277 5.49012 5.49510 5.50329 5.50389 5.49395
## [955] 5.49270 5.50190 5.49598 5.50538 5.49595 5.49838 5.50787 5.51125 5.50968
## [964] 5.49332 5.50376 5.50177 5.51040 5.49770 5.52082 5.51301 5.48756 5.49821
## [973] 5.50683 5.50796 5.49985 5.50261 5.52459 5.48210 5.49559 5.50985 5.50083
```

```
## [982] 5.49762 5.49440 5.48690 5.49239 5.49497 5.51449 5.50674 5.48948 5.48816
## [991] 5.49419 5.49661 5.49654 5.50681 5.49732 5.49600 5.50649 5.49396 5.50013
## [1000] 5.50743
```

```
stopCluster(cl)
```

Unfortunately, sometimes parallel processes can take longer than serial! This is because cpu to cpu communication takes time, so not all processes will work faster in parallel.

Another Example

```
sapply(1:100,function(x) x^3)
```

```
##   [1]       1       8      27      64     125     216     343     512     729
##  [10]    1000    1331    1728    2197    2744    3375    4096    4913    5832
##  [19]    6859    8000    9261   10648   12167   13824   15625   17576   19683
##  [28]   21952   24389   27000   29791   32768   35937   39304   42875   46656
##  [37]   50653   54872   59319   64000   68921   74088   79507   85184   91125
##  [46]   97336  103823  110592  117649  125000  132651  140608  148877  157464
##  [55]  166375  175616  185193  195112  205379  216000  226981  238328  250047
##  [64]  262144  274625  287496  300763  314432  328509  343000  357911  373248
##  [73]  389017  405224  421875  438976  456533  474552  493039  512000  531441
##  [82]  551368  571787  592704  614125  636056  658503  681472  704969  729000
##  [91]  753571  778688  804357  830584  857375  884736  912673  941192  970299
## [100] 1000000
```

```
cl = makeCluster(4)
parSapply(cl = cl, X = 1:100,FUN = function(x) x^3 )
```

```
##   [1]       1       8      27      64     125     216     343     512     729
##  [10]    1000    1331    1728    2197    2744    3375    4096    4913    5832
##  [19]    6859    8000    9261   10648   12167   13824   15625   17576   19683
##  [28]   21952   24389   27000   29791   32768   35937   39304   42875   46656
##  [37]   50653   54872   59319   64000   68921   74088   79507   85184   91125
##  [46]   97336  103823  110592  117649  125000  132651  140608  148877  157464
##  [55]  166375  175616  185193  195112  205379  216000  226981  238328  250047
##  [64]  262144  274625  287496  300763  314432  328509  343000  357911  373248
##  [73]  389017  405224  421875  438976  456533  474552  493039  512000  531441
##  [82]  551368  571787  592704  614125  636056  658503  681472  704969  729000
##  [91]  753571  778688  804357  830584  857375  884736  912673  941192  970299
## [100] 1000000
```

```
stopCluster(cl)

# for external functions, one has to pass functions using clusterExport
cl = makeCluster(4)
cube = function(x){
  return(x^3)
}
clusterExport(cl,'cube')
parSapply(cl = cl, X = 1:100,FUN = cube )
```

```
##   [1]       1       8      27      64     125     216     343     512     729
##  [10]    1000    1331    1728    2197    2744    3375    4096    4913    5832
##  [19]    6859    8000    9261   10648   12167   13824   15625   17576   19683
##  [28]   21952   24389   27000   29791   32768   35937   39304   42875   46656
##  [37]   50653   54872   59319   64000   68921   74088   79507   85184   91125
##  [46]   97336  103823  110592  117649  125000  132651  140608  148877  157464
```

```
##  [55]  166375  175616  185193  195112  205379  216000  226981  238328  250047
##  [64]  262144  274625  287496  300763  314432  328509  343000  357911  373248
##  [73]  389017  405224  421875  438976  456533  474552  493039  512000  531441
##  [82]  551368  571787  592704  614125  636056  658503  681472  704969  729000
##  [91]  753571  778688  804357  830584  857375  884736  912673  941192  970299
## [100] 1000000
```

```
stopCluster(cl)
```

Process on Database

R users are comfortable with manipulating and analyzing data in an R environment. However, for large databases or data that changes frequently, downloading the data into R may not be efficient or even feasible. One approach to addressing this issue is to move some computation to the database.

Processing data in-database can be achieved by:

- Computation with SQL. Packages that provides a database interface include `RPostgreSQL`, `RMySQL`, and `ROracle`.
- For those who would rather work with R syntax, there are packages that can translate R syntax into SQL statements that are then executed on the database. These include `dplyr` and `PivotalR`.
- Running advanced computation in the database using database specific algorithms or open source projects like MADlib.
- Using columnar databases (e.g., MonetDB)for improved performance.
- Using array databases (e.g., SciDB)for maximum scientific computing performance.

Big Data

One of the solutions to analyzing large datasets is to use a distributed computing environment such as Hadoop. Apache Spark, a part of the Hadoop ecosystem, works particularly well for machine learning problems.

- Apache Spark is a fast and general engine for large-scale data processing
- Multi-stage in-memory primitives provides performance up to 100 times faster for certain applications
- Allows user programs to load data into a cluster's memory and query it repeatedly
- Well-suited to machine learning
- All the major cloud platforms - AWS, Google Cloud, Microsoft Azure - will rent and run a cluster. R packages for using Spark include `SparkR` and `sparklyr`.