R and High Performance Computing

SciNet's Virtual Summer Training Program 2021

Marcelo Ponce

August 23-27, 2021

Some details about the course

For getting credit for this 3-days course:

- You need to attend 2 out of 3 sessions
- For demonstrating that, please
  - take the attendance test each day
  - enter the corresponding attendance code for each day
- Submit the assignment
- Slides and all material, including attendance test, recordings, forum, etc is available in the course website,
  `https://scinet.courses/583`
- 3 sessions of 1.5 hs each

Acknowledgment

This course is based on material developed by Dr. Jonathan Dursi (SickKids) and Dr. Erik Spence (SciNet).

Topics to cover

We will discuss the following topics:

- Memory management in R.
- Out-of-core computation.
- Profiling.
- Compiling R code.
- The 'parallel' package.
- mc-parallel/collect/apply.
- foreach and doparallel.
- Rdsm and pbdR.

The material in this class is not introductory, a working knowledge of R is assumed.
Please ask questions if you don't understand something, including after the course is over:

courses@scinet.utoronto.ca

or use the forum in the course website.

Getting set up on the cluster

Please perform the following steps to get yourself setup for today's class, but use your own username.

Your username most likely would be something like `lcl_uot2021ssXXXX`

```
myuser@mycomp ~>
myuser@mycomp ~> ssh lcl_uot2021ssXXXX@teach.scinet.utoronto.ca -X
scinetguestXXX@teach01 ~>
scinetguestXXX@teach01 ~> cd $SCRATCH
scinetguestXXX@teach01 >  pwd
/scratch/t/tempacct/lcl_uot2021ssXXXX
scinetguestXXX@teach01 >
scinetguestXXX@teach01 >  cp -r /scinet/course/ss2020/9_hpcr/parallelR .
scinetguestXXX@teach01 >
```

This will copy the code and data you need to your $SCRATCH directory.

Getting set up on the cluster, continued

```
scinetguestXXX@teach01 >  debugjob
debugjob:  Requesting 1 nodes with 1 tasks for 240 minutes and 0 seconds
SALLOC: Granted job allocation 89089
SALLOC: Waiting for resource configuration
SALLOC: Nodes teach36 are ready for job
scinetguestXXX@teach36 ~>
```

```
scinetguestXXX@teach36 ~> cd $SCRATCH/parallelR
```

```
scinetguestXXX@teach36 ~>
```

```
scinetguestXXX@teach36 parallelR> pwd
/scratch/t/tmpacct/lcl_uot2021ssXXXX/parallelR
```

```
scinetguestXXX@teach36 parallelR>
```

```
scinetguestXXX@teach36 parallelR> ls
code data pbd setup
```

```
scinetguestXXX@teach36 parallelR> source setup
```

```
scinetguestXXX@teach36 parallelR>
```

```
scinetguestXXX@teach36 parallelR>  R
>
```

It should only take a moment to get your compute node.

High-performance R

Just a reminder:

- R is an interpreted language. As such, there is an extra layer of infrastructure (the interpreter) needed to make R run.
- As a general rule, because of the extra layer of infrastructure, interpreted languages (R, Python, Bash, Perl, ...) are not high-performance languages.
- True high-performance languages are compiled, because they lack this extra layer of infrastructure: C, C++, Fortran.
- That being said, there are ways of making things not quite so bad. That is the goal of this class.

High Performance Computing (HPC)

- Moore's Law for processors
- Exponentially growing datasets
- Led to increasing compute time
- Bottom line: use HPC/ Super-Computing Infrastructure
- Basic Approach: *Divide and conquer* – divide code into smaller chunks/ parallelize to run multiple functions simultaneously
- Then communicate and bam! You have your output faster than before.

Basics

- Split the problem into pieces
- Execute the pieces in parallel
- Combine the results back together

How does R do this?

- Several packages make it easier!
- For single node (no inter node communication): Multicore and doMC
- For multi node (with internode communication): foreach, parallel, doMC, doSNOW

Most computational problems can be grouped in

- Memory-bound: memory is the limiting aspect in the computation, eg. datasets too large to fit in memory
- Computer-bound: the actual computation is the element that us being done most of the time, your code spents most of its time using the CPU
- Communication-bound: communication between different elements of the code, and in particular among different "nodes" is the bottle neck
- File-IO: too much reading/writting/accesing information from files

Most computational problems can be grouped in

- Memory-bound: memory is the limiting aspect in the computation, eg. datasets too large to fit in memory
- Computer-bound: the actual computation is the element that us being done most of the time, your code spents most of its time using the CPU
- Communication-bound: communication between different elements of the code, and in particular among different "nodes" is the bottle neck
- File-IO: too much reading/writting/accesing information from files

Dealing with some of these will require to rethink some aspects and implementations (algorithms) in your code.
Others can be mitigated by using some useful packages.

R and memory

One must be cognisant of how R manages memory:

- R is "pass by value" if the variables being passed are being modified. As such, R frequently needs to make temporary copies of variables, and hitting the memory limit of your machine can be a frequent problem.
- Like many dynamic languages, R relies on "garbage collection" to limit its memory usage.
- In a running code, "every so often" a garbage collection task runs and deletes variables that won't be used any more.
- You can force the garbage collector to run at any given time by calling gc(), but this almost never fixes anything significant.
- How can GC know that you're not going to use that big variable in the next line? The garbage collector needs your help to be effective.

Useful memory-management commands

- gc(verbose = TRUE), or just gc(TRUE)
  - Calling gc(TRUE) alone probably won't help anything, but it does give verbose output, returning memory usage as a matrix.
- ls()
  - Lists all existing variables, as strings.
- object.size(variablename)
  - Pass it a variable, and it prints out its size.
  - Pass it get("variablename") and it will also print its size.
- rm(variablename)
  - Deletes a variable you no longer need. Lets gc go to work.
- Fun little one-liner which prints out all variables by size in bytes:

```
> sort(sapply(ls(), function(x) {object.size(get(x))}), decreasing = TRUE)
```

object.size() and gc()

Let's play with object.size() and gc():

```
> gc()
            used   (Mb)   gc trigger   (Mb)   max used   (Mb)
 Ncells   183250    9.8       407500   21.8     350000   18.7
 Vcells   377223    2.9       905753    7.0     864975    6.6
```

```
> old.mem <- gc()[, 1:2]
```

```
> x <- rep(0., (16 * 1024)**2)
```

```
> xsize <- object.size(x)
```

```
> xsize
2147483688 bytes
```

```
> print(xsize, units = "MB")
2048 Mb
```

```
> new.mem <- gc()[, 1:2]
```

```
> new.mem - old.mem
              used   (Mb)
 Ncells        445      0
 Vcells   268436139   2048
```

object.size() and gc(), some more

Now let's delete the object and see how system memory behaves:

```
> rm(x)
>
> final.mem <- gc()[, 1:2]
>
> final.mem - old.mem
         used   (Mb)
 Ncells   451    0.1
 Vcells  1781    0.0
>
```

Use 'rm' in your scripts whenever you are done with a large variable.

Memory-bound/Out-of-core computation

Some problems require doing fairly simple analysis on data that is too large to fit into memory

- Min/mean/max.
- Data cleaning.
- Even linear fitting is pretty simple.

In this case, one processor may be enough; you just want a way to not run out of memory.

"Out of core" or "external memory" computation leaves the data on disk, bringing into memory only what is needed, or what fits, at any given time.

For some computations, this works out well (but note: disk access is always much slower than memory access).

Out-of-core computation

The "bigmemory" package defines a generalization of a matrix class, big.matrix, which can be "file-backed". That is, can exist primarily on disk, with parts being brought into memory as necessary.

This approach works when one's data access involves passing through the entire data set once or a very small number of times, either combining data or extracting a subset.

The packages "bigalgebra", "biganalytics" and "bigtabulate" are built on bigmemory.

Ideal gas data set

In data/idealgas, we have a set of synthetic data files describing an ideal gas experiment - setting temperature, amount of material, and volume, and measuring pressure.

Simple data sets:

```
> small.data <- read.csv("data/idealgas/ideal-gas-fixedT-small.csv")
> small.data[1:2,]
     X    pres          vol     n   temp
 1   1   99000   0.02036345   0.8    300
 2   2   99250   0.02018306   0.8    300
```

Row name, pressure (Pa), volume (m^3), N (moles), and temperature (K).

A larger data set consisting of 124M rows, 4.7 GB, is sitting in ideal-gas-fixedT-large.csv, and we'd like to do some analysis of this data set. But the size is a problem.

Creating a file-backed big matrix

We've already created a big.matrix file from this data set, using

```
> # DO *NOT* RUN THIS!
> library(bigmemory)
> data <- read.big.matrix("data/idealgas/ideal-gas-fixedT-large.csv",
+ header = FALSE, backingfile = "data/idealgas/ideal-gas-fixedT-large.bin",
+ descriptorfile = "ideal-gas-fixedT-large.desc")
>
```

This reads in the .csv file and outputs a binary equivalent (the "backingfile") and a descriptor (in the "descriptorfile") which contains all of the information which describes the binary blob.

I've already created the descriptor file, "ideal-gas-fixedT-large.desc". since the conversion takes 12 minutes for this data set . . .

Note: this converts the data into a matrix, which is a less flexible data type than a data frame; homogeneous type. Here, we'll use all numeric.

Using a big.matrix

Let's load the data set and see how memory behaves.

```
> library(bigmemory, quiet = TRUE)
>
> orig.gc <- gc()[, 1:2]
> data <- attach.big.matrix("data/idealgas/ideal-gas-fixedT-large.desc")
>
> new.gc <- gc()[, 1:2]
> new.gc - orig.gc
           used   (Mb)
 Ncells    4975    0.6
 Vcells   18256    0.1
>
```

Using a big.matrix, continued

```
> data[1:2,]
              pres          vol     n   temp
 [1,]    1   90000.0   0.01328657   0.5    280
 [2,]    2   90012.5   0.01285503   0.5    280
>
> system.time(min.p <- min(data[,"pres"]))
   user   system   elapsed
  9.004    2.096    11.175
>
> gc()[, 1:2] - orig.gc
          used   (Mb)
 Ncells   3077    0.2
 Vcells   3484    0.1
> min.p
[1] 90000
>
```

That only took about 11 seconds to scan through 124M entries.

Summary: bigmemory

If you just have a data file much larger than memory that you have to crunch and the amount of actual computation is not a bottleneck, the 'bigmemory' and related packages may be all you need.

It works best if:

- the data is of homogeneous type - e.g. all integer, all numeric, all string.
- you just need to work on a subset of data at a time, or,
- you just need to make one or two passes through the data to complete analysis.

Profiling

To push your code to new heights of awesome, or to make it useful at all (depending on your situation), you will need to profile your code. What is profiling?

- Profiling is analyzing where the code is spending its time. Which parts of the code are slowest?
- Testing how long individual functions take can be performed with the 'microbenchmark' package, or more crudely, 'system.time'.
- To test the whole program we use 'Rprof'.

We'll do some examples of each.

Profiling individual functions

The 'system.time' command uses the OS's 'time' command to determine how long the code takes to run.

The "microbenchmark" function is more systematic. It takes an average over 100 calls of the function. Consequently, it can take a while to run.

Note that the microbenchmark package will need to be downloaded.

```
> f <- function() {
+   a <- 1
+   for (i in 1:1e6) {
+     a <- a + i
+   }
+ }
>
> system.time(f())
   user  system elapsed
  0.433   0.004   0.437
>
> library(microbenchmark)
>
> microbenchmark(f())
Unit: milliseconds
  expr    min     mean      max
   f()  427.0071 432.8328 482.3085
>
```

Profiling whole programs

```
> addme <- function(a, b) return(a + b)
```

```
> test <- function() {  a <- 1
+   for (i in 1:1e6) a <- addme(a, i)
+ }
```

```
>
```

```
> Rprof("Rprof.data")
```

```
> test()
```

```
> Rprof(NULL)
```

```
>
```

```
> s <- summaryRprof("Rprof.data")
```

```
> names(s)
[1] "by.self"   "by.total"   "sample.interval"   "sample.time"
```

```
> s$by.total
           total.time   total.pct   self.time   self.pct
  "test"         1.32      100.00        0.46      34.85
 "addme"         0.82       62.12        0.76      57.58
     "+"         0.06        4.55        0.06       4.55
     ":"         0.04        3.03        0.04       3.03
```

Rprof

Some notes about the last slide:

- Rprof samples the program every 20ms, by default, to see where the program is spending its time.
- Use "Rprof('filename')" to store the Rprof results in a particular file.
- Use "Rprof(NULL) to turn off profiling.
- You can read 'filename' if you want. It's easier to just use "summaryRprof('filename')" to analyse the results.
- Results are given in data frames.
- total.time and total.pct include all time spent within a function, including calls to other functions.
- self.time and self.pct indicate actually real time spent in each function (self.pct should add up to 100%, give or take rounding).

Compiled code

It is possible to interface your R code with compiled code. Why would you want to do that?

- It's fast! Compiled code is always faster than interpreted code.
- If you can get the slowest parts of your code into a compiled language, you can leave the rest in R.
- R comes with the ability to byte-compile specific functions.
- It's also possible to write your own pure C++ or Fortran code to interface with R, but it's a pain.
- It's easier to use the Rcpp package, written by Dirk Eddelbuettel, Romain Francois, and others.
- This package allows you to easily interface with C++ code.

Byte-compiled R code

We can byte-compile specific R functions using the "compiler" package.

The "microbenchmark" function can be used to benchmark the performance of functions.

Here we're using the "enableJIT" (Just In Time compiler) function to turn off automatic byte compiling. In general, you should NOT do this. We're only doing this for the purposes of comparing speeds.

```
> library(compiler)
> library(microbenchmark)
> oldJIT <- enableJIT(0)
>
> f <- function(n) { x <- 1
+   for (i in 1:n) x <- 1 / (1 + x)
+ }
>
> lf <- cmpfun(f)
> n <- 1e5
>
> microbenchmark(f(n), lf(n))
Unit:  milliseconds
   expr | min   | mean  | max
  f(n)  | 15.88 | 17.40 | 32.17
 lf(n)  | 2.35  | 2.38  | 2.54
>
```

Byte-compiled R code, continued

Some notes about the last slide:

- Byte compiling is not the same as actually compiling code, as is done with compiled languages:
  - Byte compiling creates a byte object, which is executed by a virtual machine.
  - Compiled languages are compiled into machine code, which is directly used by the hardware.
- Nonetheless, byte compiling can be significantly faster than running the code through the R interpreter.
- If you run a function multiple times, R will automatically byte-compile it for you. Better to just byte-compile it in your utilities file.
- Automatic byte compiling can be turned off using the "enableJIT" function, though this is not recommended.
- The microbenchmark package is used for benchmarking. It runs the same code 100 times by default, and gets statistics.

Installing Rcpp

We're going to be doing examples with Rcpp. But, if you're using Windows...

- Rcpp is not a default R package; you will need to download and install it.
- Because Rcpp compiles code (that's the point), you will need a compiler on your computer.
- If you're using Linux or a Mac, you're probably ok.
- On Windows, you need to go here, and download "Rtools":

`https://cran.r-project.org/bin/windows/Rtools`

Note that Rtools is quite large, and will require some time to download.

Using Rcpp

Once the function is defined, it will automatically be compiled, this is why it takes a moment for the "cppFunction" command to finish.

Once compiled, Rcpp creates an R function which links to the compiled C++ code.

```
>
> library(Rcpp)
>
> cppFunction("int times(int x, int y) {
+   int product = x * y;
+   return product;
+ }")
>
> times(34, 4)
[1] 136
>
> 34 * 4
[1] 136
>
```

Using Rcpp, continued

Some notes about his example.

- Defining the functions "inline", as we have here, is difficult if the function is large and complex. We will deal with this problem on the next slide.
- Rcpp defines special C++ data types which are compatible with R data types:
  - IntegerVector, NumericVector, LogicalVector, CharacterVector.
  - IntegerMatrix, NumericMatrix, LogicalMatrix, CharacterMatrix.
  - Lists, DataFrames.
- These data types allow the ability to deal with missing values, using the is_na() function.

Note that you should always test your code carefully when using multiple languages. Sometimes surprises can creep in.

Using Rcpp, predefined functions

Some notes:

- Comments start with "//" in C++.
- The "sourceCpp" command causes the C++ code to be compiled.
- "// [[Rcpp::export]]" must be placed before each function you wish to export to R.
- If you get error messages about the "R.h" file, you may need to download the R development packages for your machine.

```cpp
// MyRcppCode.cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int times(int x, int y) {
  // This function returns the
  // product of the two input
  // arguments.
  int product = x * y;
  return product;
}
```

```
> library(Rcpp)
> sourceCpp("code/MyRcppCode.cpp")
> times(10, 3)
[1] 30
>
```

Using Rcpp, why bother?

```cpp
// MyRcppCode2.cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double mysum(NumericVector x,
  NumericVector y) {
  // Returns the sum of the
  // product of two vectors.

  int n = x.size();
  double answer = 0.0;

  for(int i = 0; i < n; i++) {
    answer += x[i] * y[i];
  }
  return answer;
}
```

```
>
> library(Rcpp)
>
> sourceCpp("code/MyRcppCode2.cpp")
>
> library(microbenchmark)
>
> x <- runif(1e6)
> y <- runif(1e6)
>
> microbenchmark(mysum(x, y), sum(x * y))
Unit:  milliseconds
   expr         min       mean      max
 mysum(x, y) 1.777204  1.832208  1.97020
 sum(x * y)  4.974938  8.248512 44.27162
>
```

Using Rcpp with vectors

```cpp
// MyRcppCode3.cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector doubleAddy(
  NumericVector x, double y) {
  // Doubles the vector x
  // and adds y.

  int n = x.size();
  NumericVector answer(n);

  for(int i = 0; i < n; i++) {
    answer[i] = 2 * x[i] + y;
  }
  return answer;
}
```

```
>
> library(Rcpp)
>
> sourceCpp("code/MyRcppCode3.cpp")
>
> doubleAddy(1:7, 3.4)
[1] 5.4 7.4 9.4 11.4 13.4 15.4 17.4
>
```

Making your code awesome

Some tips:

- Save your function profiling until you know that the function works correctly. Don't succumb to "premature profiling".
- Do byte-compiling first. It's easy and may be good enough.
- Don't be afraid of Rcpp. Once you know how to program in one language, you're at least 80% of the way to programming in all languages.
- Ask us for help, if speed becomes an issue for your productivity.

Scalable data analysis in R

One turns to parallel computing to solve one of two problems:

- My program is too slow. Perhaps using more processors will make things faster:
  - Your program is compute bound.
  - Tools to use: parallel/multicore, Rdsm.
- My program crashes due to lack of memory. Perhaps splitting the problem up into smaller pieces will allow it to run.
  - Your program is memory bound.
  - Tools to use: parallel/snow, pbdR.

Note what is not on this list:

- My program constantly reads from, and write to, thousands of files, and these operations are very slow.

These I/O-bound problems are not easily solved with parallelism (adding more processors or nodes doesn't usually help).

Using multiple processors in R

The rest of this class we will cover using multiple processors and/or nodes to do large-scale computations in R.

- no-work parallelism: existing packages.
- "parallel" package:
  - "multicore" (use all cores on a computer): non-windows.
  - "snow" (use all cores on a computer, or across a cluster).
- "foreach" package: different interface to similar functionality.
- "Rdsm": shared-memory parallelism (on-node) with big.matrix.
- "pbdR": massive-scale computation with MPI + R.

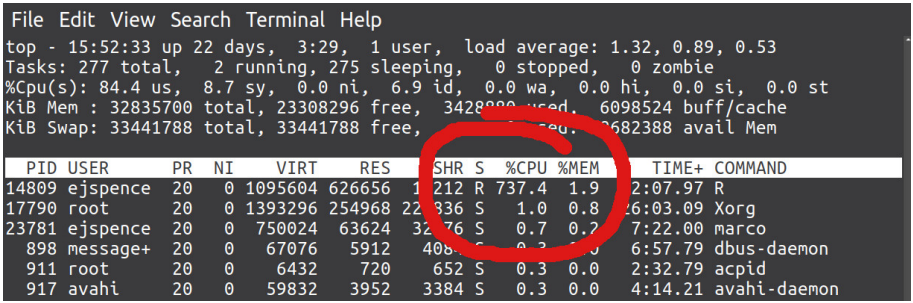
Existing parallelism

It's important to realize that many fundamental routines as well as higher-level packages come with some degree of scalability and parallelism "baked in".

Open another terminal to your node, and run "top" while executing the following in R:

```
>
> n <- 4 * 1024
>
> A <- matrix( rnorm(n * n), ncol = n, nrow = n )
> B <- matrix( rnorm(n * n), ncol = n, nrow = n )
>
> C <- A %*% B
>
```

Existing parallelism, continued



R can (and should) be built using high-performance threaded libraries for math in general, and linear algebra in particular.

Here the single R process has launched several threads of execution – all of which are part of the same process, and so can see the same memory.

Packages that explicitly use parallelism

For a complete list, see

`http://cran.r-project.org/web/views/HighPerformanceComputing.html`

- Biopara
- BiocParallel for Bioconductor
- bigrf - Random Forests
- caret - cross-validation, bootstrap characterization of predictive models
- GAMBoost - boosting glms

Plus packages that use linear algebra or other expensive math operations which can be implicitly multithreaded.

When at all possible, don't do the hard work yourself — look to see if a package already exists which will do your analysis at scale.

The parallel Package

Since R 2.14.0 (late 2011), the "parallel"' package has been part of core R. It incorporates - and mostly supersedes - two other packages:

- "multicore": for using all cores on a single processor. Not on Windows.
- "snow": for using any group of processors, possibly across a cluster.

Many packages which use parallelism use one of these two, so it is worth understanding.

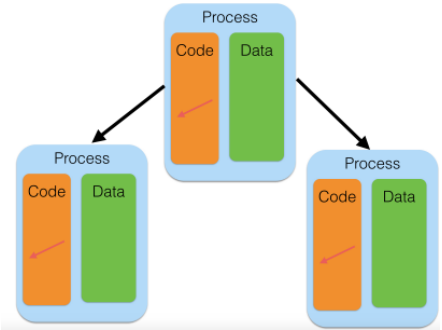
Both create new processes (not threads) to run on different processors; but differ in important ways.

Multicore - forking

Multicore creates new processes by forking — cloning – the original process.

That means the new processes start off seeing a copy of exactly the same data as the original. If a first process can read a file, and it then forks two new processes - each will see a copy of the file.

These are not shared memory; changes in one process will not be reflected in others.



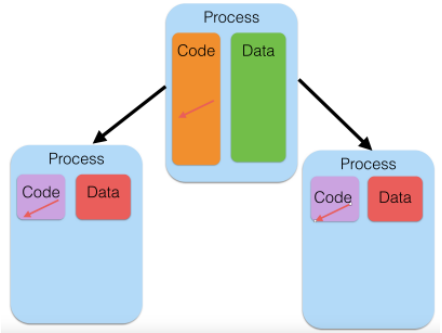
Windows doesn't have fork(), so windows can't use these routines.

Snow - Spawning

In contrast, Snow creates entirely new R processes to run the jobs.

A downside is that you need to explicitly copy over any needed data and functions.

But the upsides are that spawning a new process can be done on a remote machine, not just current machine. So you can, in principle, use entire clusters.



In addition, the flipside of the downside: new processes don't have any unneeded data - less total memory footprint.

mcparallel/mccollect

The simplest use of the "multicore" package is the pair of functions "mcparallel" and "mccollect":

- mcparallel() forks a task to run a given function; it then runs in the background.
- mccollect() waits for and gets the result.

Let's pick an example: reading the airlines data set, we want — for a particular month — to know both the total number of planes in the data (by tail number) and the median elapsed flight time. These are two independent calculations, and so can be done independently.

mcparallel/mccollect, continued

We start two tasks with mcparallel, and collect the answers with mccollect:

```
> library(parallel, quiet=TRUE)
> source("data/airline/read_airline.R")
> jan2010 <- read.airline("data/airline/airOT201001.csv")
> unique.planes <- mcparallel( length( unique( sort(jan2010$TAIL_NUM) )))
> median.elapsed <- mcparallel(median( jan2010$ACTUAL_ELAPSED_TIME,
+ na.rm = TRUE ))
> ans <- mccollect( list(unique.planes, median.elapsed) )
> ans
$'30113'
[1] 4555

$'31286'
[1] 110
```

We get a list of answers, with each element "named" by the process ID that ran the job.
There are 4555 planes in the data set, with a mean flight time of 110 minutes.

mcparallel/mccollect, continued more

Does this save any time? Let's do some independent fits to the data. Let's try to see what the average in-flight speed is by fitting time in the air to distance flown; and let's see how the arrival delay correlates with the departure delay. (Do planes, on average, make up some time in the air, or do delays compound?)

```
>
> system.time(fit1 <- lm(DISTANCE ~ AIR_TIME, data=jan2010))
   user  system elapsed
  1.071   0.009   0.976
> system.time(fit2 <- lm(ARR_DELAY ~ DEP_DELAY, data=jan2010))
   user  system elapsed
  0.659   0.005   0.524
>
```

Total time: about 1.5 seconds.

mcparallel/mccollect, continued even more

So the time to beat is about 1.5s:

```
> parfits <- function() {
+ pfit1 <- mcparallel(lm(DISTANCE ~ AIR_TIME, data=jan2010))
+ pfit2 <- mcparallel(lm(ARR_DELAY ~ DEP_DELAY, data=jan2010))
+ mccollect( list(pfit1, pfit2) )
}
> system.time( parfits() )
   user  system elapsed
  0.620   0.089   1.685
```

We don't see a savings of time: 1.7s vs 1.5s. Clearly actually forking the processes and waiting for them to rejoin itself takes some time.

This overhead means that we want to launch jobs that take a significant length of time to run - much longer than the overhead (hundredths to tenths of seconds for fork().)

Clustering

Typically we want to do more than an itemized list of independent tasks - we have a list of similar tasks we want to perform.

'mclapply' is the multicore equivalent of 'lapply' - apply a function to a list, get a list back.

Let's say we want to see what similarities there are between delays at O'Hare airport in Chicago in 2010. Clustering methods attempt to uncover "similar" rows in a data set by finding points that are near each other in some p-dimensional space, where p is the number of columns.

k-Means is a particularly simple, randomized, method; it picks k cluster centre-points at random, finds the rows closest to them, assigns them to the cluster, then moves the cluster centres towards the centre of mass of their cluster, and repeats.

The quality of the result depends on the number of random trials.

Clustering, continued

Let's try that with our subset of data. Run this:

```
> load('data/airline/ord.delays.Rdata')
```

which does this:

```
> air2010 <- read.csv("data/airline/airOT20101.csv")
> delaycols <- c(18, 28, 40:44)      # columns listing various delay measures
> ord.delays <- air2010[air2010$ORIGIN == "ORD", delaycols]
> rm(air2010)
> ord.delays <- ord.delays[ord.delays$ARR_DELAY_NEW > 0,]
> ord.delays <- ord.delays[complete.cases(ord.delays),]
```

```
> system.time(serial.res <- kmeans(ord.delays, centers = 2, nstart = 40))
   user  system elapsed
  1.219   0.026   1.248
> serial.res$betweenss
[1] 236714813
```

Clustering with lapply

Running 40 random trials is the same as running 10 random trials 4 times. Let's try that approach with "lapply":

```
> do.n.kmeans <- function(n) {kmeans(ord.delays, centers = 2, nstart = n) }
> system.time(list.res <- lapply(rep(10, 4), do.n.kmeans))
   user  system elapsed
  1.845   0.002   1.848
> res <- sapply(list.res, function(x) return(x$tot.withinss))
> lapply.res <- list.res[[which.min(res)]]
> lapply.res$withinss
[1] 205574263 117857364
> lapply.res$betweenss
[1] 236714813
```

Get the same answer, but it took longer - bit of overhead from splitting it up and starting the process four times. We could make the overhead less important by using more trials, which would be better anyway.

Clustering with mclapply

"mclapply" works the same way as lapply, but forking off the processes (as with "mcparallel")

```
> system.time(list.res <- mclapply(rep(10,4), do.n.kmeans, mc.cores = 4))
   user  system elapsed
  1.858   0.205   2.064
>
> res <- sapply(list.res, function(x) x$tot.withinss)
>
> mclapply.res <- list.res[[which.min(res)]]
>
> mclapply.res$betweenss
[1] 236714813
>
```

Clustering with mclapply, continued

Note what the output of top looks like when this is running:

```
> system.time(list.res<- mclapply(rep(10,4), do.n.kmeans, mc.cores=4))
   user  system elapsed
  1.837   0.192   2.040
>
```

```
top - 13:50:08 up 23 days,  4:47,  0 users,  load average: 0.29, 0.13, 0.08
Tasks: 316 total,   5 running, 311 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3.2 us,  0.4 sy,  0.0 ni, 96.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 13200635+total, 11230968+free,  2558216 used, 17138444 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 12688553+avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
14023 mponce    20   0  782280 436168   1108 R  13.6  0.3   0:00.41 R
14024 mponce    20   0  782280 436168   1108 R  13.3  0.3   0:00.40 R
14025 mponce    20   0  782280 436168   1108 R  13.3  0.3   0:00.40 R
14026 mponce    20   0  782280 436168   1108 R  13.3  0.3   0:00.40 R
13289 mponce    20   0  782280 442576   7516 S   2.3  0.3   0:15.83 R
13959 mponce    20   0  172372   2484   1624 R   0.3  0.0   0:00.13 top
```

There are four separate processes running - not one process using multiple CPUs via threads.

Clustering with mclapply

Looks good! Let's take a look at the list of results:

```
>
> res
[1] 323431626 323431626 323431626 323431626
>
```

What happened here?

Parallel RNG

Depending on what you are doing, it may be very important to have different (or the same!) random numbers generated in each process. Here, we definitely want them different - the whole point is to generate different random realizations.

"parallel" has a good RNG suitable for parallel work based on the work of Pierre L'Ecuyer in Montréal:

```
> RNGkind("L'Ecuyer-CMRG")
> mclapply( rep(1,4), rnorm, mc.cores = 3, mc.set.seed=TRUE)
[[1]]
[1] 1.113293

[[2]]
[1] 1.258494

[[3]]
[1] 0.7554586
```

Load balancing

Suppose, instead of running multiple random trials to find the best, given a set of clusters, we were unsure of how many clusters we wanted to run:

```
> do.kmeans.nclusters <- function(n) {
+ kmeans(ord.delays, centers = n, nstart = 10)}
```

```
> time.it <- function(n) { system.time( res <- do.kmeans.nclusters(n)) }
```

```
> lapply(1:4, time.it)
[[1]]
   user  system elapsed
  0.238   0.000   0.238
[[2]]
   user  system elapsed
  0.443   0.001   0.451
[[3]]
   user  system elapsed
  0.954   0.000   0.955
[[4]]
   user  system elapsed
  1.562   0.001   1.572
```

Load balancing, continued

More clusters takes longer. If we were to mclapply these four tasks on 2 CPUs, the first CPU would get the two short tasks, and the second CPU would get the second, longer tasks - bad "load balance".

Normally, we want to hand multiple tasks of work off to each processor and only hear back when they're completely done - minimal overhead. But that works best when all tasks have similar lengths of time.

If you don't know that this is true, you can do dynamic scheduling - give each processor one task, and when they're done they can ask for another task.

More overhead, but better distribution of work.

Load balancing, continued more

```
>
> system.time(res <- mclapply(1:4, time.it, mc.cores = 2) )
   user   system   elapsed
  1.355    0.072     3.415
>
> system.time(res <- mclapply(1:4, time.it, mc.cores = 2,
+ mc.preschedule = FALSE))
   user   system   elapsed
  1.859    0.174     3.482
>
```

Summary: parallel/multicore

The ‘mc*‘ routines in parallel work particularly well when:

- You want to make full use of the processors on a single computer
- Each task only reads from some big common data structure and produces modest-sized results

Things to watch for:

- Modifying the big common data structure:
  - Won’t be seen by other processes,
  - But will blow up the memory requirements
- Won’t work on Windows (but what does?)
- ‘mc.cores‘ is a lie. It’s the number of tasks, not cores. On an 8-core machine, if you have multithreaded libraries and launch something ‘mc.cores=8‘ you’ll end up with 64 threads competing for 8 cores. Either make sure to turn off threading (‘export OMP_NUM_THREADS=1‘), or use fewer tasks.

Multiple computers with parallel/snow

The other half of parallel, routines that were in the still-active ‘snow‘ package, allow you to again launch new R processes — by default, on the current computer, but also on any computer you have access to. (SNOW stands for "Simple Network of Workstations", which was the original use).

The recipe for doing computations with snow looks something like:

```
>
> library(parallel)
> cl <- makeCluster(nworkers,...)
> results1 <- clusterApply(cl, ...)
> results2 <- clusterApply(cl, ...)
> stopCluster(cl)
>
```

Other than the ‘makeCluster()‘/‘stopCluster()‘, it looks very much like multicore and ‘mclapply‘.

Hello world with parallel

Let's try starting up a "cluster" (eg, a set of workers) and generating some random numbers from each:

```
> library(parallel)
> cl <- makeCluster(4)
> clusterCall(cl, rnorm, 5)
[[1]]
[1] -0.19542059 -0.09533088 -0.21122094 -1.52002161 1.24074398

[[2]]
[1] 1.60195084 0.47906454 0.74859881 0.03488538 -0.49270944

[[3]]
[1] 0.3162637 -0.3729758 0.8680270 0.4741110 0.7736880

[[4]]
[1] -0.1799470 -0.7960984 -0.1628196 -0.9641411 1.8729729
> stopCluster(cl)
```

Hello world with parallel, continued

'clusterCall()' runs the same function (here, 'rnorm', with argument '5') on all workers in the cluster. A related helper function is 'clusterEvalQ()' which is handier to use for some setup tasks:

```
> cl <- makeCluster(4)
```

```
> clusterEvalQ(cl, {library(party); print("Hello World!")})
[[1]]
[1] "Hello World"

[[2]]
[1] "Hello World"

[[3]]
[1] "Hello World"

[[4]]
[1] "Hello World"
```

Clustering on clusters

Emboldened by our success so far, let's try re-doing our k-means calculations:

```
> load('data/airline/ord.delays.Rdata')
>
> do.n.kmeans <- function(n) {kmeans(ord.delays, centers = 2, nstart = n) }
>
> library(parallel)
> cl <- makeCluster(4)
>
> res <- clusterApply(cl, rep(5,4), do.n.kmeans)
Error in checkForRemoteErrors(val) :
4 nodes produced errors; first error:  object 'ord.delays' not found
>
> stopCluster(cl)
>
```

Ah! Failure.

Clustering on clusters, continued

Recall that we aren't forking here; we are creating processes from scratch. These processes, new to this world, are not familiar with our ways, customs, or data sets. We actually have to ship the data out to the workers:

```
> cl <- makeCluster(4)
> system.time(clusterExport(cl, "ord.delays"))
   user  system elapsed
  0.193   0.039   0.607
>
                                                        user  system elapsed
> system.time(cares <- clusterApply(cl, rep(5,4), do.n.kmeans))  1.049   0.045  25.650
> stopCluster(cl)
> system.time(mcres <- mclapply(rep(5,4), do.n.kmeans, mc.cores = 4))
   user  system elapsed
  0.379   0.051  24.068
```

Clustering on clusters, continued more

Note that the costs of shipping out data back and forth, and creating the processes from scratch, is relatively costly - but this is the price we pay for being able to spawn the processes anywhere (meaning off node).

(And if our computations take hours to run, we don't really care about several-second delays.)

Note that with makeCluster we are still restricted to a single node.

Running across machines

The default cluster is a sockets-based cluster; you can run on multiple machines by specifying them to a different call to makeCluster:

```
> hosts <- c( rep("localhost",8), rep("teach36", 2) )
> cl <- makePSOCKcluster(names = hosts)
> clusterCall(cl, rnorm, 5)
[[1]]
[1] -0.02141595 0.55431769 -0.64238398 -2.18983521 0.50568289

⋮

[[10]]
[1] -1.434019700 -1.016475875 1.385483544 0.003703908 0.536871928
> stopCluster(cl)
```

Cluster notes

There are too many variations on the makeCluster family of functions to go over today. Here are a few more highlights:

- There is an MPI-based cluster. This is similar to the PSOCK cluster, but startup and communication can be much faster once you start going to large numbers (say >64) of hosts.
- clusterApplyLB: "LB" stands for "Load Balanced". The default 'clusterApply' sends off one task to each worker, waits until they're both done, then sends off another. clusterApplyLB fires off tasks to each worker as needed (like "mc.preschedule = FALSE" for mclapply).
- clusterSplit: use this function to split up a data set across your cluster.
- parLapply: use this to chunk up the data, and send all the data to all the tasks at once.

Summary: parallel

The ‘cluster‘ routines in ‘parallel‘ are good if you know you will eventually have to move to using multiple computers (nodes in a cluster, or desktops in a lab) for a single computation.

- Use ‘clusterExport‘ for functions and data that will be needed by everyone.
- Communicating data is slow, but much faster than having every worker read the same data from a file.
- Use clusterApplyLB if the tasks vary greatly in runtime.
- Use clusterApply if each task requires an enormous amount of data.
- Use makePSOCKcluster for small clusters; consider makeMPIcluster for larger (but see ‘pbdR‘ section).

foreach and doparallel

The "master/slave" approach that 'parallel' enables works extremely well for moderately sized problems, and isn't that difficult to use. It is all based on one form of R iteration, apply, which is well understood.

However, going from serial to parallel requires some re-writing, and even going from one method of parallelism to another (eg, 'multicore'-style to 'snow'-style) requires some modification of code.

The 'foreach' package is based on another style of iterating through data - a for loop - and is designed so that one can go from serial to several forms of parallel relatively easily. There are then a number of tools one can use in the library to improve performance.

foreach - serial

The foreach operator looks similar to the standard for loop, but returns a list of the iterations:

The foreach function creates an object, and the ‘%do%‘ operator operates on the code (here just one statement, but it can be multiple lines between braces, as with a for loop) and the foreach object.

```
>
> for (i in 1:3) print(sqrt(i))
[1] 1
[1] 1.414214
[1] 1.732051
>
> library(foreach)
> foreach (i = 1:3) %do% sqrt(i)
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051
>
```

foreach + doParallel

Foreach works with a variety of backends to distribute computation - ‘doParallel‘, which allows snow- and multicore-style parallelism, and ‘doMPI‘ (not covered here).

Switching the previous loop to parallel just requires registering a backend and using ‘%dopar%‘ rather than ‘%do%‘:

```
> library(doParallel)
>
> # use multicore-style forking
> registerDoParallel(3)
>
> foreach (i = 1:3) %dopar% sqrt(i)
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

> stopImplicitCluster()
>
```

foreach + doParallel, continued

One can also use a PSOCK cluster:

```
>
> cl <- makePSOCKcluster(3)
> registerDoParallel(cl) # use the just-created PSOCK cluster
>
> foreach (i = 1:3) %dopar% sqrt(i)
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

> stopCluster(cl)
>
```

Combining results

While returning a list is the default, 'foreach' has a number of ways to combine the individual results:

```
> foreach (i = 1:3, .combine = c) %do% sqrt(i)
[1] 1.000000 1.414214 1.732051
```

```
> foreach (i = 1:3, .combine = cbind) %do% sqrt(i)
        result.1   result.2   result.3
  [1,]          1   1.414214   1.732051
```

```
> foreach (i = 1:3, .combine = "+") %do% sqrt(i)
[1] 4.146264
```

```
> foreach (i = 1:3, .multicombine = TRUE, .combine = "sum") %do% sqrt(i)
[1] 4.146264
```

By default, foreach will combine each new item individually. If ".multicombine = TRUE", then you are saying that you're passing a function which will do the right thing even if foreach gives it a whole wack of new results as a list or vector - e.g., a whole chunk at a time.

Combining foreach objects

There's one more operator: '%:%'. This lets you nest foreach objects:

```
>
> foreach (i = 1:3, .combine = "c") %:%
+ foreach (j = 1:3, .combine = "c") %do% {
+ i * j
}
[1] 1 2 3 2 4 6 3 6 9
>
```

And you can also filter items, using "when":

```
>
> foreach (a = rnorm(25), .combine = "c") %:%
+ when (a >= 0) %do%
+ sqrt(a)
[1] 0.5265719 0.2187333 0.1730294 0.9077089 0.2466300 1.1946766 1.1086728
>
```

foreach iterators

Another problem that one can quickly run into: we often create a large vector to loop over (1:1000000 for example) which in general is the same size as the data set. For large data sets this can mean big memory.

One can use the iterators package to get a loop variable without creating something the size of the object. For instance, icount() is like the difference between Python 2.x range and xrange:

```
>
> library(iterators)
>
> foreach (i = icount(3), .combine = 'c') %do% sqrt(i)
[1] 1.000000 1.414214 1.732051
>
```

isplit

If we want each task to only work on a subset of the data, the 'isplit' iterator will split the data, and send off the partitioned data to workers:

```
> jan2010 <- read.csv('data/airline/airOT201001.csv')
>
> ans <- foreach (byAirline = isplit(jan2010$DISTANCE,
+ jan2010$UNIQUE_CARRIER), .combine = cbind) %do% {
+ df <- data.frame(sum(byAirline$value));
+ colnames(df) <- byAirline$key;
+ return(df) }
>
> ans$UA
[1] 32036702
>
> ans$OH
[1] 5289850
>
```

Summary: foreach

Foreach is a wrapper for the other parallel methods we've seen, so it inherits some of the advantages and drawbacks of each.

Use 'foreach' if:

- your code already relies on for-style iteration; the transition is easy.
- you don't know if you want multicore vs. snow style 'parallel' use: you can switch just by registering a different backend!
- You want to be able to incrementally improve the performance of your code.

Note that you can have portions of your analysis code use 'foreach' with 'parallel' and portions using the backend with apply-style parallelism; it doesn't have to be all one or the other.

Advanced R: Rdsm, pbdR

We've looked at some of the standard scalable computing packages for R.

Now we're going to look at two somewhat more advanced packages, that solve very different problems.

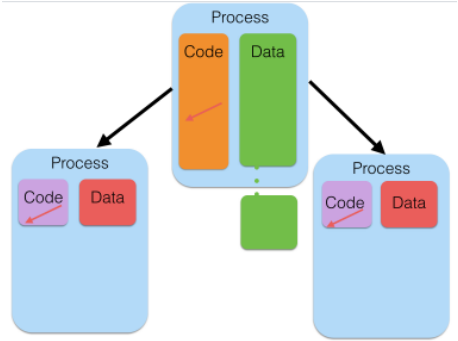
- Rdsm: Get the most (performance, memory) out of a single-computer computation by using *shared memory*.
- pbdR: Get the most (performance, scale) out of a cluster computation by ditching master-slave, and using very large-scale distributed routines.

Rdsm

While it's generally true that processes can't peer into each other's memory, there is an exception.

Processes can explicitly make a window of memory shared - visible to other processes.

This isn't necessary for threads within a process; but it is necessary for multiple processes working on the same data.



The only works on-node; you can't share memory across a network.

Rdsm, continued

Some notes about the motivation for Rdsm:

- Rdsm allows you to share a matrix across processes on a node - for reading and for writing.
- Normally when we split a data structure up across tasks we make copies (PSOCK), or we use read-only (multicore/fork).
- If the output is also going to be large, we now have 2-3 copies of the data structure floating around.
- Rdsm allows (on-node) cluster tasks to collaboratively make a large output without making copies.
- Rdsm: *R distributed shared memory*

Rdsm, continued more

Simple example - let's create a shared matrix, and have everyone fill it.

- Create a PSOCK cluster
- Create an Rdsm instance
- Create a shared matrix
- Create a barrier.

Make sure you're somewhere in your $SCRATCH directory.

```
>
> library(parallel)
> library(Rdsm)
>
> nrows <- 7
>
> # form a 3-process PSOCK cluster
> cl <- makePSOCKcluster(3)
>
> # initialize Rdsm
> init <- mgrinit(cl)
>
> # make a 7x7 shared matrix
> mgrmakevar(cl, "m", nrows, nrows)
>
> bar <- makebarr(cl)
>
```

Rdsm, continued some more

Each process gets its own id, and each is assigned its own rows of the matrix.

```
> # at each thread, set id to Rdsm built-in ID variable for that thread
> clusterEvalQ(cl, myid <- myinfo$id)
[[1]]
[1] 1
[[2]]
[1] 2
[[3]]
[1] 3
> clusterExport(cl, c("nrows"))
>
> # The line below breaks up the data by rows.
> dmy <- clusterEvalQ(cl, myidxs <- getidxs(nrows))
> dmy <- clusterEvalQ(cl, m[myidxs,1:nrows] <- myid)
> dmy <- clusterEvalQ(cl, "barr()")
>
```

Each process fills its rows with its id.

Rdsm, continued even more

Now, print the results.

```
>
> print(m[,])
      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
 [1,]    1    1    1    1    1    1    1
 [2,]    1    1    1    1    1    1    1
 [3,]    2    2    2    2    2    2    2
 [4,]    2    2    2    2    2    2    2
 [5,]    2    2    2    2    2    2    2
 [6,]    3    3    3    3    3    3    3
 [7,]    3    3    3    3    3    3    3
>
> stoprdsm(cl)  # stops cluster
>
```

Summary: Rdsm

Your takeaway for Rdsm:

- Rdsm allows collaborative use of a single pool of memory.
- It avoids performance and memory problems of making copies to send back and forth.
- It works well when:
  - Outputs are as large/larger than inputs. (Correlation matrix of stocks).
  - Inputs are very large, and want to do transformation in-place (values to log-returns).
- But remember that it will only work on a single node.

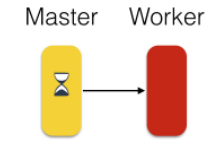
pbdR : Programming with Big Data in R

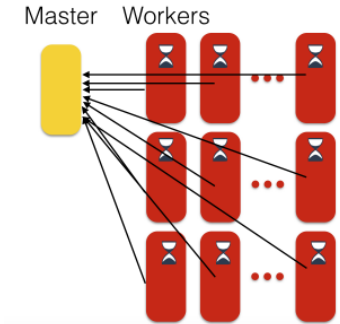
The master-worker approach works well for interactive work, is easy to load balance, and is easy to understand.

But there's a narrow range of number of workers where master-worker works well. For a small number of total processors (2-4), it hurts to have one processor doing nothing except some small amount of coordination.

For a large number of processors (hundreds or more, depending on the size of each task), the workers can overwhelm the master, with all the workers waiting while the master catches up.

`https://pbdr.org`



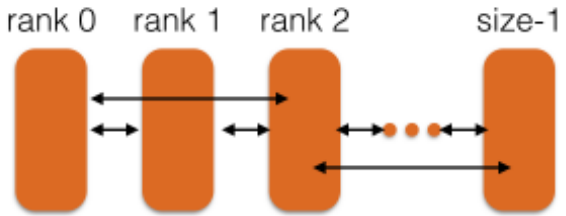


pbdR, continued

At scale, the idea of a single master isn't helpful. It's better to coordinate between peers.

Rather than a single master parcelling out work, the workers themselves decide which part of the problem they should be working on, and combine their results cooperatively. This is more efficient and can scale better, but there are downsides:

- Dynamic load-balancing is substantially trickier (but doable).
- Can't really do this interactively; need to write a script.



Departure hour histogram example

In ‘pbd/mpi-histogram.R‘ we have a script that does an hour-histogram calculation for eight full years of airline data, sifting through 40 million flights, in about a minute:

We’ll use the jupyter hub, `https://njupyter.scinet.utoronto.ca`, for running this! You will need to install the following packages `float, pbdMPI`.

```
scinetguestXXX@teach36 ~> cd $SCRATCH/parallelR/pbd
scinetguestXXX@teach36 pbd>
scinetguestXXX@teach36 pbd> time mpirun -np 8 Rscript mpi-histogram.R
COMM.RANK = 0
   [1]      4081    118767     27633      7194      9141    194613   2235007
   [8]   2902703   3003510   2649823   2373934   2473105   2757256   2772498
  [15]   2362334   2485699   2503423   2794298   2626931   2282125   2074739
  [22]   1386485    649392    344257
[1] 41038948
 real   1m15.357s
 user   9m39.943s
 sys    0m10.910s
scinetguestXXX@teach36 pbd>
```

COMM.RANK = 0

Departure hour histogram example, cont

```
scinetguestXXX@teach36 pbd> cat mpi-histogram.R
library(pbdMPI, quiet = TRUE)
⋮
# count.hours and get.hour definitions...
start.year <- 1990
init()
rank <- comm.rank()
my.year <- start.year + rank

myfile <- paste0("data/airline/airOT", as.character(my.year),".RDS")
data <- readRDS(myfile)
data <- data$DEP_TIME
myhrs <- count.hours(data)

hrs <- allreduce( myhrs, op = "sum" )
comm.print( hrs )
comm.print( sum(hrs) )

finalize()
```

Departure hour histogram example, cont

Let's look at the first few lines:

```
myuser@mycomp ~> cat mpi-histogram.R
⋮
# count.hours and get.hour definitions...
start.year <- 1990
init()
rank <- comm.rank()
my.year <- start.year + rank

myfile <- paste0("data/airline/airOT", as.character(my.year),".RDS")
data <- readRDS(myfile)
data <- data$DEP_TIME
⋮
```

Each task decides which year's data to work on. First (zeroth) task works on 1990, next on 1991, etc. Every task has to call the 'init()' routine when starting, and 'finalize()' routine when done.

Departure hour histogram example, cont

```
⋮
myhrs <- count.hours(data)
hrs <- allreduce( myhrs, op = "sum" )
comm.print( hrs )
comm.print( sum(hrs) )
finalize()
```

```
myuser@mycomp ~>
```

Once the file is read, we use the count.hours routine to work on the entire vector.

Then an ‘allreduce‘ function sums each workers hours, and returns the sum to all processors. We then print it out.

Rather than only the master running the main program and handing off bits to workers, every task runs this identical program; the only difference is the value of ‘comm.rank()‘.

Summary: pbdR

To remember for pbdR:

- Allows data to be parallelized across a number of workers.
- Coordination between the workers is automatic. No need to do the array-index bookkeeping that is usually required.
- Comes with alot of MPI-like functionality build in.

pbdR comes with a tonne of functionality, which we won't be covering:

- MPI-like reductions.
- Parallel *apply functions.
- Specialized data distributions, ddmatrix.
- Built-in function that work on ddmatrices, lm, solve, chol.

If you think your data analysis might need to be parallelized in this manner, you should check it out.