Parallel R for Data Science

Erik Spence

SciNet HPC Consortium

12 July 2016

Material for this class

All the material for the HPC Summer School can be found here:

`https://wiki.scinet.utoronto.ca/wiki/index.php/2016_Ontario_Summer_School_for_High_Performance_Computing_Central`

The slides for this class can be found here:

`http://tinyurl.com/ss2016-R2`

and at the SciNet education website:

`http://support.scinet.utoronto.ca/education`

Getting set up on SciNet

Please perform the following steps to get yourself setup for today's class.

```
ejspence@mycomp ~>
ejspence@mycomp ~> ssh ejspence@login.scinet.utoronto.ca -X
ejspence@scinet01-ib0 ~>
ejspence@scinet01-ib0 ~> ssh -X gpc03
ejspence@gpc-f103n084-ib0 ~>
ejspence@gpc-f103n084-ib0 ~> type the command below

            qsub -l nodes=1:ppn=8,walltime=4:00:00 -X -q teach -I
⋮
ejspence@gpc-f108n045-ib0 ~>
```

It should only take a moment to get your compute node. Raise your hand if it takes more than a minute.

Getting setup on SciNet, continued

You now have your own compute node on SciNet. This is where you will run the code for today's class.

```
ejspence@gpc-f108n045-ib0 ~>
ejspence@gpc-f108n045-ib0 ~> pwd
/home/s/scinet/ejspence
ejspence@gpc-f108n045-ib0 ~> cd /scinet/course/ss2016/R
ejspence@gpc-f108n045-ib0 R>
ejspence@gpc-f108n045-ib0 R> pwd
/scinet/course/ss2016/R
ejspence@gpc-f108n045-ib0 R>
ejspence@gpc-f108n045-ib0 R> ls
code data pbd setup
ejspence@gpc-f108n045-ib0 R> source setup
ejspence@gpc-f108n045-ib0 R>
ejspence@gpc-f108n045-ib0 R>  R
>
```

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).

SciNet compute • calcul CANADA

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.

SciNet compute • calcul CANADA

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)
```

SciNet compute • calcul CANADA

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()[, c(1:2, 5:6)]
```

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

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

```
> xsize
2147483688 bytes
```

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

```
> new.mem <- gc()[, c(1:2, 5:6)]
```

```
> new.mem - old.mem
               used   (Mb)    max used     (Mb)
  Ncells        445      0           0      0.0
  Vcells   268436139   2048   268080411   2045.3
```

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

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

```
> rm(x)
>
> final.mem <- gc()[, c(1:2, 5:6)]
>
> final.mem - old.mem
         used   (Mb)     max used      (Mb)
Ncells    451    0.1            0       0.0
Vcells   1781    0.0    268080411    2045.4
>
```

It's better to use functions

Be sure to rm() any temporary intermediate variables.

```
> trunc.gc <- function() {gc()[, c(1:2, 5:6)]}
> orig.gc <- trunc.gc()
> x <- rnorm(16 * 1024 * 1024)
> s <- sum(x)
> s
[1] -1851.947
> rm(x)
> after.gc <- trunc.gc()
> after.gc - orig.gc
         used   (Mb)   max used   (Mb)
 Ncells     35      0          0      0
 Vcells    362      0          0      0
>
```

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.

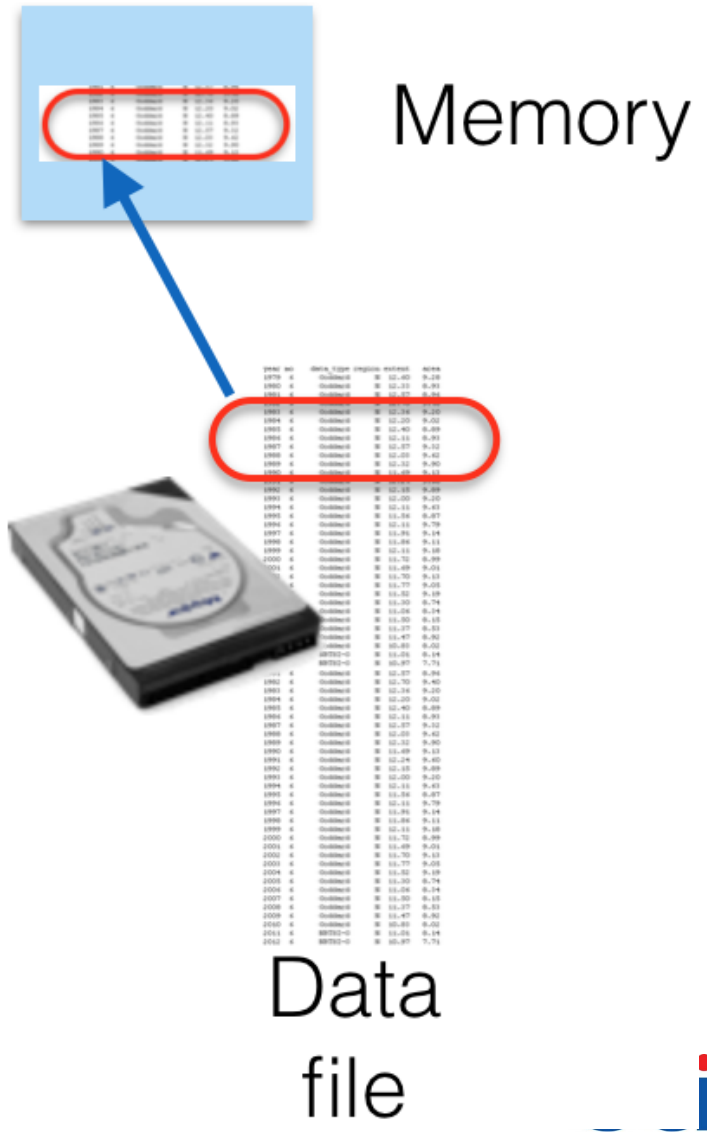
For some computations, this works out well (but note: disk access is always much slower than memory access).

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 fairly well 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.

Packages like "bigalgebra" or "biganalytics" build 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

```
> # Don't run this!
> data <- read.big.matrix("data/idealgas/ideal-gas-fixedT-large.csv",
+ header = TRUE, 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.

You can read the descriptorfile. At the command line type "more ideal-gas-fixedT-large.desc". We've done this for you since the conversion takes 12 minutes for this data set - kind of boring.

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 <- trunc.gc()
> data <- attach.big.matrix("data/idealgas/ideal-gas-fixedT-large.desc")
>
> new.gc <- trunc.gc()
> new.gc - orig.gc
           used   (Mb)   max used   (Mb)
 Ncells    4975    0.6          0      0
 Vcells   18256    0.1          0      0
>
```

Using a big.matrix, continued

Let's do some simple analysis on the data set and see how memory behaves.

```
> 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
 16.407    2.049    18.515
```

```
>
```

```
> trunc.gc() - orig.gc
         used   (Mb)   max used   (Mb)
 Ncells   3077    0.2          0      0
 Vcells   3484    0.1          0      0
```

```
>
```

Using a big.matrix, continued more

That only took about 18 seconds to scan through 124M records to find a minimum. Let's try a few other calculations:

```
> min.p
[1] 90000
>
> system.time(max.p <- max(data[,"pres"]))
   user  system elapsed
 16.300   0.328  16.640
>
> system.time(mean.t <- mean(data[,"temp"]))
   user  system elapsed
 16.587   2.046  18.690
>
```

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.

Works best if:

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

Using multiple processors in R

The rest of today 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

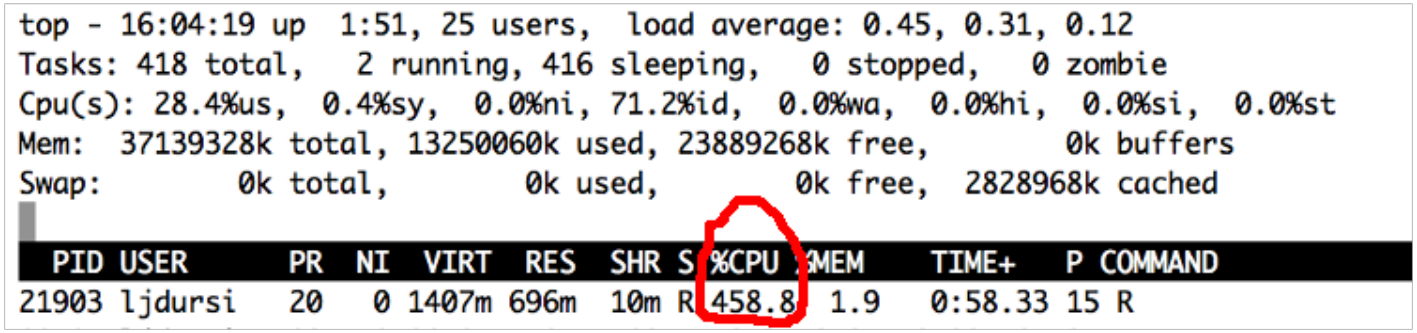
It's important to realize that many fundamental routines as well as higher-level packages come with some degree of scalability and parallelism "baked in".

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
>
```

SciNet compute • calcul CANADA

Existing parallelism, continued



One R process using 458% of a processor.

R can be built using high performance threaded libraries for math in general, and linear algebra — which underlies many data analysis algorithms — 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, eg the large matrices.

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.

SciNet compute • calcul CANADA

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.

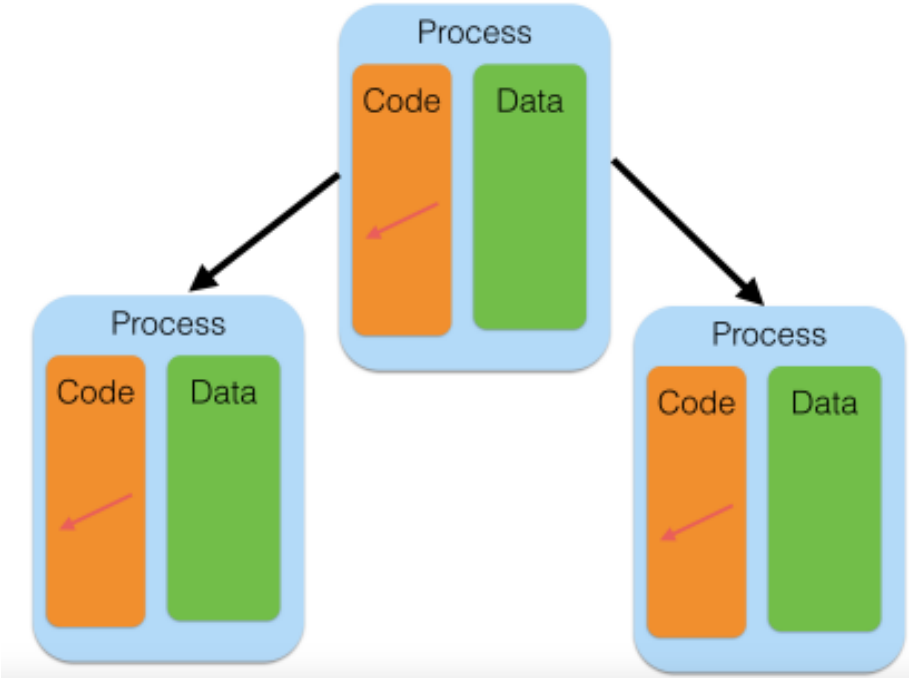
Both create new processes (not threads) to run on different processors; but differ in important ways.

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 copy of the file.

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



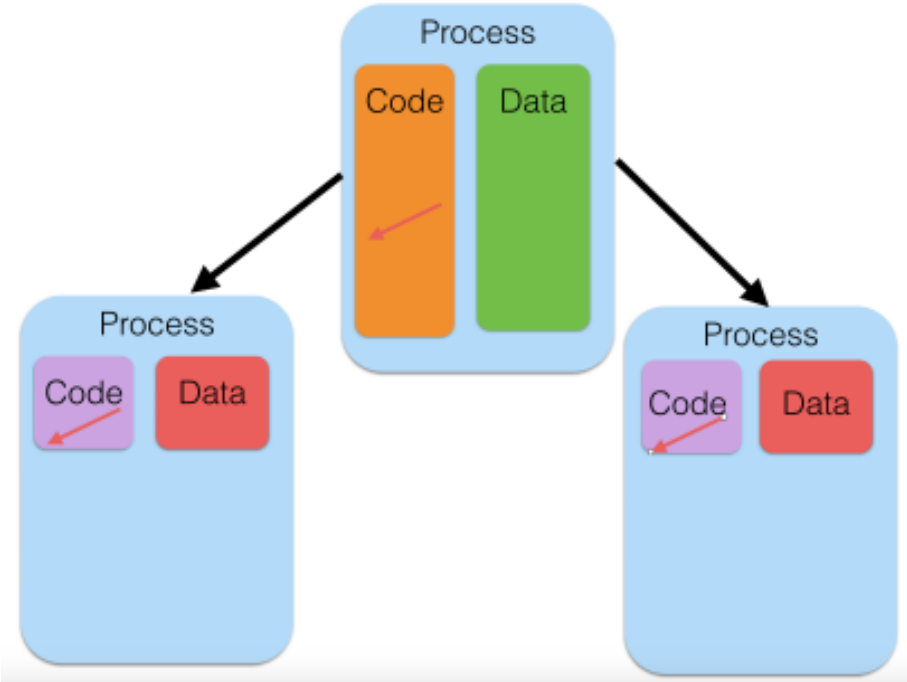
Windows doesn't have fork(), so windows can't use these routines.

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
>
```

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 dataset 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.

Quality of result depends on number of random trials.

Clustering, continued

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

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

Or this:

```
> load("data/airline/airOT2010.Rdata")
> 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
  3.761   0.045   3.809
> 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
  8.212   0.000   8.219
```

```
> res <- sapply(list.res, function(x) 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
  0.068    0.017     2.138
```

```
>
```

```
> res <- sapply(list.res, function(x) x$tot.withinss)
```

```
>
```

```
> mclapply.res <- list.res[[which.min(res)]]
```

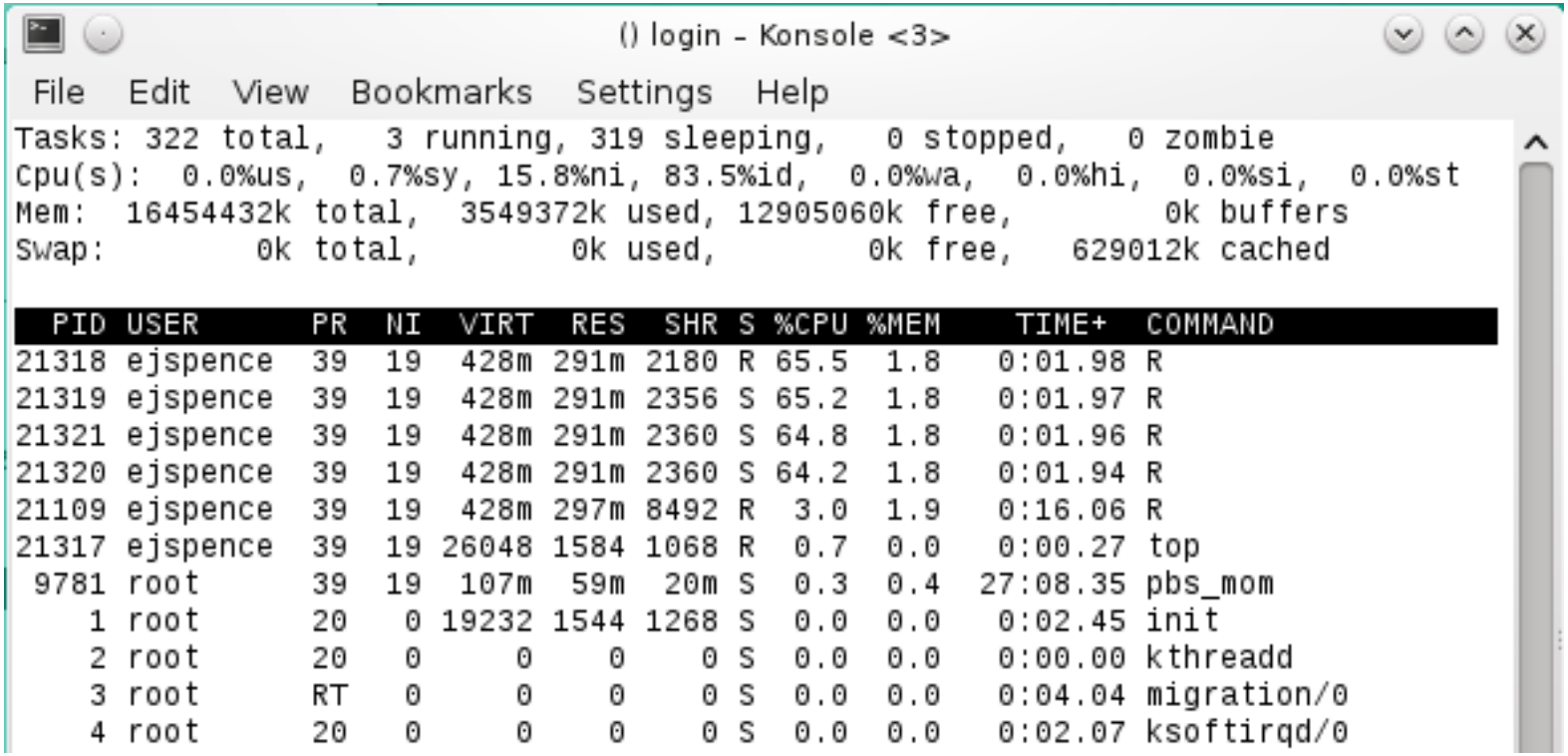
```
>
```

```
> mclapply.res$betweenss
[1] 236714813
```

```
>
```

Clustering with mclapply, continued

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



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

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.

SciNet compute • calcul CANADA

parallel/multicore hands-on

Using the entire 2010 dataset, and the examples above, examine one of the following questions:

- In 2010, what airport (with more than say 10 outgoing flights) had the largest fraction of outgoing flights delayed?
- For some given airport - what hour of the day had the highest relative fraction of delayed flights?
- For all airports?
- What is the effect of including the ‘split()‘ and the ‘Reduce()‘ on the serial-vs-parallel timings for this histogram? Is there a better way of doing the splitting?

SciNet compute • calcul CANADA

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:

```
> delaycols <- c(18, 28, 40:44)
> source("data/airline/read_airline.R")
> jan2010 <- read.airline("data/airline/airOT201001.csv")
> jan2010 <- jan2010[,delaycols]
> jan2010 <- jan2010[complete.cases(jan2010),]
> do.n.kmeans <- function(n) {kmeans(jan2010, centers = 4, 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 'jan2010' 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 datasets. We actually have to ship the data out to the workers:

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

```
> system.time(clusterExport(cl, "jan2010"))
   user  system elapsed
  0.193   0.039   0.607
```

```
>
```

```
> system.time(cares <- clusterApply(cl, rep(5,4), do.n.kmeans))
   user  system elapsed
  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("gpc01", 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)
```

For this to work, you will have to (temporarily) tack the line "source /scinet/course/ss2016/R/setup" to the bottom of your ".bashrc" file.

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 dataset across your cluster.
- parLapply: use this to chunk up the data, and send all the data to all the tasks at once.

SciNet compute • calcul CANADA

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 this afternoon).

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    result3
 [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 some subset of the data, the 'isplit' iterator will split the data at the master, and send off the partitioned data to workers:

```
> ans <- foreach (byAirline = isplit(jan2010$DEP_TIME,
+ jan2010$UNIQUE_CARRIER), .combine = cbind) %do% {
+ df <- data.frame(count.hours(byAirline$value));
+ colnames(df) <- byAirline$key;
+ df }
```

```
> ans$UA
  [1]    2    4    0    0    0  957 1595 1817 2598 1401
 [11] 1713 1774 1509 1907 1442 1230 1510 1888 1775 1311
 [21]  964  783  785  268
```

```
> ans$OH
  [1]   2   2   0   0   0 185 654 469 674 679 572
 [11] 682 843 763 699 671 839 777 507 764 467 186
 [21] 130  20
```

```
>
```

Stock prices example

In 'data/stocks/stocks.csv', we have 419 daily closing stock prices going back to 2000 (3654 prices). For stocks, it's often useful to deal with "log returns", rather than absolute price numbers. We use:

```
>
> stocks <- read.csv("data/stocks//stocks.csv")
> log.returns <- function(values) {
+ nv = length(values)
+ log(values[2:nv]/values[1:nv-1]) }
>
```

How would we parallelize this with 'foreach'? (Imagine we had thousands of stocks and decades of data, which isn't implausable.)

Stock prices example

```
>
> library(doParallel)
> registerDoParallel(4)
>
> mat.log <-
+ foreach(col = iter(stocks[,-c(1,2)], by = "col"),
+ .combine = "cbind") %dopar% log.returns(col)
>
> stopImplicitCluster()
> stocks.log <- as.data.frame(mat.log)
> colnames(stocks.log) <- colnames(stocks)[-c(1,2)]
>
> # get rid of the first day; no "return" for then
> stocks.log$date <- stocks$date[-1]
>
```

Stock correlations

A quantity we might be interested in is the correlation between the log returns of various stocks: we can use R's 'cor()' function to do this.

```
> nstocks <- 419
> cors <- matrix(rep(0,nstocks*nstocks), nrow=nstocks, ncol=nstocks)
> system.time(
+ for (i in 1:419) {
+ for (j in 1:419) {
+ cors[i,j] <- cor(stocks.log[[i]],stocks.log[[j]])
+ }
+ } )
   user  system elapsed
 29.091   0.000  29.130
>
```

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 relys on 'for'-style iteration; 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.

SciNet compute • calcul CANADA

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.

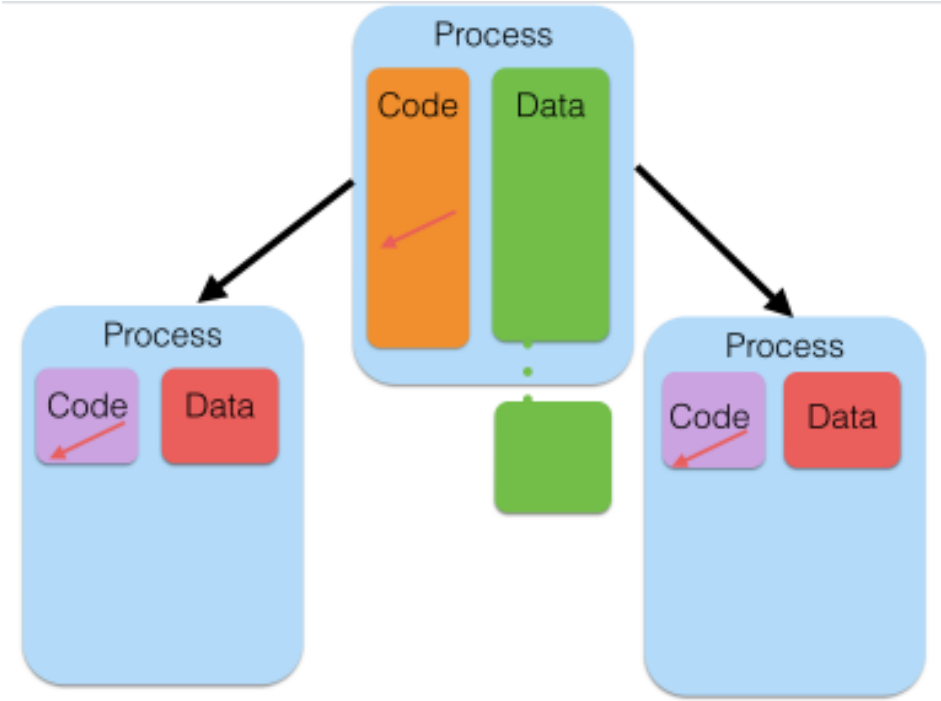
- Rdsm: Get the most (performance, memory) out of a single-computer computation by using shared memory.
- pbdR: Get the most (performance, scale) out of a cluster computation by ditching master-slave, and using very large-scale distributed routines.

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, 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.

```
> setwd(Sys.getenv("SCRATCH"))
> 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 rowsof 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"))
> 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

You 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.

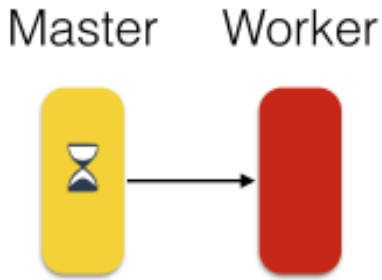
SciNet compute • calcul CANADA

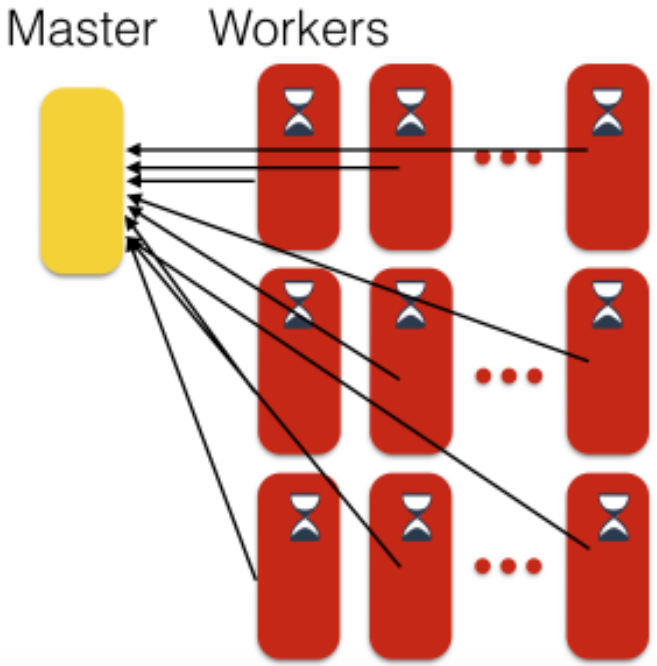
pbdR

The master-worker approach works well for interactive work, is easy to loadbalance, 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.



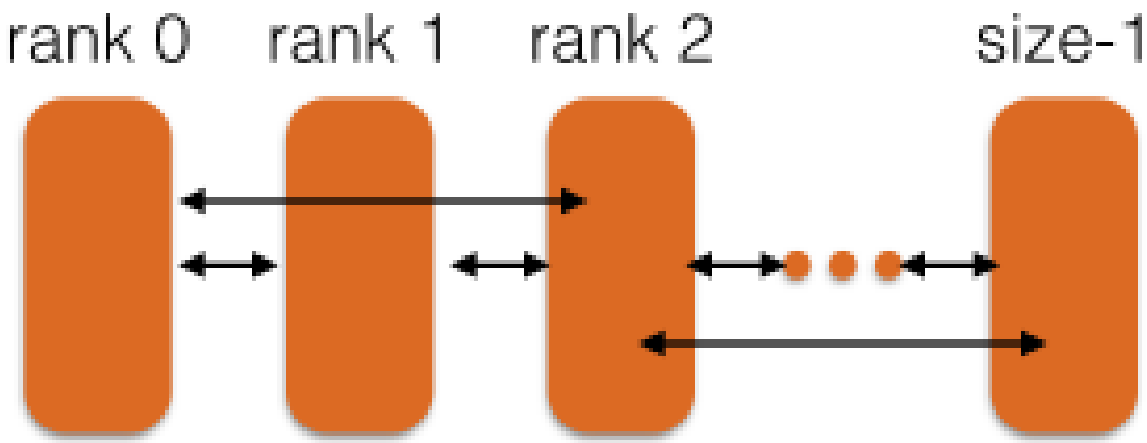


pbdR, continued

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:

```
ejspence@gpc-f108n045-ib0 ~> cd /scinet/course/ss2016/R/pbd
ejspence@gpc-f108n045-ib0 pbd>
ejspence@gpc-f108n045-ib0 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
COMM.RANK = 0
[1] 41038948
 real    1m15.357s
 user    9m39.943s
 sys     0m10.910s
ejspence@gpc-f108n045-ib0 pbd>
```

Departure hour histogram example, cont

```
ejspence@gpc-f108n045-ib0 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()
```

```
ejspence@gpc-f108n045-ib0 pbd>
```

Departure hour histogram example, cont

Let's look at the first few lines:

```
ejspence@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()
```

```
ejspence@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()‘.

Reductions

```
ejspence@gpc-f108n045-ib0 pbd> cat min-median-max.R

library(pbdMPI);   init()
rank <- comm.rank()
my.year <- start.year + rank

myfile <- paste0("../data/airline/airOT",as.character(my.year),".RDS")
data <- readRDS(myfile); data <- data$CRS_ELAPSED_TIME
data <- data[!is.na(data)]

data.median <- pbd.quantile(data,0.5)
data.min <- allreduce(min(data), op = "min")
data.max <- allreduce(max(data), op = "max")

comm.print(data.min)
comm.print(data.median)
comm.print(data.max)

finalize()
```

Reductions, continued

Reductions are one way of combining results, and they're very powerful:

```
ejspence@gpc-f108n045-ib0 pbd>
ejspence@gpc-f108n045-ib0 pbd> mpirun -np 4 Rscript min-median-max.R
COMM.RANK = 0
[1] -70
COMM.RANK = 0
[1] 93.00004
COMM.RANK = 0
[1] 1613
ejspence@gpc-f108n045-ib0 pbd>
```

Finding the median

R's higher-level functions plus reductions are very powerful ways to do otherwise tricky distributed problems - like median of distributed data:

```
pbd.quantile <- function( data, q = 0.5 ) {

  if (q < 0 | q > 1) {
    stop("q should be between 0 and 1.")
  }

  N <- allreduce(length(data), op = "sum")
  data.max <- allreduce(max(data), op = "max")
  data.min <- allreduce(min(data), op = "min")

  f.quantile <- function(x, prob=0.5) {
    allreduce(sum(data <= x), op="sum" )/N - prob
  }
  uniroot(f.quantile, c(data.min, data.max), prob=q)$root
}
```

pbd*apply

'pbd' has parallel apply functions. Note that work isn't farmed out by a master task; the tasks decide which parts of the list are theirs.

```
ejspence@gpc-f108n045-ib0 pbd> cat histogram-pbdsapply.R
⋮
year.hours <- function(my.year) {
  myfile <- paste0("data/airline/airOT",as.character(my.year),".RDS")
  data <- readRDS(myfile)$DEP_TIME
  count.hours(data)
}

init();    years <- 1990:1993
all.hours.list <- pbdLapply(years, year.hours)
all.hours <- Reduce("+", all.hours.list)

comm.print( all.hours )
comm.print( sum(all.hours) )
finalize()
```

pbd data distributions

pbd has a couple of ways of distributing data.

What we've used before is their so-called "GBD" distribution - globally distributed data. It's split up by rows.

However, for linear algebra computations, a block-cyclic distribution is much more useful.

Row Distributed

Block-Cyclic (dmatrix)

Reading a pbd Ddmatrix

pbdR comes with several packages for reading a data file and distributing it as a ddmatrix:

- ‘read.csv.ddmatrix()‘ for reading from csv
- ‘nc_get_dmat()‘ to read from a NetCDF4 file
- ‘gbd2dmat()‘ for conversions from row-oriented to a ddmatrix.

pbd lm

Several operations defined on regular R matrices also work transparently on ddmatrix: ‘lm‘, ‘solve‘, ‘chol‘.

```
ejspence@mycomp ~> cat pbd-lm.R
⋮
init.grid()
rank <- comm.rank()
my.year <- start.year + rank

data <- cleandata(my.year)
Y <- data[[1]]
X <- as.matrix(data[,-1])

X.dm <- gbd2dmat(X)
Y.dm <- gbd2dmat(Y)

fit <- lm(Y   X)
comm.print(summary(fit))

finalize()
```

pbd lm, continued

```
ejspence@gpc-f108n045-ib0 pbd> mpirun -np 4 Rscript pbd-lm.R
Using 2x2 for the default grid size
COMM.RANK = 0
Call:
lm(formula = Y ~ X)
Residuals:
      Min       1Q   Median       3Q      Max
 -1307.62   -6.03    -2.29    3.53   1431.70
Coefficients:  (6 not defined because of singularities)
                        Estimate   Std.  Error   t value     Pr(>|t|)
       (Intercept)    1.152e+01     9.616e-02    119.77   <2e-16 ***
 XORIGIN_AIRPORT_ID   -1.895e-04     5.193e-06    -36.50   <2e-16 ***
   XDEST_AIRPORT_ID   -2.257e-04     5.213e-06    -43.29   <2e-16 ***
                                        ⋮

---
Signif.  codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.'  0.1 ' ' 1
Residual standard error:  13.43 on 2741063 degrees of freedom
Multiple R-squared:  0.7809, Adjusted R-squared:  0.7809
F-statistic:  1.628e+06 on 6 and 2741063 DF, p-value:  < 2.2e-16
```