Lecture 13

Parallel R

GEOG 489

SPRING 2019

Parallel R

The basic concept of parallel programming is the following:

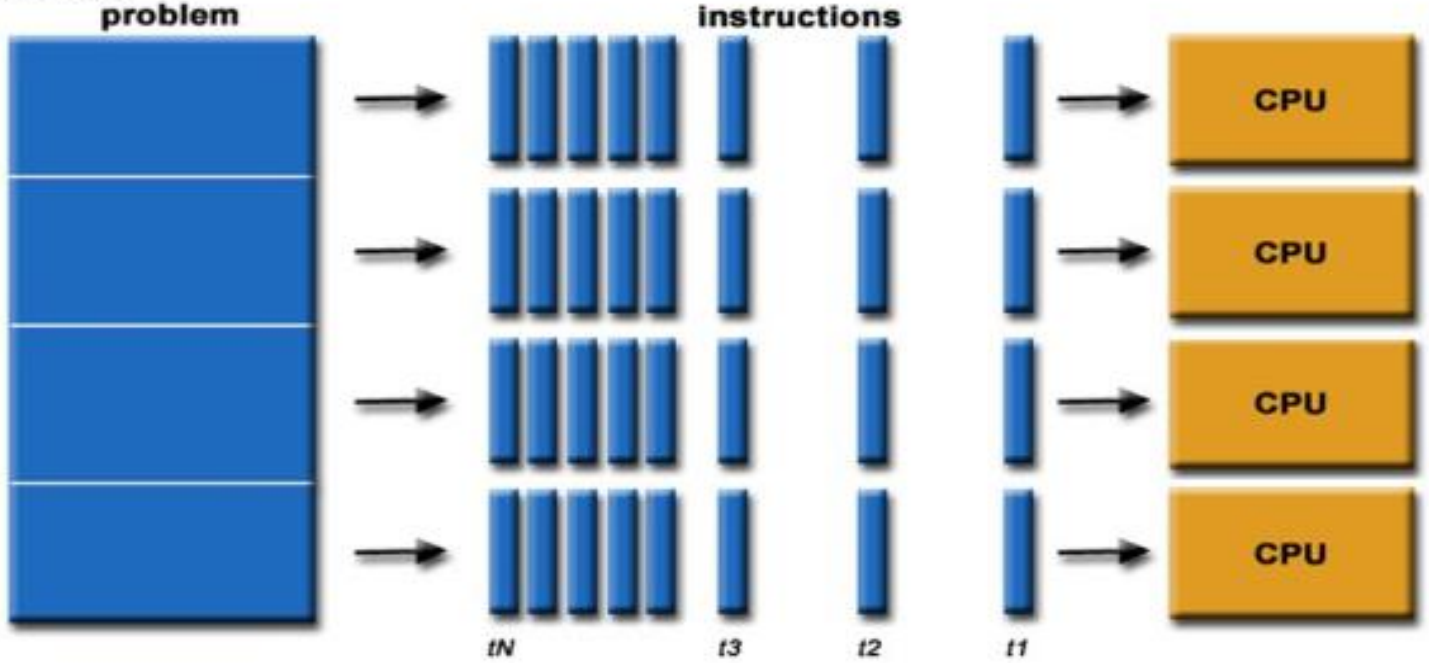
Given some problem X that can be divided into subproblems x1, x2, x3, each subproblem can be sent to a different "worker" processor (which may be located on a different physical computer).

Each processor then sends the results of its subproblem back to a central "master".

The master often then pieces the subproblems back together to return to the user.

Parallel R

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs



Parallel R

We are going to run a test using genetic data. The goal is to calculate the correlation coefficient between all pairs of genes across all samples.

```
source("http://bioconductor.org/biocLite.R")

biocLite("Biobase")

data(geneData, package = "Biobase")
```

Parallel R

This data represents 500 genes (organized by rows) with expression data (numerical) (500 rows * 26 columns)

	A	B	C	D	E	F	G	H	I	J
AFFX-MurIL2_at	192.742000	85.753300	176.757000	135.575000	64.493900	76.356900	160.505000	65.963100	56.9039000	135.608000
AFFX-MurIL10_at	97.137000	126.196000	77.921600	93.371300	24.398600	85.508800	98.908600	81.693200	97.8015000	90.483800
AFFX-MurIL4_at	45.819200	8.831350	33.063200	28.707200	5.944920	28.292500	30.969400	14.792300	14.2399000	34.487400
AFFX-MurFAS_at	22.544500	3.600930	14.688300	12.339700	36.866300	11.256800	23.003400	16.213400	12.0375000	4.549780
AFFX-BioB-5_at	96.787500	30.438000	46.127100	70.931900	56.174400	42.675600	86.515600	30.792700	19.7183000	46.352000
AFFX-BioB-M_at	89.073000	25.846100	57.203300	69.976600	49.582200	26.126200	75.008300	42.335200	41.1207000	91.530700
AFFX-BioB-3_at	265.964000	181.080000	164.926000	161.469000	236.976000	156.803000	211.257000	235.994000	175.6400000	229.671000
AFFX-BioC-5_at	110.136000	57.288900	67.398000	77.220700	41.348800	37.978000	110.551000	47.769000	24.7875000	66.730200
AFFX-BioC-3_at	43.079400	16.800600	37.600200	46.527200	22.247500	61.640100	33.662300	31.442300	23.1008000	39.741900
AFFX-BioDn-5_at	10.918700	16.178900	10.149500	9.736390	16.902800	5.333280	25.118200	38.757600	31.4041000	0.398779
AFFX-BioDn-3_at	751.227000	515.004000	622.901000	669.859000	414.165000	654.078000	704.781000	472.087000	456.4960000	601.335000
AFFX-CreX-5_at	76.943700	40.907000	62.031400	54.421800	29.070400	19.527100	56.316400	36.204400	34.4118000	54.076500
AFFX-CreX-3_at	105.378000	97.493200	74.029900	54.527700	54.984900	58.087700	96.632000	52.731000	35.4588000	60.264200
AFFX-BioB-5_st	40.482600	7.458010	19.406900	20.624600	25.049600	12.480400	21.910200	23.772000	24.1840000	29.703200
AFFX-BioB-M_st	58.170600	15.792600	25.196200	46.505700	15.315700	16.683300	93.175900	-2.286000	9.0048500	13.125300
AFFX-BioB-3_st	257.619000	113.690000	187.796000	210.580000	137.390000	104.159000	296.287000	110.536000	123.7670000	165.210000
AFFX-BioC-5_st	129.056000	74.609500	82.827100	101.534000	83.498600	73.198600	110.631000	116.742000	149.3290000	113.737000

Parallel R

```
# Let's make an even bigger dataset, making 26 x 4 = 104 columns.

fakeData <- cbind(geneData,geneData,geneData,geneData)

# Now lets make a more complex function that calculates the 95%
confidence intervals of the correlation coefficients through a process of
bootstrapping:

geneCor2 <- function(x, gene = fakeData)

{

    mydata <- cbind(gene[x[1], ], gene[x[2], ])

    mycor <- function(x, i) cor(x[i,1], x[i,2])

    boot.out <- boot(mydata, mycor, 1000)

    boot.ci(boot.out, type = "bca")$bca[4:5]

}
```

Parallel R

The basic order of using 'parallel' is as follows:

1) library("parallel")

2) make a parallel cluster using makeCluster(...)

3) load packages that your function needs into the workers using clusterEvalQ(cl=...,library(...))

4) load objects from the master environment into the worker environments using clusterExport(cl=...)

5) Following basic lapply() semantics, use e.g. clusterApplyLB(cl=...) to apply your function to an input list, where each iteration of the "loop" will be sent to an available processor. The output is usually a list.

6) Shut down your cluster using stopCluster(...)

Parallel R

The basic order of using 'parallel' is as follows:

1) library("parallel")

2) make a parallel cluster using makeCluster(...)

We are going to create a cluster with 4 cpus of type "PSOCK".

Creates a set of copies of R running in parallel and communicating over sockets

myCluster <- makeCluster(spec=4,type="PSOCK")

We can send the same function to each node (worker) in the cluster

workerDates <- clusterCall(cl=myCluster,fun=date)

workerPackages <- clusterCall(cl=myCluster,fun=search)

Parallel R

The basic order of using 'parallel' is as follows:

3) load packages that your function needs into the workers using clusterEvalQ(cl=...,library(...))

We are using a package called "boot" in our function, so we need to load up this package on every worker:

clusterEvalQ(cl=myCluster, library("boot"))

We can confirm the boot package is now loaded:

clusterCall(cl=myCluster,fun=search)

Parallel R

The basic order of using 'parallel' is as follows:

4) load objects from the master environment into the worker environments using clusterExport(cl=...)

Next, we will export the dataset "fakeData" to each worker:

clusterExport(cl=myCluster,"fakeData")

Check the environment on each worker:

clusterEvalQ(cl=myCluster,ls())

Note we could have sent the entire Global environment over to the workers.

clusterExport(cl=myCluster,ls())

clusterEvalQ(cl=myCluster,ls())

Parallel R

The basic order of using 'parallel' is as follows:

5) Following basic lapply() semantics, use e.g. clusterApplyLB(cl=...) to apply your function to an input list, where each iteration of the "loop" will be sent to an available processor. The output is usually a list.

This is VERY similar to an lapply statement, except we identify the cluster to send the command to.

```
system.time(outcor2 <- clusterApplyLB (cl=myCluster,pair2,geneCor2))
 user  system elapsed
  0.11    0.02   11.20
# vs the non-clustered version:
system.time(outcor <- lapply(pair2,geneCor2))
 user  system elapsed
 30.20    1.25   32.22
```

Parallel R

foreach: making parallel computing EVEN EASIER. There are multiple parallel "backends" to R, including parallel, snow, multicore, Rmpi, to name a few.

foreach is a meta-wrapper that works on many parallel backends. What this means is you can write one set of code, and not have to mod it if the user prefers to use Rmpi instead of parallel (in which case the commands are very different).

Parallel R

Parallel computation depends upon a parallel backend that must be registered before performing the computation.

The parallel backends available will be system-specific, but include doParallel, which uses R's built-in parallel package, doMC, which uses the multicore package, and doMPI.

Each parallel backend has a specific registration function, such as registerDoParallel or registerDoSNOW.

Parallel R

The basic order of using foreach is as follows:

1) library("foreach")
2) Load a parallel backend and foreach registration package
   e.g. library("doParallel")
3) Start a parallel backend with e.g. makeCluster(...)
4) Register the parallel backend with foreach,
   e.g.: registerDoParallel(...)
5) Use foreach(...) %dopar% function() to run your function
   in parallel.
6) Use .packages parameter in foreach(...) to load needed packages.
7) Stop your cluster using e.g. stopCluster(...)
8) Register the sequential backend to foreach using registerDoSEQ()

Parallel R

The basic order of ops with using foreach is as follows:

3) Start a parallel backend with e.g. makeCluster(...)

```
# Create a cluster using parallel:
myCluster <- makeCluster(spec=4,type="PSOCK")
```

4) Register the parallel backend with foreach, e.g.: registerDoParallel(...)

```
# Register the backend with foreach:
registerDoParallel(myCluster)
```

Parallel R

The basic order of ops with using foreach is as follows:

5) Use foreach(...) %dopar% function() to run your function in parallel.

6) Use .packages parameter in foreach(...) to load needed packages.

```
system.time(
  outcor <- foreach(p = pair2, .packages="boot", .combine="rbind")
  %dopar% { geneCor2(p) }
)
```

Parallel R

The basic order of ops with using foreach is as follows:

7) Stop your cluster using e.g. stopCluster(...)

```
# Don't forget to stop the cluster:
stopCluster(myCluster)
```

8) Register the sequential backend to foreach using registerDoSEQ()

```
# You should also register the default, non-parallel backend for
foreach, otherwise the next time you use foreach it will not work.
registerDoSEQ()
```

Parallel R

Sequential mode:

```
 system.time(
  outcor <- foreach(p = pair2, .packages="boot", .combine="rbind")
     %dopar%
     {
          return(geneCor2(p))
     }
)
```

Elapsed time

27.33

Parallel R

Parallel mode (4 cores):

```
 myCluster <- makeCluster(spec=4,type="PSOCK")
 registerDoParallel(myCluster)


system.time(
   outcor <- foreach(p = pair2, .packages="boot", .combine="rbind")
   %dopar% { geneCor2(p) }
 )
```

Elapsed time

8.92

Parallel R

Sources of overhead

Parallel computing, optimally, should be linear in terms of number of processors vs. time. However, this is never the case. A process running on one core does not take twice the time as a process running on two cores.

There are losses along the way from various sources. These need to be thought about when writing the most efficient code.

First, many programs will have parallel and non-parallel components. If your non-parallel components take X amount of time, no matter how many processors you have available, you will never run faster than X amount of time.

Parallel R

Sources of overhead

Chunking is one of the most important considerations. So far, we've been iterating one "row" at a time. For faster computations, this may not be very efficient, as the overhead of sending/receiving/managing the parallel cluster swamps out gains from the processing.

We can get more clever with this by sending MULTIPLE rows at one time to a worker. This optimization of the chunk size (number of rows to send at one time to a single worker) can dramatically speed up your computation, at the cost of heavier RAM usage.

Parallel R

Sources of overhead

Shared-memory machines: a single computer like the one you are working on is a "shared memory machine" -- each processor has access to the same physical RAM.

When two processors "ask" for an R object in RAM, they must compete against each other. One process will have to wait until the other process are done reading that area of memory. Memory access, thus, can be a bottleneck.

Parallel R

Sources of overhead

Networked systems of computers: a cluster computer ("supercomputer") is really just a bunch of individual computers networked together. Each individual computer ("node" in cluster terminology) has its own set of processors, memory, and hard drive space.

When the master R process does things like sends variables to the workers, it sends this data through a network, which is a lot slower than sending it within a single computer. This transfer of data causes relatively severe bottlenecks that can be ameliorated by faster networking or more clever code.

Assignment 3

Your goal is to write a function that calculates a set of descriptive statistics for a data frame, and return the output in a list format.

Input: x=data frame, probs=quantile probabilities, na.rm (logical)

Output: list, one list component per column, named after the data frame columns

Sublist components (the components should be named "mean","median","sd", and "quantiles"):

```
#  mean: the column's mean
#  median: the column's median
# sd: the column's standard deviation
# quantiles: a matrix of probs vs. the quantile of x for those probs.
# The matrix should have one prob/quantile per row.
# The first column should be named "prob" and the second "quantile".
```

Assignment 3

Your goal is to write a function that calculates a set of descriptive statistics for a data frame, and return the output in a list format

The na.rm flag should be used with all stats calculations

if a column cannot have statistics calculated, the value should return NA

WITHOUT a warning. Note that:

```
column1 <- c("abc","def")
mean(column1)
```

returns an NA but also a warning... you will lose a point if this happens.

The quantiles, as well, should just be an NA (not a matrix).

Assignment 3

The requirements are as follows:

1) The function name should be "descriptiveStats".

2) probs should be, by default, set to a 3-element vector of 0.25, 0.5 and 0.75.

3) na.rm should be set to FALSE by default.

4) x must be checked to make sure it is a data.frame, and stopped with a warning if not.

5) probs must be checked to make it is a numeric vector, and that all values are greater than or equal to 0.0 and less than or equal to 1.0. It should stop with a warning, if not.

Assignment 3

The requirements are as follows:

6) Comment your code in at least 3 places.

7) You may NOT use any R packages except the default set.

8) The code should be submitted to Compass 2g as a single function with the filename: lastname_firstname_geog489-s20-assignment-03.R and should have at the top:

```
#    [Your name]
#    Assignment #3
```

Assignment 3

Hint:

lapply() applied to a data frame will apply the function to each *column*, and return a list.

```
d2 <- data.frame(ages=c(10,7,12),names=c("Jill","Jillian","Jack"))

dl <- lapply(d2,sort)
```

Assignment 3 will be due the coming Tuesday, March 10, 2020