PARALLEL COMPUTATION WITH R

Isabelle Beaudry, Evan L. Ray
University of Massachusetts, Amherst

April 14, 2014

OUTLINE

1. Some context: options when your code is slow
2. A running example: network simulation
3. Parallel computing on one computer, multiple cores
   - Overview
   - snowfall (+ rstream)
   - foreach with doParallel and doRNG
4. Parallel computing on a cluster: the MGHPCC
   - Overview
   - Logistics: connecting, transferring files, and submitting jobs

SO YOUR CODE RUNS SLOWLY...

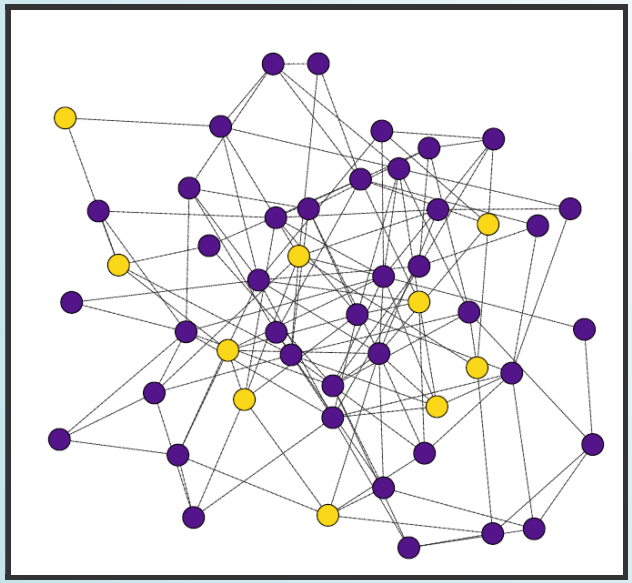
- **Step 0:** Make sure you're getting the right answer.
  - *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." - Donald Knuth*
  - Consider unit testing: See packages Runit and testthat
- **Step 1:** Profile your code to see where it's slow
  - See Rprof and the package profr
- **Step 2:** Consider using a different algorithm.
- **Step 3:** Consider modifying your R code
  - Pre-allocate memory
  - Use built-in functions instead of loops
- **Step 4 (a):** Consider using a faster language for the slow parts.
  - See package Rcpp
- **Step 4 (b):** Consider parallelizing

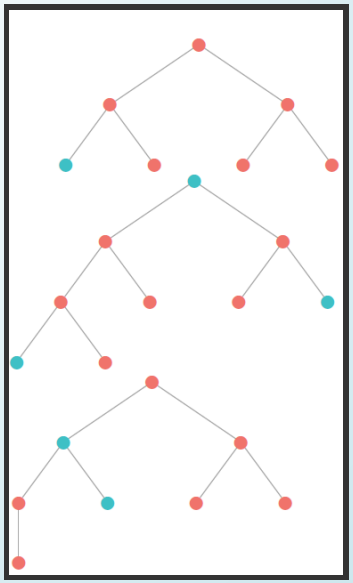
SOME GENERAL RESOURCES

- Advanced R development, by Hadley Wickham
  - http://adv-r.had.co.nz/
- 2008 UseR presentation by Dirk Eddelbuettel:
  - http://www.statistik.uni-dortmund.de/useR-2008/tutorials/useR2008introhighperfR.pdf
- High performance computing task view on CRAN:
  - http://cran.r-project.org/web/views/HighPerformanceComputing.html

EXAMPLE: NETWORK SIMULATIONS

- We want to conduct a simulation study to evaluate a method for estimating a population proportion from a respondent-driven sample (RDS).
- Each step of the simulation will require the following:
  1. Simulate a network at random
  2. Draw a sample from this network according to the RDS design
  3. Estimate the population proportion





EXAMPLE: NETWORK SIMULATIONS

- Here's some code to do this simulation study:

```
# ...load packages, define necessary functions, etc...

# allocate memory to store results
rds.sample1 <- array(NA, dim=c(n.net, n.nodes, 5))
est <- rep(NA, n.net)

for (j in 1:n.net){
  # simulate one network
  net <- create.nets(1)

  # simulate a respondent driven sample from the network
  rds.sample1[j, , ] <- rds.s(net)
  rds.frame <- create.df(A, rds.sample1[j, , ], rds.sample1[j, , 2])

  # estimate a population proportion based on the respondent driven sample
  est[j] <- RDS.II.estimates(rds.frame, outcome.variable="outcome")$estimate
}
```

PARALLELIZATION WITH SNOWFALL

- We can parallelize this simulation using the snowfall package as follows:

```
library(snowfall)

# create a cluster with 4 cpus, using sockets for communication
sfInit(parallel = TRUE, cpus = 4, type = "SOCK")

# export all objects in the global environment to the cluster nodes.
# see also the sfExport function for exporting specific objects.
sfExportAll()

# load libraries on the cluster nodes
sfLibrary("rstream", character.only = TRUE)
sfLibrary("statnet", character.only = TRUE)
sfLibrary("RDS", character.only = TRUE)

# replace the for loop with sfSapply: snowfall's equivalent of sapply
est <- sfSapply(seq_len(n.net), function(j) {
  net <- create.nets(1)
  rds.sample <- rds.s(net)
  rds.frame <- create.df(A, rds.sample, rds.sample[, 2])
  return(RDS.II.estimates(rds.frame, outcome.variable = "outcome")$estimate)
}

# stop the cluster
sfStop()
```

BE CAREFUL ABOUT RANDOM NUMBERS!!

- You must take care to ensure that:
  - results are reproducible
  - numbers generated are independent

REVISED EXAMPLE: USING THE RSTREAM PACKAGE

```
library(snowfall)
library(rstream)
sfInit(parallel = TRUE, cpus = 4, type = "SOCK")

# create an rstream object.  It requires 6 integers as a seed.
set.seed(1)
rngstream <- new("rstream.mrg32k3a", seed=sample(1:10000, 6, replace = FALSE))

# pack the rng stream in preparation for exporting to cluster nodes
rstream.packed(rngstream) <- TRUE

sfExportAll()

sfLibrary("rstream", character.only = TRUE)
sfLibrary("statnet", character.only = TRUE)
sfLibrary("RDS", character.only = TRUE)

est <- sfSapply(seq_len(n.net), function(j) {
  # unpack the rng stream
  rstream.packed(rngstream) <- FALSE

  # advance to a substream specific to this iteration of the simulation
  for(i in seq_len(j))
    rstream.nextsubstream(rngstream)

  # set the rng stream so that it is used by R in random number generation
  rstream.RNG(rngstream)

  net <- create.nets(1)
  rds.sample <- rds.s(net)
  rds.frame <- create.df(A, rds.sample, rds.sample[, 2])
  return(RDS.II.estimates(rds.frame, outcome.variable = "outcome")$estimate)
```

THE FOREACH PACKAGE WITH DORNG

- The foreach package provides the following general construction:

```
foreach(i = 1:3) %dopar% {
  # do some stuff
}
```

- We have to register a parallel backend with foreach.
- There are many options: doParallel/parallel, doMPI/Rmpi, doMC/multicore, d
- We will focus on doRNG: Ties into doParallel or doMPI to handle parallelizati reproducible RNG.

IMPLEMENTING OUR EXAMPLE WITH DORNG

```
library(doParallel)
library(doRNG)

# create a cluster with 4 cpus
nCores <- 4
cl <- makeCluster(nCores)

# register the cluster so that doParallel is used as the back end
registerDoParallel(cl)

# export all objects to the cluster nodes
clusterExport(cl = cl, ls(), envir = environment())

# parallelize the for loop using foreach and doRNG
# load the packages statnet and RDS on the cluster nodes
# set the RNG seed to 123
# combine results using the cbind function
est <- foreach(i = 1:n.net, .packages = c("statnet", "RDS"),
        .options.RNG = 123, .combine = cbind) %dorng% {
  net <- create.nets(1)
  rds.sample3 <- rds.s(net)
  rds.frame <- create.df(A, rds.sample3, rds.sample3[, 2])
  return(RDS.II.estimates(rds.frame, outcome.variable = "outcome")$estimate)
}

# stop the cluster
stopCluster(cl)
```

MGHPCC

- The Massachusetts Green High Performance Computing Center is:
  - Run by University of Massachusetts, Boston University, Harvard University, MIT, and Northeastern University
  - 5312 cores available and 400TBs of storage
  - LEED Platinum certified
  - located in Holyoke
- There is a wiki at http://wiki.umassrc.org/wiki
- You can request access at http://wiki.umassrc.org/wiki/index.php/Requesting_Access

LOGISTICS: CONNECTING, TRANSFERRING FILES, AND SUBMITTING JOBS

- To use the cluster, we need to do the following:
  1. Transfer data/scripts to the cluster with FTP
  2. Log in to the cluster
  3. Install any needed packages
  4. Submit a job
  5. Transfer data/results back from the cluster to your computer
- Detailed instructions for transferring files and logging in are on the BiP slides
  - But there is an important step missing for Windows users!!!
    After uploading your scripts, run dos2unix to convert file formats. For exa

```
dos2unix network_sf_rstream.R
```

- Next, we will discuss installing packages and submitting jobs

INSTALLING PACKAGES

- The cluster does not have many R packages installed by default. To install th
  1. Load necessary modules (software packages):
     - You will need the R module:

       ```
       module load R/3.0.2
       ```

     - For some packages, you may need to load other modules such as the G

       ```
       module load gcc/4.8.1
       ```

     - You can view the full list of available modules with the following comm

       ```
       module avail
       ```

  2. Install the package as usual, using install.packages() from within R or R CM command line.

SUBMITTING JOBS

- Once you have uploaded your scripts to the cluster, there are two steps to ru
  1. Create a shell script like the following:

```
#!/bin/bash
#BSUB -R rusage[mem=1024] # ask for memory
#BSUB -n 4                # how many cores we want for our job
#BSUB -R span[hosts=1]    # ask for all the cores on a single machine
#BSUB -W 0:10             # not sure what this is doing
#BSUB -q short            # which queue we want to run in

module load R/3.0.2
R CMD BATCH /home/er71a/ParallelR/network_sf_rstream.R
```

  2. Submit to the scheduler

ghpcc06.umassrc.org - PuTTY

```
-bash-4.1$ bjobs
No unfinished job found
-bash-4.1$ bsub < submit_network_sf_rstream.sh
Job <125543> is submitted to queue <short>.
-bash-4.1$ bjobs
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
125543  er71a   PEND  short      ghpcc06                 *rstream.R Apr 11 21:57
-bash-4.1$ bqueues
QUEUE_NAME      PRIO STATUS          MAX JL/U JL/P JL/H NJOBS  PEND   RUN  SUSP
priority        100  Open:Active       -    -    -    -     0     0     0     0
interactive      90  Open:Active       -    5    -    -     1     0     1     0
short            80  Open:Active    2854    -    -    -  1099     0  1099     0
parallel         70  Open:Active    2854    -    -    -  2136   444  1112     0
long             60  Open:Active       -    -    -    -  5633  3365  1341   839
gpu              50  Open:Active       -    -    -    -    15     9     6     0
-bash-4.1$
```

RESULTS

- After your job runs, the console log will be in a file like network_sf_rstream.Rout.
- If you will need access to other R objects, you will need to explicitly save them!
  - For saving plots: png(), jpeg(), pdf()
  - For saving objects: save(), save.image()
- You may want to save intermediate results.

RESOURCES

- These slides and our example code are available on GitHub:
  - https://github.com/elray1/ParallelR
  - The HTML slides probably only display correctly with google chrome; there is also a pdf version.
- The Biostatistics in Practice slides and examples are also available on GitHub:
  - https://github.com/nickreich/BiPSandbox/
  - Module 2 talks about parallel computation on a local machine
  - Module 3 talks about parallel computation on the MGHPCC