Parallel Programming in R
PS236b
Spring 2010

Mark Huberty

February 24, 2010

Concepts

Parallelization in R
- snow
- foreach
- multicore
- Three benchmarks

Considerations

Good habits: testing, portability, and error handling

The Political Science Cluster

GPU Computing

Parallelization: basic concepts

The commodity computing revolution

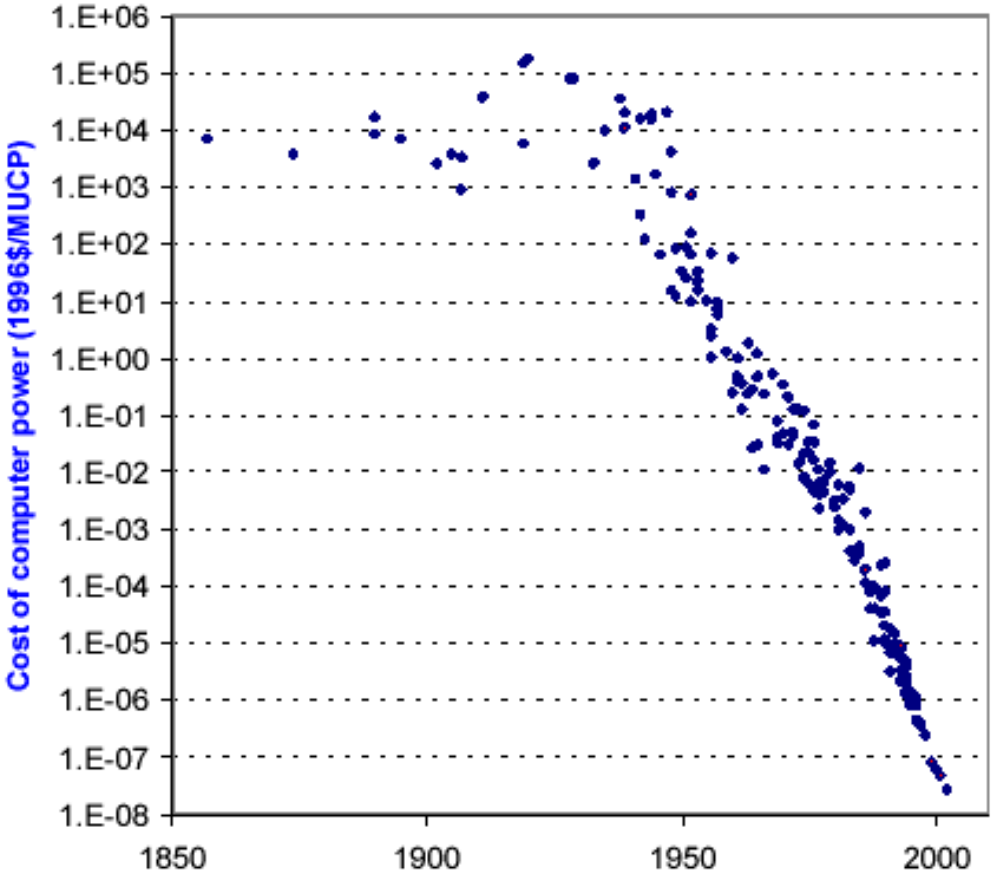
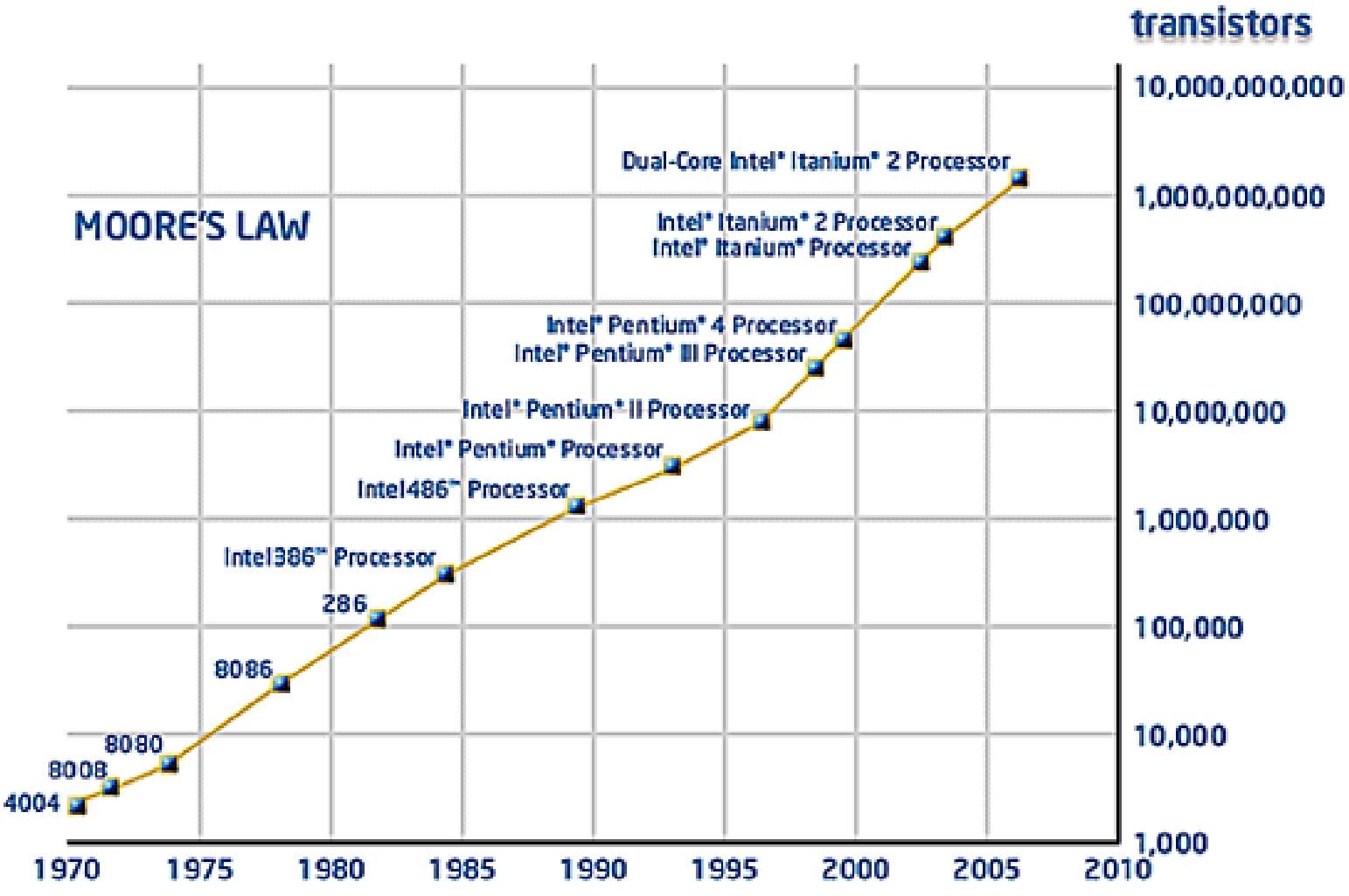


Figure 1. **The progress of computing measured in cost per million standardized operations per second (MUCP) deflated by the price index for GDP**

Source: See Appendix.

The fast computing revolution



The three aspects of the revolution

Cheap computing now comes in three forms:

- Many *cores* on the same chip
- Many *chips* in the same computer
- Many *computers* joined with high-speed connections

We can refer to any of these parallel processing units as `nodes`.

Kinds of parallelization

Taking advantage of these forms of compute power requires code that can do one of several kinds of parallelization:

- *Bit*-based parallelization
  Already have this: the move up the chain via 4-8-16-32-64 bit machines changes the number of steps required to run a single instruction
- *Instruction*-based parallelization
  Processor/program layer
- *Data*-based parallelization
  Decompose large data structures into independent chunks, on which you perform the same operation
- *Task*- based parallelization
  Perform different, independent tasks on the same data

For R, we are mostly interested in *data* and *task* parallelization

Data Parallelization, cont'd

Data parallelization is very common:

- Bootstrapping: Sample *N* times from data *D* and apply function *F* to each sample
- Genetic matching: generate *N* realizations of matches between groups *T* and *C* and calculate the balance on each; repeat for *G* generations
- Monte Carlo simulations

Google does a ton of this kind of work via its MapReduce framework

Task parallelization

Task parallelization is a little less obvious. Ideas include:

- Given N possible estimators of treatment effect β, test all against data set D
- Machine learning: given N different classification schemes for some data set D, generate some test statistics S for all of them

Data parallelization: an overview

So implement data parallelization, we must:

- Conceptualize the problem as a set of operations against independent data sets
- Break up this set of operations into independent components
- Assign each component to a node for processing
- Collect the output of each component and return it to the master process

Technical requirements for parallelization

For the hardware geeks:

- Multiple cores or servers

Technical requirements for parallelization

For the hardware geeks:

- Multiple cores or servers
- Some means to connect them

Technical requirements for parallelization

For the hardware geeks:

- Multiple cores or servers
- Some means to connect them
- A way to communicate among them
  - Sockets
  - MPI (Message Passing Interface)

Technical requirements for parallelization

For the hardware geeks:

- Multiple cores or servers
- Some means to connect them
- A way to communicate among them
  - Sockets
  - MPI (Message Passing Interface)
- A means of sharing programs and data

Technical requirements for parallelization

For the hardware geeks:

- Multiple cores or servers
- Some means to connect them
- A way to communicate among them
  - Sockets
  - MPI (Message Passing Interface)
- A means of sharing programs and data
- A framework to organize the division of tasks and the collection of results

How fast can we get?

For some program containing a function F that will operate on some data set D, decompose D into d_i, $i \in N$ and perform F on each, splitting the tasks across M nodes

For the program containing F, the maximum gain from parallelization is given by Amdahl's Law.

For a program with P percent parallizability running on M nodes:

$$S = \frac{1}{1 - P + \frac{P}{M}} \tag{1}$$

The Political Science Compute Cluster

Here's what we have:

- 9 2-chip Opteron 248 servers
- Gigabit ethernet interconnects
- OpenMPI message passing
- A Network File System

Parallelization in R

Frameworks for parallelization

R has several frameworks to manage data parallelization. Three mature and effective ones are:

- `snow`, which uses the `apply` model for task division
- `foreach`, which uses a `for` loop model for division
- `multicore`, which is only suitable for the many-cores hardware model

There are several other possibilities (`nws`, `mapreduce`, `pvm`) at different levels of obsolescence or instability.

The snow model

snow is a master/worker model: From N nodes create 1 master and $N - 1$ workers[1]; farm jobs out to the workers.

This is a little weird when using MPI-based systems where the nodes are undifferentiated; keep this in mind when using MPI for R jobs.

[1] Older parlance uses the master/slave terminology, which has been abandoned for obvious reasons

`snow` and parallelization

The `snow` library makes parallization straightforward:

- Create a cluster (usually with either **sockets** or **MPI**)
- Use parallel versions of the `apply` functions to run stuff across the nodes of the cluster

So this is pretty easy: we already know how to use `apply`

Cluster creation in `snow`

See the `clusterCreate` function in the `clusterSetup.R` code.

snow example

```
1 ## assuming you've already created a cluster cl:
2 m <- matrix(rnorm(16), 4, 4)
3 clusterExport("m")
4 parSapply(cl, 1:ncols(m), function(x){
5     mean(m[,x])
6 }
7 )
```

Notice there that `parSapply` has replaced `sapply`, but nothing much else has changed.

Data vs. Task parallelization with snow

Data:

```
1       parLapply(cl, 1:nSims, function(x){
2         n <- dim(data)[1]
3         simdata <- data[sample(1:n, n, replace=TRUE),]
4         out <- myfunc(simdata)
5         return(out)
6   }
7   )
```

Task:

```
1       funclist <- list(func1, func2, func3)
2       parLapply(cl, 1:length(funclist), function(x){
3           out <- funclist[[x]](data)
4           return(out)
5   }
```

Object management in `snow`

`snow` requires some additional object management:

- Libraries must be called on all nodes
- Data objects must be exported to all nodes

Object management in snow

snow requires some additional object management:

- Libraries must be called on all nodes
- Data objects must be exported to all nodes

As in:

```
1  ## Given a cluster called cl
2  m <- matrix(rnorm(100), ncol=10, nrow=10)
3  clusterExport(cl, "m")
4  clusterEvalQ(cl, library(Matching))
```

Notice that this doesn't apply to objects created *inside* a call to the cluster (i.e. parLapply().

`foreach` and parallelization

REvolution Computing released the `foreach` libraries. To use them, you install:

- `foreach`
- `doSNOW` for using `snow` clusters
- `doMPI` for working directly with MPI
- `doMC` for use on multicore machines

The basic idea: looks like a `for` loop, performs like an `apply`, and portable

foreach example

```
1  ## Load the libraries. I assume I'm on an MPI-based
2  ## cluster; other options are doSNOW and doMC
3  library(foreach); library(doMPI)
4
5  ## Get the cluster configuration.
6  cl <- startCluster()
7
8  ## Tell doMPI that the cluster exists
9  registerDoMPI(cl)
10
11 m <- matrix(rnorm(16), 4, 4)
12
13 ## Run the for loop to calculate the column means
14 foreach(i=1:ncol(m)) %dopar% {
15     mean(m[,i]) # makes m available on nodes
16 }
```

`foreach` continued

Notice the important bit:

```
## Run the for loop to calculate the mean
## of each column in m.
foreach(i=1:ncol(m)) %dopar% {
    mean(m[,i]) # makes m available on nodes
}
```

Here, the `foreach` term controls the repetition, but the `%dopar%` term does the actual work.

If this weren't running on a cluster, you would use `%do%` instead

multicore and parallelization

multicore only works on chips with > 1 core (dual-core, quad-core, etc). It's basic function is mclapply:

```
1  library(multicore)
2  m <- matrix(rnorm(16), 4, 4)
3
4  ## By default the function finds all cores on
5  ## the chip and uses them all
6  mclapply(1:ncols(m), function(x){
7      mean(m[,x])
8  },
9              mc.cores=getOption("cores")
10 )
11 ## Notice the last argument; can specify a
12 ## number of cores if desired
```

Three benchmarks, three lessons

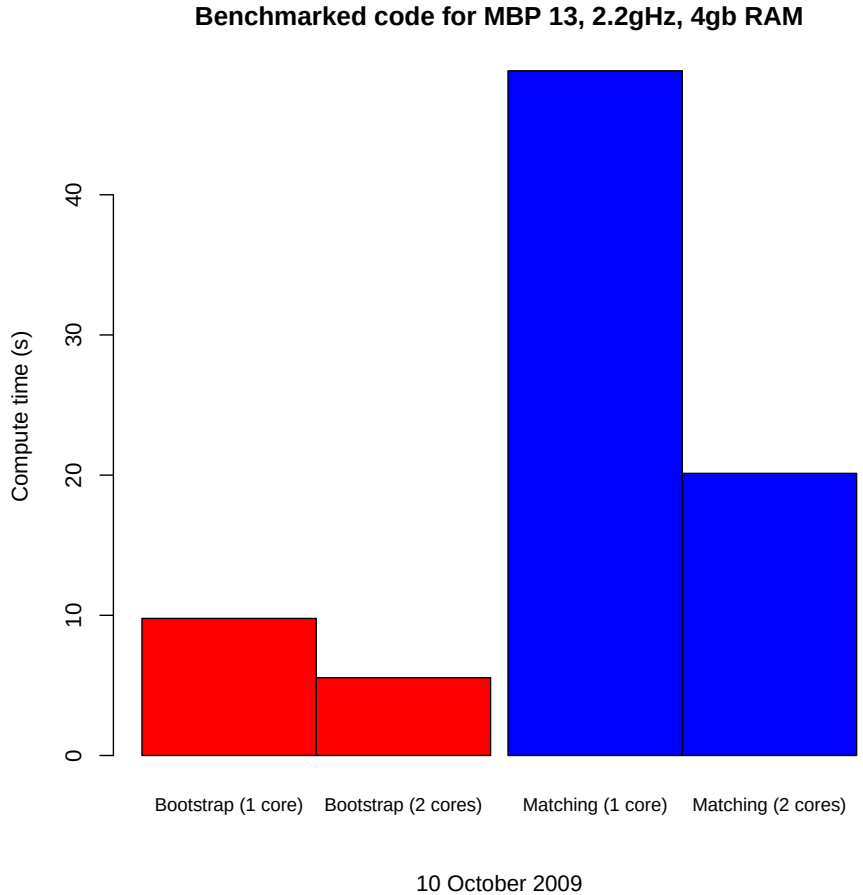
Parallel computing on your laptop

New laptops today will almost certainly come with *dual-core* chips.

Usual speed gains run around 50%

Parallel computing on your laptop

New laptops today will almost certainly come with *dual-core* chips.



Usual speed gains run around 50%

Matrix multiplication

Results for:

```
1  library(snow)
2  testmat <- matrix(rnorm(10000000), ncol=10000)
3  mm.serial <- system.time(testmat %*% t(testmat))
4
5  testmat.t <- t(testmat)
6  source("setupCode.R")
7
8  clusterCreate()
9  clusterExport(cl, c("testmat", "testmat.t"))
10 mm.parallel <- system.time(parMM(cl, testmat, testmat.t))
11
12 save.image("mm.results.RData")
13 clusterShutdown()
```

`mm.parallel` \ `mm.serial` = 0.6 for an 8-node cluster

Parallelization and speed: an example

So why choose one? Let's look at speed and features.

The idea: write the same bootstrap as a serial job, a `snow` job, and a `foreach` job, and see what we get.

All the code is available at `http://pscluster.berkeley.edu`

Results: timing

Table: Benchmark results for 500 repetitions of a 1000-trial bootstrap for different coding methods; parallel methods use 8 nodes

	Mean time (s)	Pct of Serial time	2.5 pct CI	97.5 pct CI
Serial	176.4	100.0	171.8	181.3
parSapply	33.4	18.9	32.9	33.5
foreach, snow	27.3	15.5	26.8	27.8
foreach, dompi	27.2	15.4	26.7	27.8

Results: distributions

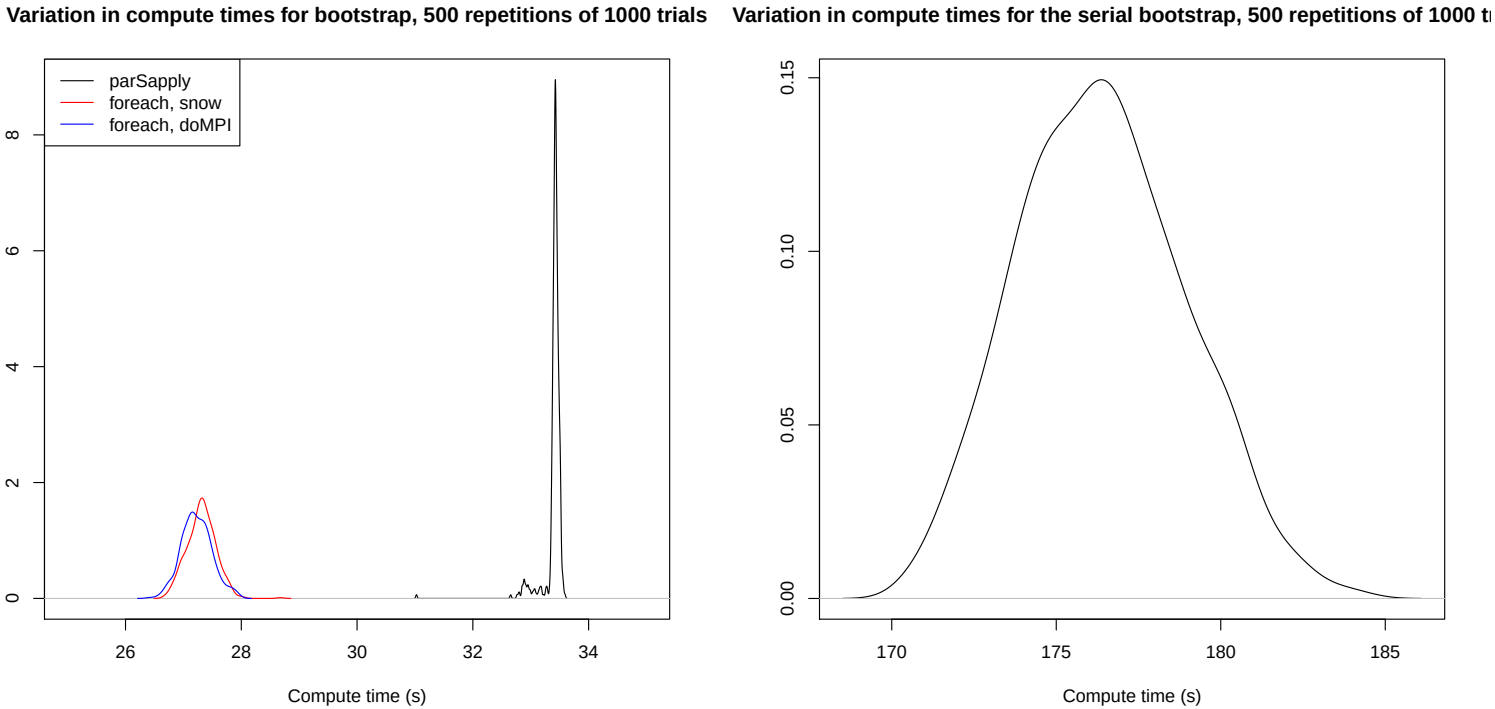


Figure: Benchmark distributions for serial and parallel bootstraps

doMPI vs. doSNOW

Given the identical performance, why choose one vs. the other?

- doMPI is not compatible with an environment that has a snow cluster running in it.
- Thus use doMPI when running things without snow, and doSNOW when combining code that requires snow with foreach-based routines.

Parallelization and data structures

Lists don't get much use in generic R but are very helpful for parallelization:

- For *N* data sets of the same format, do some identical analysis *A* on each of them

Parallelization and data structures

Lists don't get much use in generic R but are very helpful for parallelization:

- For *N* data sets of the same format, do some identical analysis *A* on each of them
  - Solution:
    1. Create some list *L* of length *N*, containing all the data sets;
    2. then loop across the list *N* times (using *P* chips)
    3. and apply function *A* to each data set

Parallelization and data structures

Lists don't get much use in generic R but are very helpful for parallelization:

- For *N* data sets of the same format, do some identical analysis *A* on each of them
  - Solution:
    1. Create some list *L* of length *N*, containing all the data sets;
    2. then loop across the list *N* times (using *P* chips)
    3. and apply function *A* to each data set
- Can be referred to as a `map/reduce` problem

R has a `mapReduce` package that claims to do this; but it's basically just an `parLapply` with more overhead.

Some complications

There are a few more issues to worry about:

- Memory management:
  - *Processors* parallelize but *memory* does not: the master node still has to hold all the results in RAM
  - If R runs out of RAM, the entire process **will die** and take your data with it
  - Solutions:
    - Reduce objects on the nodes as much as possible
    - Run things in recursive loops

Memory mgmt and object size

Comparative object sizes for the output of a regression, $N = 1000$, $P = 10$:

- Output from `lm`: 321k
- Output from `lm$coefficients`: 0.6k

Memory mgmt and object size

Comparative object sizes for the output of a regression, $N = 1000$, $P = 10$:

- Output from `lm`: 321k
- Output from `lm$coefficients`: 0.6k
- Factor of 535 difference!

Memory mgmt and object size

Comparative object sizes for the output of a regression, $N = 1000$, $P = 10$:

- Output from `lm`: 321k
- Output from `lm$coefficients`: 0.6k
- Factor of 535 difference!

So if you only need the coefficients, you use much less memory–and `lm` is a simple object compared to, say, `MatchBalance`

Recursive loops for memory mgmt: a `GenMatch` example

Common desire: want to run GenMatch with large population sizes.

Common problem: the MemoryMatrix grows very large and the kills off the R session with an "out of memory" error.

Solution: recursion:

```
1  ## Assuming a gen.out from a "dummy" run of GenMatch
2  par <- gen.out
3  for(i in 1:10){
4  par <- GenMatch(..., starting.values=par$starting.values,
5                        hard.generation.limit=TRUE,
6                        max.generations=25
7                  )}
```

Notice that here, the MemoryMatrix never grows large; but GenMatch still retains the history via recursion

Good habits: testing, portability, and error handling

Portability

To achieve portability of code across platforms, use `if` statements to set the appropriate environment variables.

Example: the working directory

```
1       if(oncluster==TRUE){
2         setwd("~/projectname/")
3         }else{
4         setwd("~/research/myprojects/projectname/")
5
6   }
```

Code testing and management

Parallel jobs are often *long* jobs, posing some issues:

- How to catch errors while writing code?
- How to test code functions?
- How to verify output before running?
- How to catch errors when running?

You want to check your code *before* starting, rather than have the process fail while running.

codetools for syntax checking

Luke Tierney's codetools package for R will do basic syntax checking for you:

```
1  library(codetools)
2  nSims <- 500
3  testFunction <- function(simcount){
4
5     sappl(1:simcount, function(x){
6
7        mean(rnorm(1000))
8
9        save.(nSims, file="nsims.RData")
10    }
11       )
12 }
```

Here, we should expect errors with sappl and save..

`codetools` output

Here's what we get:

```
> checkUsage(testFunction)
<anonymous>: no visible global function
               definition for 'sappl'
<anonymous> : <anonymous>: no visible global function
                definition for 'save.'
<anonymous> : <anonymous>: no visible binding for
                global variable 'nSims'
```

So `checkUsage()` will help catch unidentified variables, bad functions, and other typos before you actually run your jobs.

Good operating practice

1. Write code as a set of functions
2. Use conditional statements to make code portable between your laptop and the parallel environment
3. Check functions with `codetools`
4. Run trials of the code on small *N*

Note that to check a set of functions, load them up and use `checkUsageEnv()` to loop through all those functions, as in:

```
1    checkUsageEnv(env=.GlobalEnv)
```

The Political Science Compute Cluster

Technology

The basic configuration

- A master server with a dual-core 2.6 gHz Opteron chip and 8gb of RAM
- 9 2-chip 64-bit worker servers w/ 4-8gb of RAM each, for a total of **18 nodes**
- CentOS linux
- The Perceus cluster administration software
- Gigabit ethernet interconnects
- OpenMPI message-passing
- SLURM job management and resource allocation
- 1tb of RAID-1 storage for users (actually about 800gb)

Accounts

Accounts are available on these terms:

- Polisci faculty and grad students: 2 years, renewable
- Non-Polisci students in 200-level courses: 1 semester
- For other purposes: on request

All accounts come with 5gb of storage on the cluster itself.

To get an account, send email to `cluster@berkeley.edu`

Logistical details

There are some user services available:

1. The cluster administrators are available at `cluster@berkeley.edu`
2. Cluster users should sign up for the listserv, at `pscluster@lists.berkeley.edu`
3. Benchmarks, code, and documentation are available at the cluster webpage: `http://pscluster.berkeley.edu`

Finally, there is a comprehensive README file that all users **should** review. It can be found on the cluster webpage.

Resources

As of right now, we have the following resources available:

- 64-bit R, compiled against GOTO BLAS
- The C, FORTRAN, and MPICC compilers
- Emacs + ESS (in both X and terminal flavors)
- `git` for version control

If you want something else on the servers (Matlab, Mathematica, Stata) and can get the right licenses, we'd be happy to look at setting it up.

Access

Access is available both on-campus and off:

- On campus: via `ssh` to `pscluster.polisci.berkeley.edu`
- Off campus: through the VPN via `ssh` to the same address

The VPN software is available for free via the Software Central service (`software.berkeley.edu`).

The README file at `http://pscluster.berkeley.edu` has more information on access and software configuration.

`ssh` clients

`ssh` and `scp` require a client program. Which program depends on your OS:

- OS X, Linux: Use the Terminal application
- Windows: PuTTY is free; HostExplorer is a commercial alternative, available free at `http://software.berkeley.edu`

Note that any of these will give command-line access. There is no GUI.

Using the cluster

Two ways:

1. Serial jobs. No special programming needed. But you can only make use of a single node. Nice for long-running, single-threaded jobs.
2. Parallel jobs Some special programming required, but can take advantage of >1 node for speedup

A generic R session

Sessions will generally follow this pattern:

- Copy your code and data to your `home` directory on the cluster (via `scp`)
- Log into the cluster (`ssh`)
- Execute your code by requesting a certain number of nodes from SLURM and initiating the batch job
- Pull the output and the R transcript file back to your own computer (again via `scp`)

Running a job on 1 node

To run code against a single job, ask SLURM for 1 node and run your code against it: That looks like:

```
salloc -n 1 orterun -n 1 Rscript <yourcode.R>
```

Here:

- `salloc` asks SLURM for nodes
- `orterun` invokes MPI to choose a node
- `Rscript` runs your code file (where you have the code to pick up the cluster you just created)

Running a file on >1 node

To run code against multiple nodes, you need to (1) ask SLURM for the nodes, (2) invoke the cluster setup in MPI, and (3) call your own code in R.

That looks like:

```
salloc -n <number of nodes> orterun -n <number of nodes>
      Rscript <yourcode.R>
```

Here:

- `salloc` asks SLURM for nodes
- `orterun` creates a cluster with those nodes
- `Rscript` runs your code file (where you have the code to pick up the cluster you just created)

Running the job: convenience

For your convenience, we have a script that internalizes all this stuff:

```
Rnotify.sh <yourcodefile.R> <number of nodes>
                                <email address>
```

This will run your code against the number of requested nodes, and send you an email when the job is complete. As in:

```
Rnotify.sh mycode.R 4 me@berkeley.edu
```

Running the job: while you are away

Usually, you want to start the job and log out. To do that, a little extra is needed:

```
nohup Rnotify.sh <yourcodefile> ...etc...
```

Here, `nohup` is the Unix "no hangup" routine, which keeps stuff running even after you log out.

This will point the console output to a file called `nohup.out`. If you want it to go elsewhere, use this syntax:

```
nohup Rnotify.sh <yourcodefile> <nodecount>
                                <email> > filename.out
```

Monitoring the job

SLURM provides command-line tools to monitor and manage your jobs and check the status of the cluster:

- `squeue`, which prints a list of your jobs and their status and runtimes
- `sinfo`, which prints the status of each node in the cluster
- `scancel`, which allows you to cancel a job

You also have access to the normal Unix commands `top` and `ps` to look at your system processes.

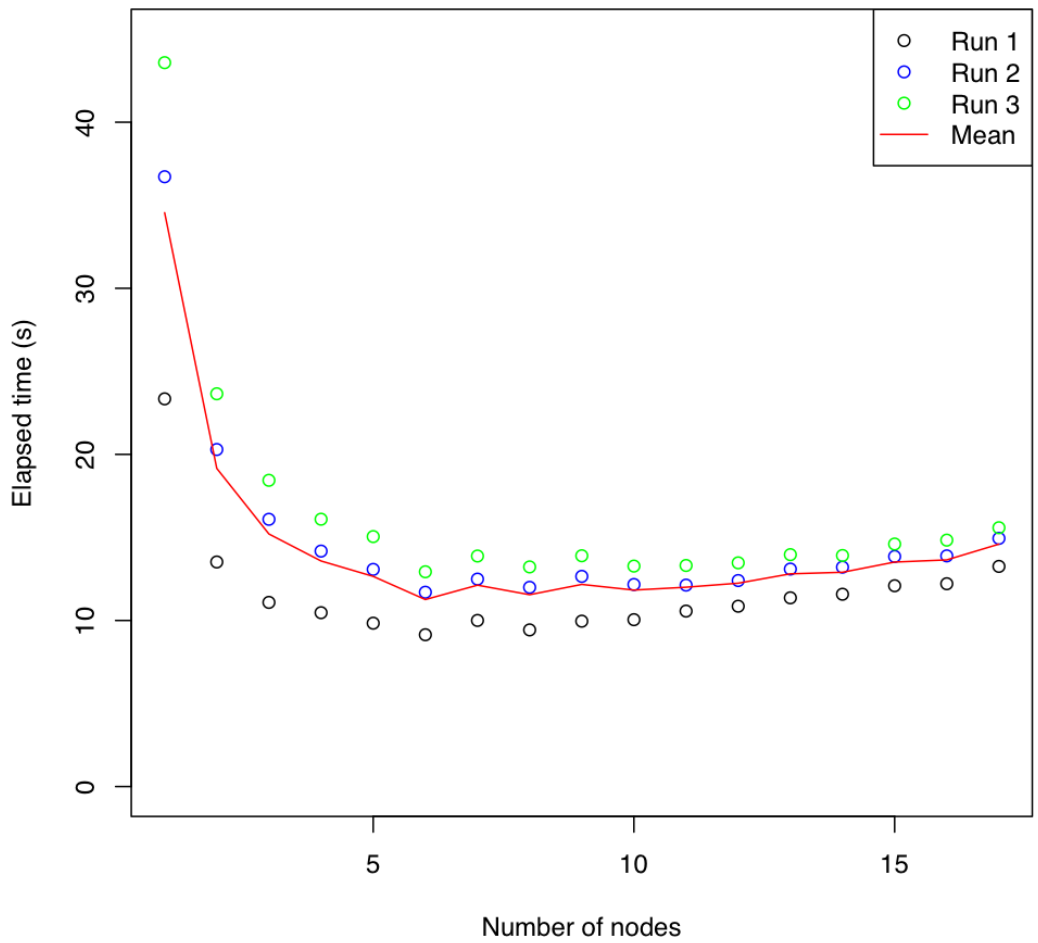
Choosing the number of nodes

You might be tempted to always ask for lots of nodes. There are three reasons this is a bad idea:

1. Bad form: this a commons, don't abuse it
2. Wait time: SLURM will force your job to wait until nodes are available; the job might start (and finish) sooner if you asked for fewer nodes
3. Processing time–more not always faster.

Choosing the number of nodes: a benchmark

This is GenMatch() run on 1-17 nodes:



The next generation: GPU computing and hundreds of cores

Blood, Gore, and Parallel computation

- High-end computer graphics cards for gaming now have 128-256 cores and can be bought for $200-400.
- A supercomputer on your desktop
- R has enabled use of these **GPUs** through a package called `gputools`

A supercomputer on your laptop

You may already have this at your disposal:

- All new MacBooks come with the nVidia 9400m GPU
- Many PC notebooks have a similar chip
- Information on how to install `gputools` can be found at the authors' website:
  - `http://brainarray.mbni.med.umich.edu/brainarray/rgpgpu/`
- Information on how to install `gputools` on a Macbook Pro can be found at:
  - `http://markhuberty.berkeley.edu/tech.html`

gputools results