Los Angeles R Users' Group

Taking R to the Limit, Part I: Parallelization

Ryan R. Rosario

July 27, 2010

The Brutal Truth

We are here because we love R. Despite our enthusiasm, R has two major limitations, and some people may have a longer list.

1. Regardless of the number of cores on your CPU, R will only use 1 on a default build.
2. R reads data into memory by default. Other packages (SAS particularly) and programming languages can read data from files on demand.
   - Easy to exhaust RAM by storing unnecessary data.
   - The OS and system architecture can only access $\frac{2^{32}}{1024^2} = 4\text{GB}$ of physical memory on a 32 bit system, but typically R will throw an exception at 2GB.

The Brutal Truth

There are a couple of solutions:

1. Build from source and use special build options for 64 bit and a parallelization toolkit.
2. Use another language like C, FORTRAN, or Clojure/Incanter.
3. Interface R with C and FORTRAN.
4. **Let clever developers solve these problems for you!**

We will discuss number 4 above.

The Agenda

This talk will survey several HPC packages in R with some demonstrations. We will focus on four areas of HPC:

1. **Explicit Parallelism:** the user controls the parallelization.
2. **Implicit Parallelism:** the system abstracts it away.
3. **Map/Reduce:** basically an abstraction for parallelism that divides *data* rather than *architecture*. (Part II)
4. **Large Memory:** working with large amounts of data without choking R. (Part II)

Disclaimer

1. There is a **ton** of material here. I provide a lot of slides for your reference.
2. We may not be able to go over all of them in the time allowed so some slides will go by quickly.
3. Demonstrations will be time permitting.
4. All experiments were run on a Ubuntu 10.04 system with an Intel Core 2 6600 CPU (2.4GHz) with 2GB RAM. I am planning a sizable upgrade this summer.

Parallelism

Parallelism means running several computations at the same time and taking advantage of multiple cores or CPUs on a single system, or CPUs on other systems. This makes computations finish faster, and the user gets more bang for the buck. "We have the cores, let's use them!"

R has several packages for parallelism. We will talk about the following:

1. `Rmpi`
2. `snowfall`, a newer version of `snow`.
3. `foreach` by Revolution Computing.
4. `multicore`
5. Very brief mention of GPUs.

Parallelism in R

The R community has developed several (and I do mean several) packages to take advantage of parallelism.

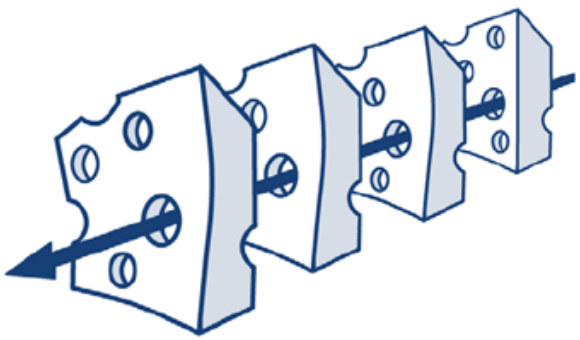
Many of these packages are simply wrappers around one or multiple other parallelism packages forming a complex and sometimes confusing web of packages.

Motivation

"R itself does not allow parallel execution. There are some existing solutions... However, these solutions require the user to setup and manage the cluster on his own and therefore deeper knowledge about cluster computing is needed. From our experience this is a barrier for lots of R users, who basically use it as a tool for statistical computing."

From *The R Journal Vol. 1/1, May 2009*

The Swiss Cheese Phenomenon



Another barrier (at least for me) is the fact that so many of these packages rely on one another. Piling all of these packages on top of each other like swiss cheese, the user is bound to fall through a hole, if everything aligns correctly (or incorrectly...).

(I thought I coined this myself...but there really is a swiss cheese model directly related to this.)

Some Basic Terminology

A CPU is the main processing unit in a system. Sometimes CPU refers to the processor, sometimes it refers to a `core` on a processor, it depends on the author. Today, most systems have one processor with between one and four cores. Some have two processors, each with one or more cores.

A **cluster** is a set of machines that are all interconnected and share resources as if they were one giant computer.

The **master** is the system that controls the cluster, and a **slave** or `worker` is a machine that performs computations and responds to the master's requests.

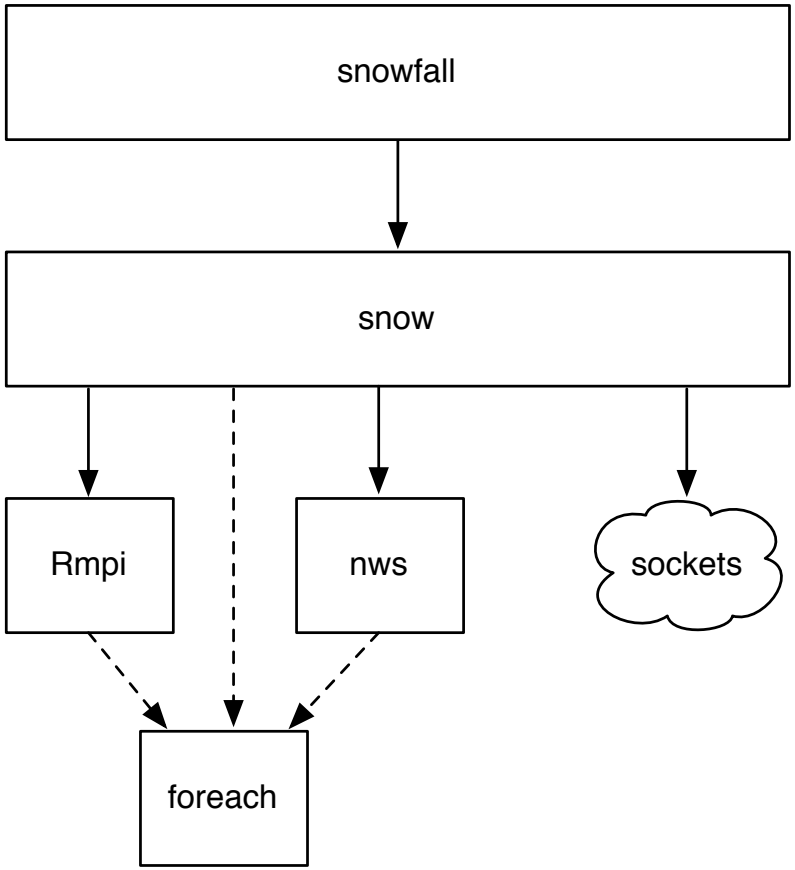
Some Basic Terminology

Since this is Los Angeles, here is fun fact:

In Los Angeles, officials pointed out that such terms as "master" and "slave" are unacceptable and offensive, and equipment manufacturers were asked not to use these terms. Some manufacturers changed the wording to primary / secondary (Source: CNN).

I apologize if I offend anyone, but I use `slave` and `worker` interchangeably depending on the documentation's terminology.

Explicit Parallelism in R



Message Passing Interface (MPI) Basics

MPI defines an environment where programs can run in parallel and communicate with each other by passing messages to each other. There are two major parts to MPI:

1. the environment programs run in
2. the library calls programs use to communicate

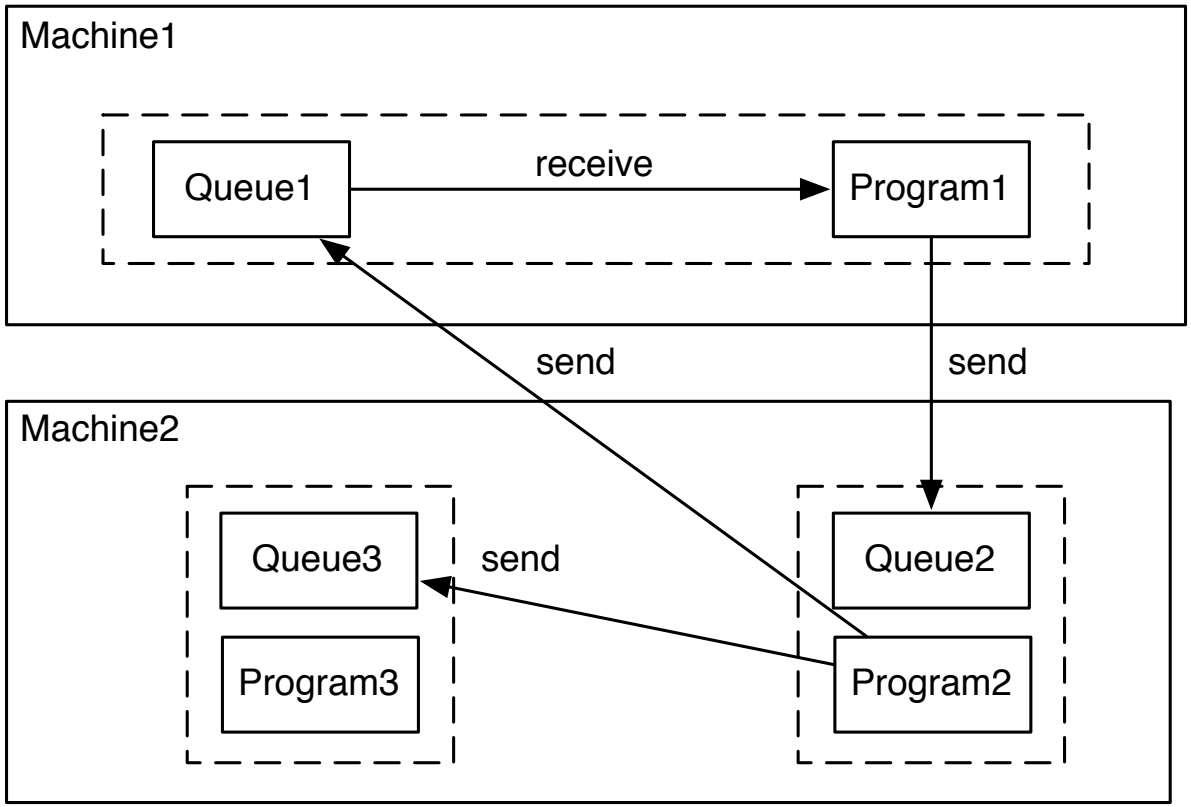
The first is usually controlled by a system administrator. We will discuss the second point above.

Message Passing Interface (MPI) Basics

MPI is so preferred by users because it is so simple to use. Programs just use *messages* to communicate with each other. Can you think of another successful systems that functions using message passing? (the Internet!)

- Each program has a mailbox (called a queue).
- Any program can put a message into the another program's mailbox.
- When a program is ready to process a message, it receives a message from its own mailbox and does something with it.

Message Passing Interface (MPI) Basics



MPI Message Passing Interface

The package `Rmpi` provides an R interface/wrapper to MPI such that users can write parallel programs in R rather than in C, C++ or Fortran. There are various flavors of MPI

1. MPICH and MPICH2
2. LAM-MPI
3. OpenMPI

Installing Rmpi and its Prerequisites

On MacOS X:

1. Install `lam7.1.3` from `http://www.lam-mpi.org/download/files/lam-7.1.3-1.dmg.gz`.
2. Edit the configuration file as we will see in a bit.
3. Build `Rmpi` from source, or use the Leopard CRAN binary.
4. To install from outside of R, use `sudo R CMD INSTALL Rmpi_xxx.tgz`.

NOTE: These instructions have not been updated for Snow Leopard!

Installing Rmpi and its Prerequisites

On Windows:

1. MPICH2 is the recommended flavor of MPI.
2. Instructions are also available for DeinoMPI.
3. Installation is too extensive to cover here. See instructions at `http://www.stats.uwo.ca/faculty/yu/Rmpi/windows.htm`

Installing Rmpi and its Prerequisites

Linux installation depends very much on your distribution and the breadth of its package repository.

1. Install `lam` using `sudo apt-get install liblam4` or equivalent.
2. Install `lam4-dev` (header files, includes etc.) using `sudo apt-get install lam4-dev` or equivalent.
3. Edit the configuration file as we will see in a bit.

Configuring LAM/MPI

1. LAM/MPI must be installed on all machines in the cluster.
2. Passwordless SSH should/must be configured on all machines in the cluster.
3. Create a blank configuration file, anywhere. It should contain the host names (or IPs) of the machines in the cluster and the number of CPUs on each. For my trivial example on one machine with 2 CPUs:

   ```
   localhost cpu=2
   ```

   For a more substantial cluster,

   ```
   euler.bytemining.com cpu=4
   riemann.bytemining.com cpu=2
   gauss.bytemining.com cpu=8
   ```

4. Type `lamboot <path to config file>` to start, and `lamhalt` to stop.

Rmpi Demonstration Code for your Reference

```
1  library(Rmpi)
2  # Spawn as many slaves as possible
3  mpi.spawn.Rslaves()
4  # Cleanup if R quits unexpectedly
5  .Last <- function(){
6      if (is.loaded("mpi_initialize")){
7          if (mpi.comm.size(1) > 0){
8              print("Please use mpi.close.Rslaves()
                   to close slaves.")
9              mpi.close.Rslaves()
10         }
11         print("Please use mpi.quit() to quit R")
12         .Call("mpi_finalize")
13     }
14 }
```

Rmpi Demonstration Code for your Reference

```
15 # Tell all slaves to return a message identifying
       themselves
16 mpi.remote.exec(paste("I am",mpi.comm.rank(),"of",
       mpi.comm.size()))
17
18 # Tell all slaves to close down, and exit the
       program
19 mpi.close.Rslaves()
20 mpi.quit()
```

`Rmpi` Demonstration

Demonstration time!

Rmpi Program Structure

The basic structure of a parallel program in R:

1. Load Rmpi and spawn the slaves.
2. Write any necessary functions.
3. Wrap all code the slaves should run into a worker function.
4. Initialize data.
5. Send all data and functions to the slaves.
6. Tell the slaves to execute their worker function.
7. Communicate with the slaves to perform the computation*.
8. Operate on the results.
9. Halt the slaves and quit.

* can be done in multiple ways

Communicating with the Slaves

Three ways to communicate with slaves.

1. **"Brute force"** : If the problem is "embarrassingly parallel", divide the problem into n subproblems and give the n slaves their data and functions. $n \leq$ the number of slaves.
2. **Task Push**: There are more tasks than slaves. Worker functions loop and retrieve messages from the master until there are no more messages to process. Tasks are piled at the slave end.
3. **Task Pull**: Good when the number of tasks $>>$ n. When a slave completes a task, it *asks the master* for another task. **Preferred.**

Example of Task Pull: Worker function

Task Pull: code walkthrough

Rmpi Demonstration

The Fibonacci numbers can be computed recursively using the following recurrence relation,

$$F_n = F_{n-1} + F_{n-2}, \quad n \in N^+$$

```
1  fib <- function(n)
2  {
3      if (n < 1)
4          stop("Input must be an integer >= 1")
5      if (n == 1 | n == 2)
6          1
7      else
8          fib(n-1) + fib(n-2)
9  }
```

A Snake in the Grass

1. Fibonacci using `sapply(1:25, fib)`

```
   user  system elapsed
  1.990   0.000   2.004
```

2. Fibonacci using `Rmpi`

```
   user  system elapsed
  0.050   0.010   2.343
```

But how can that be???

A Snake in the Grass

Is parallelization **always** a good idea? **NO!**

Setting up slaves, copying data and code is very costly performance-wise, especially if they must be copied across a network (such as in a cluster).

General Rule (in Ryan's words):
"Only parallelize with a certain method if the cost of computation is (much) greater than the cost of setting up the framework."

A Snake in the Grass

In an effort to be generalist, I used the Fibonacci example, but

$$T(\texttt{fib}) << T(\texttt{MPI setup})$$

so we do not achieve the expected boost in performance.

For a more comprehensive (but much more specific) example, see the **10-fold cross validation** example at `http://math.acadiau.ca/ACMMaC/Rmpi/examples.html`.

Example of Submitting an R Job to MPI

```
1  # Run the job.  Replace with whatever R script should run
2  mpirun -np 1 R --slave CMD BATCH task_pull.R
```

Wrapping it Up...The Code, not the Presentation!

`Rmpi` is flexible and versatile for those that are familiar with MPI and clusters in general. For everyone else, it may be frightening.

`Rmpi` requires the user to deal with several issues *explicitly*

1. sending data and functions etc. to slaves.
2. querying the master for more tasks
3. telling the master the slave is done

There must be a better way for the rest of us!

The snow Family

Snow and Friends:

1. `snow`
2. `snowFT`*
3. `snowfall`

* - no longer maintained.

`snow`: Simple Network of Workstations

Provides an interface to several parallelization packages:

1. **MPI**: Message Passing Interface, via `Rmpi`
2. **NWS**: NetWork Spaces via `nws`
3. **PVM**: Parallel Virtual Machine
4. **Sockets** via the operating system

All of these systems allow *intra*system communication for working with multiple CPUs, or *inter*system communication for working with a cluster.

snow: Simple Network of Workstations

Most of our discussion will be on `snowfall`, a wrapper to `snow`, but the API of `snow` is fairly simple.

1. `makeCluster` sets up the cluster and initializes its use with R and returns a cluster object.
2. `clusterCall` calls a specified function with identical arguments on each node in the cluster.
3. `clusterApply` takes a cluster, sequence of arguments, and a function and calls the function with the first element of the list on first node, second element on second node. At most, the number of elements must be the number of nodes.
4. `clusterApplyLB` same as above, with load balancing.

snow: Simple Network of Workstations

Most of our discussion will be on `snowfall`, a wrapper to `snow`, but the API of `snow` is fairly simple.

1. `clusterExport` assigns the global values on the master to variables of the same names in global environments of each node.
2. `parApply` parallel apply using the cluster.
3. `parCapply`, `parRapply`, `parLapply`, `parSapply` parallel versions of column `apply`, row `apply` and the other `apply` variants.
4. `parMM` for parallel matrix multiplication.

snowfall

snowfall adds an abstraction layer (not a technical layer) over the snow package for better usability. snowfall works with snow commands.

For computer scientists, snowfall is a *wrapper* to snow.

snowfall

The point to take home about snowfall, before we move further:

snowfall uses list functions for parallelization. Calculations are distributed onto workers where each worker gets a portion of the full data to work with.

Bootstrapping and cross-validation are two statistical methods that are trivial to implement in parallel.

snowfall Features

The snowfall API is very similar to the snow API and has the following features

1. all wrapper functions contain extended error handling.
2. functions for loading packages and sources in the cluster
3. functions for exchanging variables between cluster nodes.
4. changing cluster settings does not require changing R code. Can be done from command line.
5. all functions work in sequential mode as well. Switching between modes requires no change in R code.
6. sfCluster

Getting Started with `snowfall`

The `snowfall` API is very similar to the `snow` API and has the following features

1. Install R, `snow`, `snowfall`, `rlecuyer` (a network random number generator) on all hosts (masters and slaves).
2. Enable passwordless login via SSH on all machines (including the master!). This is trivial, and there are tutorials on the web. This varies by operating system.
3. If you want to use a cluster, install the necessary software and R packages.
4. If you want to use `sfCluster` (LAM-MPI only), install LAM-MPI first as well as `Rmpi` and configure your MPI cluster.*
5. If you want to use `sfCluster`, install the `sfCluster` tool (a Perl script).*

* = more on this in a bit!

The Structure of a `snowfall` Program

1. Initialize with `sfInit`. If using `sfCluster`, pass no parameters.
2. Load the data and prepare the data for parallel calculation.
3. Wrap computation code into a function that can be called by an R list function.
4. Export objects needed in the calculation to cluster nodes and load the necessary packages on them.
5. Start a network random number generator to ensure that each node produces random numbers independently. (optional)
6. Distribute the calculation to the cluster using a parallel list function.
7. Stop the cluster.

`snowfall` Example

For this example/demo we will look at the pneumonia status of a random sample of people admitted to an ICU. We will use the bootstrap to obtain an estimate of a regression coefficient based on "subdistribution functions in competing risks."

snowfall Example: Step 1

```
1  #Load necessary libraries
2  library(snowfall)
3
4  #Initialize
5  sfInit(parallel=TRUE, cpus=2, type="SOCK", socketHosts=
       rep("localhost",2))
6  #parallel=FALSE runs code in sequential mode.
7  #cpus=n, the number of CPUs to use.
8  #type can be "SOCK" for socket cluster, "MPI", "PVM" or "
       NWS"
9  #socketHosts is used to specify hostnames/IPs for
10 #remote socket hosts in "SOCK" mode only.
```

snowfall Example: Step 2/3

```
11  #Load the data.
12  require(mvna)
13  data(sir.adm)
14  #using a canned dataset.
15
16  #create a wrapper that can be called by a list function.
17  wrapper <- function(idx) {
18       index <- sample(1:nrow(sir.adm), replace=TRUE)
19       temp <- sir.adm[index, ]
20       fit <- crr(temp$time, temp$status, temp$pneu)
21       return(fit$coef)
22  }
```

snowfall Example: Step 4

```
24 #export needed data to the workers and load packages on
       them.
25 sfExport("sir.adm") #export the data (multiple objects in
       a list)
26 sfLibrary(cmprsk) #force load the package (must be
       installed)
```

snowfall Example: Steps 5, 6, 7

```
27  #STEP 5: start the network random number generator
28  sfClusterSetupRNG()
29  #STEP 6: distribute the calculation
30  result <- sfLapply(1:1000, wrapper)
31  #return value of sfLapply is always a list!
32
33  #STEP 7: stop snowfall
34  sfStop()
```

This is CRUCIAL. If you need to make changes to the cluster, stop it first and then re-initialize. Stopping the cluster when finished ensures that there are no processes going rogue in the background.

`snowfall` Demonstration

For this demonstration, suppose Ryan's Ubuntu server has a split personality (actually it has several) and that its MacOS X side represents one node, and its Ubuntu Lucid Lynx side represents a second node. They communicate via sockets.

Demonstration here.

`snowfall`: Precomputed Performance Comparison

Using a dual-core Ubuntu system yielded the following timings:

1. Sequential, using `sapply(1:1000, wrapper)`

```
   user  system elapsed
 98.970   0.000   99.042
```

2. Parallel, using `sfLapply(1:1000, wrapper)`

```
   user  system elapsed
  0.170   0.180   50.138
```

snowfall and the Command Line

snowfall can also be activated from the command-line.

```
#Start a socket cluster on localhost with 3 processors.
R CMD BATCH myProgram.R --args --parallel --cpus=3
#myProgram.R is the parallel program and --parallel
    specifies parallel mode.

#Start a socket cluster with 6 cores (2 on localhost, 3
    on "a" and 1 on "b")
R --args --parallel --hosts=localhost:2,a:3,b:1 <
    myProgram.R

#Start snowfall for use with MPI on 4 cores on R
    interactive shell.
R --args --parallel --type=MPI --cpus=4
```

snowfall's Other Perks

snowfall has other cool features that you can explore on your own...

`snowfall`'s Other Perks: Load Balancing

In a cluster where all nodes have comparable performance specifications, each node will take approximately the same time to run.

When the cluster contains many different machines, speed depends on the *slowest* machine, so faster machines have to wait for its slow poke friend to catch up before continuing. **Load balancing** can help resolve this discrepancy.

By using `sfClusterApplyLB` in place of `sfLapply`, faster machines are given further work without having to wait for the slowest machine to finish.

`snowfall`'s Other Perks: Saving and Restoring Intermediate Results

Using `sfClusterApplySR` in place of `sfLapply` allows long running clusters to save intermediate results after processing n indices (n is a number of CPUs). These results are stored in a temporary path. For more information, use `?sfClusterApplySR`.

snowfall apply

By the way, while we are discussing a bunch of different functions...

There are other apply variants in snowfall such as

```
sfLapply( x, fun, ... )
sfSapply( x, fun, ..., simplify = TRUE, USE.NAMES = TRUE )
sfApply( x, margin, fun, ... )
sfRapply( x, fun, ... )
sfCapply( x, fun, ... )
```

The documentation simply chooses to use sfLapply for consistency.

snowfall and snowFT

snowFT is the fault tolerance extension for snow, but it is *not* available in snowfall. The author states that this is due to the lack of an MPI implementation of snowFT.

But you're not missing much as snowFT does not appear to be maintained any longer.

snowfall and sfCluster

`sfCluster` is a Unix command-line tool written in Perl that serves as a management system for cluster and cluster programs. It handles cluster setup, as well as monitoring performance and shutdown. **With `sfCluster`, these operations are hidden from the user.**

Currently, `sfCluster` works only with LAM-MPI.

`sfCluster` can be downloaded from

`http://www.imbi.uni-freiburg.de/parallel/files/sfCluster_040.tar.gz`.

snowfall and sfCluster

1. `sfCluster` must be installed on every participating node in the cluster.
2. Specifying the machines in the cluster is done in a hostfile, which is then passed to LAM/MPI!
3. Call `sfNodeInstall --all` afterwards.
4. More options are in the `README` shipped with `sfCluster`.

sfCluster Features

sfCluster wants to make sure that your cluster can meet your needs without exhausting resources.

1. It checks your cluster to find machines with free resources if available. The available machines (*or even the available* **parts** *of the machine*) are built into a new sub-cluster which belongs to the new program. This is implemented with LAM sessions.
2. It can optionally monitor the cluster for usage and stop programs if they exceed their memory allotment, disk allotment, or if machines start to swap.
3. It also provides diagnostic information about the current running clusters and free resources on the cluster including PIDs, memory use and runtime.

`sfCluster` is Preventative Medicine for your Jobs

`sfCluster` execution follows these steps

1. Tests memory usage by running your program in sequential mode for 10 minutes to determine the maximum amount of RAM necessary on the slaves.
2. Detect available resources in the cluster and form a sub-cluster.
3. Start LAM/MPI cluster using this sub-cluster of machines.
4. Run R with parameters for `snowfall` control.
5. Do over and over: Monitor execution and optionally display state of the cluster or write to logfiles.
6. Shutdown cluster on fail, or on normal end.

Some Example Calls to sfCluster

```
#Run an R program with 10 CPUs and a max of 1GB of
    RAM in monitoring mode.
sfCluster -m --cpus=10 --mem=1G myProgram.R

#Run nonstopping cluster with quiet output
nohup sfCluster -b --cpus=10 --mem=1G myProgram.R
    --quiet

#Start R interactive shell with 8 cores and 500MB
    of RAM.
sfCluster -i --cpus=8 --mem=500
```

sfCluster Administration

```
jo@biom9:~$ sfCluster -o
SESSION          | STATE |  M | MASTER       #N   RUNTIME R-FILE / R-OUT
-----------------+-------+----+-------------------------------------------------
LrtpdV7T_R-2.8.1 | run   | MO | biom9.imbi    9   2:46:51 coxBst081223.R / coxBst081223.Rout
baYwQ0GB_R-2.5.1 | run   | IN | biom9.imbi    2   0:00:18 -undef- / -undef-

jo@biom9:~$ sfCluster -o --all
SESSION          | STATE | USR |  M | MASTER       #N   RUNTIME R-FILE / R-OUT
-----------------+-------+-----+----+-------------------------------------------------
LrtpdV7T_R-2.8.1 | run   | jo  | MO | biom9.imbi    9   3:16:09 coxBst081223.R / coxBst081223.Rout
jlXUhxtP_R-2.5.1 | run   | jo  | IN | biom9.imbi    2   0:00:22 -undef- / -undef-
bSpNLNhd_R-2.7.2 | run   | cp  | BA | biom9.imbi    8   0:32:57 getPoints11.R / getPoints11.Rout
NPS5QHkK_R-2.7.2 | run   | cp  | MO | biom9.imbi   10   3:50:42 box2.R / box2.Rout

jo@biom9:~$ sfCluster --universe --mem=1G
Assumed memuse: 1024M (use '--mem' to change).

Node                          | Max-Load | CPUs | RAM    | Free-Load | Free-RAM | FREE-TOTAL
------------------------------+----------+------+--------+-----------+----------+-----------
biom8.imbi.uni-freiburg.de    |        8 |    8 | 15.9G  |     0     |    9.3G  |     0
biom9.imbi.uni-freiburg.de    |        8 |    8 | 15.9G  |     0     |   12.6G  |     0
biom10.imbi.uni-freiburg.de   |        8 |    8 | 15.9G  |     0     |   14.0G  |     0
biom12.imbi.uni-freiburg.de   |        2 |    4 |  7.9G  |     0     |    5.8G  |     0
knecht5.fdm.uni-freiburg.de   |        8 |    8 | 15.7G  |     1     |    1.2G  |     1
knecht4.fdm.uni-freiburg.de   |        8 |    8 | 15.7G  |     1     |    4.3G  |     1
knecht3.fdm.uni-freiburg.de   |        5 |    8 | 15.7G  |     3     |   11.1G  |     3
biom6.imbi.uni-freiburg.de    | no-sched |    4 |  7.9G  |     -     |      -   |     -
biom7.imbi.uni-freiburg.de    |        2 |    4 |  7.9G  |     1     |    2.1G  |     1

Potential usable CPUs: 6
```

Overview of running clusters in `sfCluster` with other important information.

Why Not use *x* for Explicit Parallel Computing?

	Learnability	Efficiency	Memorability	Errors	Satisfaction
RmpiR	+	−−	++	+	+
rpvm	−−	−−	+	−	−−
nws	+	+	++	+	+
snow	+	++	++	+	++
snowFT	+	++	++	+	++
snowfall	++	++	++	+	++
papply	+	+	++	+	0
biopara	−−	−−	0	0	−−
taskPR	+	−	++	0	−

borrowed from *State-of-the-art in Parallel Computing with R*, from *Journal of Statistical Software* August 2009, Volume 31, Issue 1.

Implicit Parallelism

Unlike explicit parallelism where the user controls (and can mess up) most of the cluster settings, with implicit parallelism most of the messy legwork in setting up the system and distributing data is abstracted away.

Most users probably prefer implicit parallelism.

multicore

multicore provides functions for parallel execution of R code on systems with multiple cores or multiple CPUs. Distinct from other parallelism solutions, multicore jobs all share the same state when spawned.

The main functions in multicore are

- mclapply, a parallel version of lapply.
- parallel, do something in a separate process.
- collect, get the results from call(s) to parallel

multicore: mclapply

`mclapply` is a parallel version of `lapply`. Works similar to `lapply`, but has some extra parameters:

```
mclapply(X, FUN, ..., mc.preschedule = TRUE, mc.set.seed = TRUE,
        mc.silent = FALSE, mc.cores = getOption("cores"))
```

1. `mc.preschedule=TRUE` controls how data are allocated to jobs/cores.
2. `mc.cores` controls the **maximum** number of processes to spawn.
3. `mc.silent=TRUE` suppresses standard output (informational messages) from each process, but not errors (`stderr`).
4. `mc.set.seed=TRUE` sets the processes' seeds to something unique, otherwise it is copied with the other state information.

multicore: Pre-Scheduling

`mc.preschedule` controls how data are allocated to processes.

- if `TRUE`, then the data is divided into n sections and passed to n processes (n is the number of cores to use).
- if `FALSE`, then a job is constructed for each data value sequentially, up to n at a time.

The author provides some advice on whether to use `TRUE` or `FALSE` for `mc.preschedule`.

- `TRUE` is better for short computations or large number of values in `X`.
- `FALSE` is better for jobs that have high variance of completion time and not too many values of `X`.

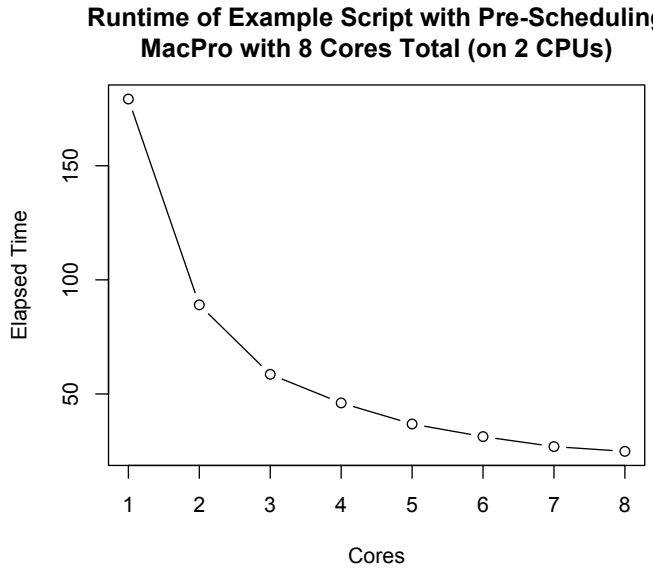
Example: mclapply

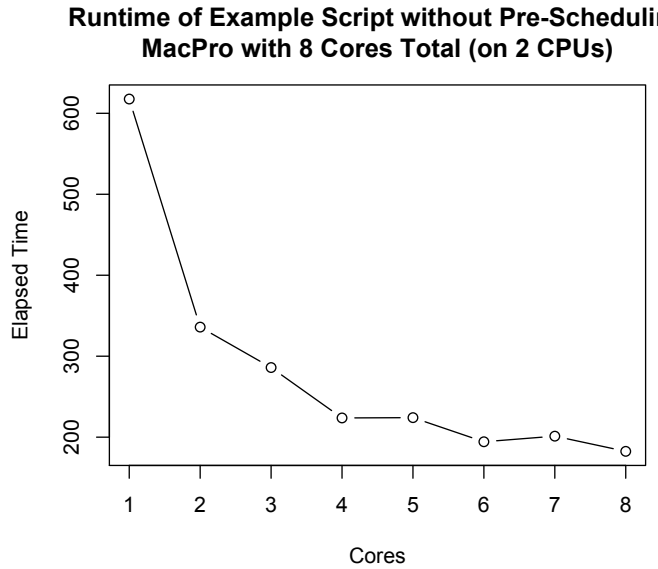
Let's use mclapply to convert about 10,000 lines of text (tweets) to JSON.

```
1  library(multicore)
2  library(rjson)
3  twitter <- readLines("twitter.dat")
4  mclapply(twitter,
5      FUN=function(x) {
6              return(fromJSON(x))
7          },
8          mc.preschedule=TRUE, mc.cores=8)
```

Example: `mclapply` Performance

For this example, processing time improves dramatically using all 8-cores.

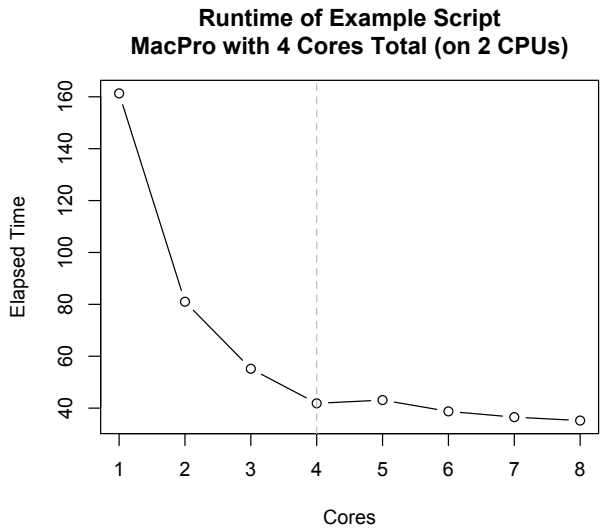


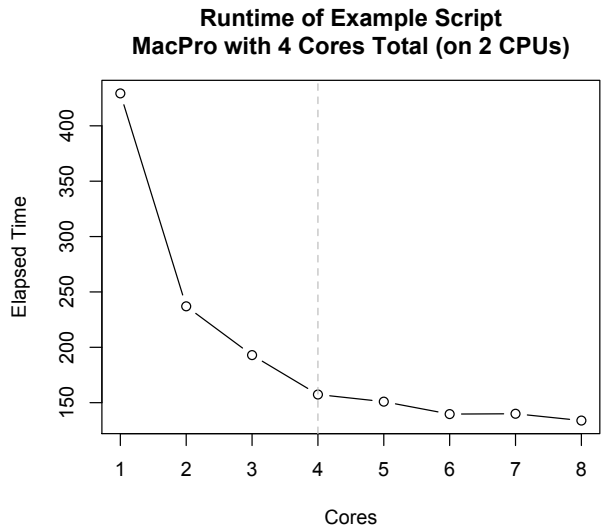


Notice that using pre-scheduling yields superior performance here, decreasing processing time from 3 minutes to 25 seconds.

Example: `mclapply` Performance

So, the higher `mc.cores` is, the better, right? **NO!** On a 4-core system, note that performance gain past 4 cores is negligible.





It is not a good idea to set `mc.cores` higher than the number of cores in the system. Setting it too high will "fork bomb" the system.

parallel and collect

`parallel` (or `mcparallel`) allows us to create an external process that does something. After hitting enter, R will not wait for the call to finish.

```
parallel(expr, name, mc.set.seed = FALSE, silent = FALSE)
```

This function takes as parameters:

- some expression to be evaluated, `expr`.
- an optional name for the job, `name`.
- miscellaneous parameters `mc.set.seed` and `silent`.

parallel and collect

`collect` allows us to retrieve the results of the spawned processes.

```
collect(jobs, wait = TRUE, timeout = 0, intermediate = FALSE)
```

This function takes as parameters:

- `jobs`:
  - a list of variable names that were bound using `parallel`.
  - an integer vector of Process IDs (PID). We can get the PID by typing the variable associated with the process and hitting ENTER.
- whether or not to `wait` for the processes in `jobs` to end. If `FALSE`, will check for results in `timeout` seconds.
- a function, `intermediate`, to execute while waiting for results.

Demonstration: `parallel` and `collect`

As a quick example, consider a silly loop that simply keeps the CPU busy incrementing a variable `i` for `duration` iterations.

```
1  my.silly.loop <- function(j, duration) {
2       i <- 0
3       while (i < duration) {
4            i <- i + 1
5       }
6       #When done, return TRUE to indicate that this
            function does *something*
7       return(paste("Silly", j, "Done"))
8  }
9  silly.1 <- parallel(my.silly.loop(1, 10000000))
10 silly.2 <- parallel(my.silly.loop(2, 5000000))
11 collect(list(silly.1, silly.2))
```

Trivially, `silly.2` will finish first, but we will wait (block) for both jobs to finish.

Demonstration: `parallel` and `collect`

Demonstration here.

Demonstration: `parallel` and `collect`

We can also be impatient and only wait s seconds for a result, and then move along with our lives. If the jobs exceed this time limit, we must poll for the results ourselves. We set `wait=FALSE` and provide `timeout=2` for a 2 second delay. (Yes, I am impatient)

```
1  silly.1 <- parallel(my.silly.loop(1, 10000000))
2  silly.2 <- parallel(my.silly.loop(2, 5000000))
3  collect(list(silly.1, silly.2), wait=FALSE, timeout=2)
4  #we can get the results later by calling,
5  collect(list(silly.1, silly.2))
```

Trivially, `silly.2` will finish first, but we will wait (block) for both jobs to finish.

Demonstration: `parallel` and `collect`

Demonstration here.

Demonstration: parallel and collect

We can ask R to execute some function while we wait for a result.

```
1  status <- function(results.so.far) {
2      jobs.completed <- sum(unlist(lapply(results.so.far,
           FUN=function(x) { !is.null(x) })))
3      print(paste(jobs.completed, "jobs completed so far.")
           )
4  }
5  silly.1 <- parallel(my.silly.loop(1, 10000000))
6  silly.2 <- parallel(my.silly.loop(2, 5000000))
7  results <- collect(list(silly.1, silly.2), intermediate=
       status)
```

Trivially, `silly.2` will finish first, but we will wait (block) for both jobs to finish.

Demonstration: `parallel` and `collect`

Demonstration here.

multicore Gotchas

Weird things can happen when we spawn off processes.

1. **never** use any on-screen devices or GUI elements in the expressions passed to `parallel`. Strange things will happen.
2. `CTRL+C` will interrupt the parent process running in R, but orphans the child processes that were spawned. If this happens, we must clean up after ourselves by killing off the children and then garbage collecting.

```
1  kill(children())
2  collect()
```

   Child processes that have finished or died will remain until they are collected!
3. I am going to hell for the previous bullet...

Other `multicore` Features

`multicore` is a pretty small package, but there are some things that we did not touch on. In this case, it is a good idea to not discuss them...

1. low-level function, should you ever need them: `fork`, `sendMaster`, `process`. Tread lightly though, you know what they say about curiosity...
2. the user can send `signals` to child processes, but this is rarely necessary.
3. **Limitation!** Cannot be used on Windows because it lacks the `fork` system call. Sorry. However, there is an experimental version that works with Windows, but it is not stable.

doMC/foreach

`foreach` allows the user to iterate through a set of values parallel.

- By default, iteration is sequential, unless we use a package such as `doMC` which is a parallel backend for `foreach`.
- `doMC` is an interface between `foreach` and `multicore`.
- Windows is supported via an experimental version of `multicore`.
- only runs on one system. To use on a cluster, can use the `doNWS` package from REvolution Computing.

doMC/foreach Usage

1. Load the `doMC` and `foreach` libraries.
2. **MUST** register the parallel backend: `registerDoMC()`.

doMC/foreach Usage

A `foreach` is an *object* and has the following syntax and parameters.

```
foreach(..., .combine, .init, .final=NULL, .inorder=TRUE,
      .multicombine=FALSE, .maxcombine=if (.multicombine) 100 else 2,
      .errorhandling=c('stop', 'remove', 'pass'), .packages=NULL,
      .export=NULL, .noexport=NULL, .verbose=FALSE)
      when(cond)
     e1 %:% e2
     obj %do% ex
     obj %dopar% ex
     times(n)
```

- `...` controls how `ex` is evaluated. This can be an *iterator* such as `icount(n)`, that counts from 1 to n.
- `.combine` is the action used to combine results from each iteration. `rbind` will append rows to a matrix, for example. Can also use arithmetic operations, or write your own function. By default, output is returned as a list.

doMC/foreach Usage

A `foreach` is an *object* and has the following syntax and parameters.

```
foreach(..., .combine, .init, .final=NULL, .inorder=TRUE,
      .multicombine=FALSE, .maxcombine=if (.multicombine) 100 else 2,
      .errorhandling=c('stop', 'remove', 'pass'), .packages=NULL,
      .export=NULL, .noexport=NULL, .verbose=FALSE)
      when(cond)
     e1 %:% e2
     obj %do% ex
     obj %dopar% ex
     times(n)
```

- `.final` is a function to call once all results have been collected. For example, you may want to convert to a `data.frame`.
- `.inorder=TRUE` will return the results in the same order that they were submitted. Setting to `FALSE` gives better performance.
- `.errorhandling` specifies what to do if a task encounters an error. `stop` kills the entire job, `remove` ignores the input that caused the error, or `pass` just ignores the error and returns the result.

doMC/foreach Usage

A `foreach` is an *object* and has the following syntax and parameters.

```
foreach(..., .combine, .init, .final=NULL, .inorder=TRUE,
      .multicombine=FALSE, .maxcombine=if (.multicombine) 100 else 2,
      .errorhandling=c('stop', 'remove', 'pass'), .packages=NULL,
      .export=NULL, .noexport=NULL, .verbose=FALSE)
      when(cond)
     e1 %:% e2
     obj %do% ex
     obj %dopar% ex
     times(n)
```

- `.packages` specifies a vector of package names that each worker needs to load.
- `.verbose=TRUE` can be useful for troubleshooting.
- For the other parameters, check out the documentation for `foreach`.

doMC/foreach Usage

A `foreach` is an *object* and has the following syntax and parameters.

```
foreach(...) when(cond) %dopar% ex times(n)
```

- `when(cond)` causes the loop to execute only if `cond` evaluates to `TRUE`.
- %dopar% causes `ex` to execute in parallel. Can replace with %do% to execute in sequence (for debugging).
- `times(n)` executes the entire statement *n* times.
- Can also nest `foreach` statements using syntax like `foreach(...) %:% foreach(...)`.

doMC/foreach Example of a Nested for Loop

```
1  sim <- function(a, b) { return(30*a + b**) }
2  avec <- seq(1,100); bvec <- seq(5,500,by=5)
3  x <- matrix(0,length(avec),length(bvec))
4  for (j in 1:length(bvec)) {
5       for (i in 1:length(avec)) {
6            x[i, j] <- sim(avec[i], bvec[j])
7       }
8  }
9  method.1 <- x
10
11 #with foreach
12 x <- foreach(b = bvec, .combine = "cbind") %:%
13              foreach(a = avec, .combine = "c") %dopar% {
14                  sim(a,b)
15          }
16 method.2 <- x
17 all(method.1 == method.2)    #both loops yield the same
       result.
```

doMC/foreach Example of a Nested for Loop

The example was trivial because each iteration performs a negligible amount of work! If we were to replace %do% with %dopar%, we would actually get *longer* processing time, which brings me to this point...

"Each iteration should execute computationally-intensive work. Scheduling tasks has overhead, and can exceed the time to complete the work itself for small jobs."

foreach Demonstration: Bootstrapping

Here is a better example. How long does it take to run 10,000 bootstrap iterations (in parallel) on a 2-core MacBook Pro? What about an 8-core MacPro?

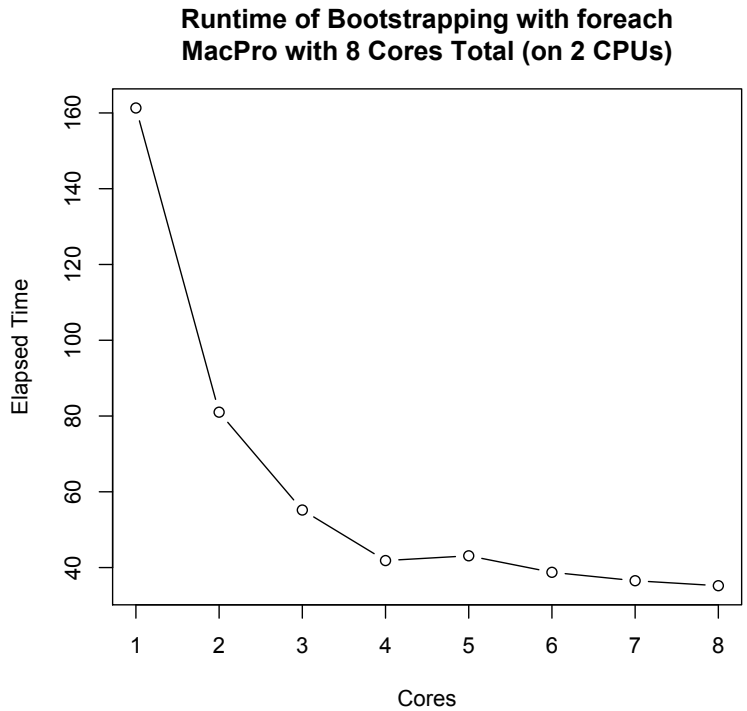
```
1  data(iris)
2
3  iris.sub <- iris[which(iris[, 5] != "setosa"), c(1,5)]
4  trials <- 10000
5  result <- foreach(icount(trials), .combine=cbind) %dopar%
       {
6      indices <- sample(100, 100, replace=TRUE)
7      glm.result <- glm(iris.sub[indices, 2]~iris.sub[
           indices, 1], family=binomial("logit"))
8      coefficients(glm.result)    #this is the result!
9  }
```

`foreach` Demonstration: Bootstrapping

Demonstration on 2-core server, here, if time.

`foreach` Demonstration: Bootstrapping Performance

On an 8-core system, the processing time for this operation decreased from 59s using one core, to about 15s using 8-cores. This is about a 4x speedup. We may expect an 8x speedup, but the improvement is dependent on many variables including system load, CPU specifications, operating system, etc.



foreach Nerd Alert: quicksort

If you are interested, here is an implementation of quicksort in R, using foreach!

```
1  qsort <- function(x) {
2    n <- length(x)
3    if (n == 0) {
4      x
5    } else {
6      p <- sample(n, 1)
7      smaller <- foreach(y=x[-p], .combine=c) %:% when(y <=
           x[p]) %do% y
8      larger  <- foreach(y=x[-p], .combine=c) %:% when(y >
           x[p]) %do% y
9      c(qsort(smaller), x[p], qsort(larger))
10   }
11 }
12 qsort(runif(12))
```

`foreach` Additional Features

`foreach`, and related packages, provide a lot more functionality that may be of interest.

- `foreach` can be used on a cluster by using a different parallel backend. `doMC` uses `multicore` whereas `doNWS` and `doSNOW` use `nws` and `snow` respectively. These are maintained by Revolution Computing.
- in this talk we assumed that we iterate over integers/counters. We can also iterate over objects such as elements in a vector by using custom iterators, using the `iterators` package.

GPUs: Towards the Future

GPU = Graphics Processing Unit

GPUs power video cards and draw pretty pictures on the screen.

They are also VERY parallel, very fast, cheap (debatable), and low-power. However, they suffer from low bandwidth.

GPUs: Towards the Future

In 2008, the fastest computer in the world was the PlayStation.

GPUs: Towards the Future

Here are just a few numbers for you.

1. PC CPU: 1-3Gflop/s average
2. GPU: 100Gflop/s average.

GPUs: Towards the Future

High level languages such as CUDA and Brook+ exist for interacting with a GPU with available libraries including BLAS, FFT, sorting, sparse multiply etc.

But still hard to use with complicated algorithms because...

Pitfall: Data transfer is very costly and slow into GPU space.

GPUs: Towards the Future

There are two packages currently for working with GPUs in R and your mileage may vary.

1. `gputools` provides R interfaces to some common statistical algorithms using Nvidia's CUDA language and its CUBLAS library as well as EMI Photonics' CULA libraries.
2. `cudaBayesreg` provides a CUDA parallel implementation of a Bayesian Multilevel Model for fMRI data analysis.
3. More to come!

In Conclusion

1. R provides several ways to parallelize tasks.
2. They are pretty easy to use.
3. They do not fall victim to the Swiss Cheese Phenomenon under typical usage.
4. Worth learning as more and more packages begin to suggest them (`tm` and some other NLP packages).

In Conclusion: Other Explicit Parallelism Packages

There are too many HPC packages to mention in one (even two) session. Here is a list of others that may be of interest:

1. `rpvm`, no longer maintained
2. `nws`, NetWorkSpaces: parallelism for scripting languages.
3. `biopara`, socket based, fault tolerance, load balancing.
4. `taskPR` parallel execution of tasks with LAM/MPI.
5. `Rdsm` threads-like parallel execution.

In Conclusion: Other Implicit Parallelism Packages

There are too many HPC packages to mention in one (even two) session. Here is a list of others that may be of interest:

1. `pnmath` uses OpenMP.
2. `fork`, mimics Unix fork command.

A Caveat

Q: But what if I want my tasks to **run** faster?
A: Parallelization does not accomplish that.

1. Throw money at faster hardware.
2. Refactor your code.
3. Interface R with C, C++ or FORTRAN.
4. Interface with C and a parallelism toolkit like OpenMP.
5. Use a different language altogether (functional languages like Clojure, based on Lisp, and Scala are emerging as popular).

Next Time

Next time we will focus more on issues with Big Data and data that does not fit in available RAM. Some of the content will overlap with our talk today.

1. Using Map/Reduce via `mapReduce`, Hadoop, `rhipe` and `HadoopStreaming`
2. `bigmemory`
3. `ff`
4. Some discussion of sparse matrices.

Next Time

Next time we will focus more on issues with Big Data and data that does not fit in available RAM. Some of the content will overlap with our talk today.

1. Using Map/Reduce via `mapReduce`, Hadoop, `rhipe` and `HadoopStreaming`
2. `bigmemory`
3. `ff`
4. Some discussion of sparse matrices.

Keep in Touch!

My email: `ryan@stat.ucla.edu`

My blog: `http://www.bytemining.com`

Follow me on Twitter: `@datajunkie`

The End

Questions?

Thank You!

References

Most examples and descriptions can from package documentation. Other references:

1. "Easier Parallel Computing in R with snowfall and sfCluster", The R Journal Vol. 1/1, May 2009. Author unknown.
2. "Developing parallel programs using snowfall" by Jochen Knaus, March 4, 2010.
3. "Tutorial: Parallel computing using R package snowfall" by Knaus and Porzelius: June 29, 2009.
4. "Getting Started with doMC and foreach" by Steve Weston: June 9, 2010.
5. "Parallel computing in R with sfCluster/snowfall", `http://www.imbi.uni-freiburg.de/parallel/`.
6. "snow Simplified", `http://www.sfu.ca/~sblay/R/snow.html`.