Los Angeles R Users' Group

Parallelization in R, Revisited

Ryan R. Rosario

April 17, 2012

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 purely another language like C, FORTRAN, or a JVM based language (Java, Clojure/Incanter, Scala etc.)
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 a few HPC packages in R. 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. **Big Data and Distributed Computing:** using resources on a cluster of machines for data processing.

Disclaimer

1. There is a **ton** of material here. I provide a lot of slides for your reference.
2. Some of these methods will not work in RStudio, even with R 2.14.
3. I intend for this talk to be more high-level than my previous talk.
4. All experiments were run on two different systems for non-scientific comparison:
   - Dual Intel Xeon E5600 2.4GHz (2x6-core, 2011), 96GB DDR3 running Ubuntu 10.04LTS (Lucid Lynx) [**12 Core, 24 threads**]
   - Quad and 8-core MacPro (late 2007, early 2009) with 4GB DDR3 running Mac OS X 10.6 (Snow Leopard). (For original presentation)

What is Parallelism?

In computer science, *parallelism* is performing two or more tasks simultaneously (at the *exact same time*). That is, two or more processes work *in parallel*.

What is Parallelism?

The following are NOT examples of parallelism:

1. Having multiple programs open at the same time and doing their own processing.
2. **Analogy:** Watching two television shows on two different televisions at the same time.

(In the above, there is an implicit *context switch* between the processes.)

What is Parallelism?

The following are examples of parallelism:

1. Displaying a progress bar in a GUI while running a Gibbs sampler.
2. **Analogy:** Rubbing your stomach and patting your head at the same time.

(No implicit context switch... actions are simultaneous)

Parallelism

Parallelism means running several computations at the *exact* same time and taking advantage of multiple cores or CPUs on a single system, or CPUs on other systems (distributed). 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. `parallel`, part of R 2.14 base.
2. `RHadoop`, by Revolution Computing.
3. Very quick mention of GPUs.
4. `foreach` by Revolution Computing is provided in an appendix.

There are many others, but the above are the easiest to use and should be useful in most situations.

Parallelism in R

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

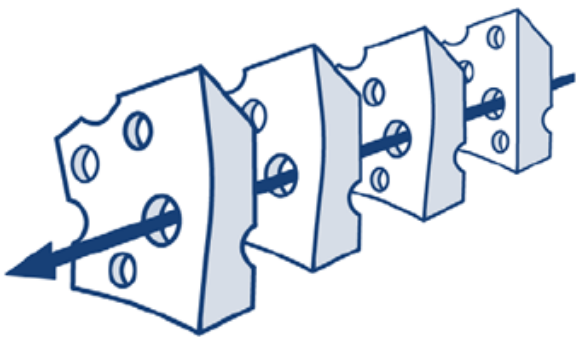
Many of these packages are simply wrappers around one or multiple other parallelism packages forming a complex and sometimes confusing web of packages. `parallel` attempts to eliminate some of this by wrapping `snow` and `multicore` into a nice bundle.

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 desktops have one processor with between ~~one~~ two and ~~four~~ six 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.

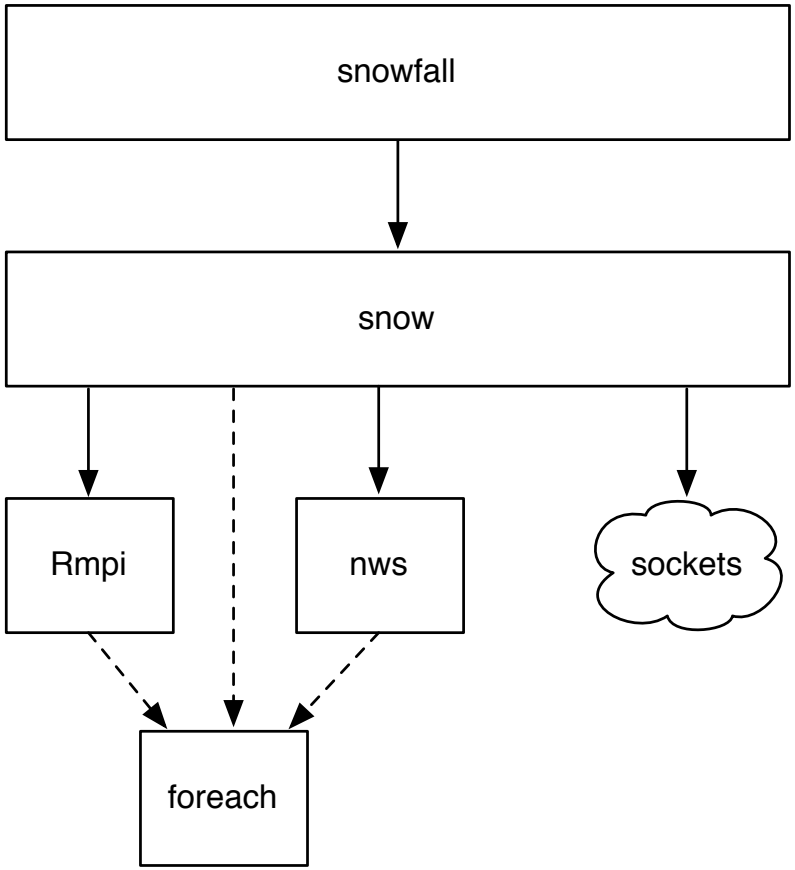
How Many Cores do I Have?

```
library(parallel)
mc <- detectCores()
mc
[1] 24
```

As the author mentions, this is a slippery slope and depends on not only your CPUs, but also the operating system.

- Windows will report the number of logical CPUs, which may exceed the number of physical cores.
- The OS may (not) take hyper-threading into account.

Explicit Parallelism in R



The `parallel` Package

A first release of the new base `parallel` package is distributed with R 2.14:

1. modified version of `snow`.
2. `multicore` (except on Windows)

`snow` Functionality: Simple Network of Workstations

Provides an interface to several parallelization and clustering 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 Functionality: Simple Network of Workstations

Note: The first version of the modified snow functionality does not support NWS, PVM or MPI clusters.

However...

The vignette specifies that NWS, PVM and MPI *should* work. YMMV...

Creating snow-like Clusters

`parallel` provides two ways to create `snow`-like clusters:

1. `makePSOCKcluster` uses `Rscript` to launch several instances of R either locally, or on other machines.
2. `makeForkCluster` uses the OS level `fork` call to create multiple *identical* R processes on the same machine with a copy of the master workspace.
3. `makeCluster` creates PVM, MPI or NWS clusters by calling `snow`. YMMV. (By default, uses sockets).

parallel: A Bit of snow

Major functions in API.

1. `makeCluster` sets up the cluster and initializes its use with R and returns a cluster object. (also, `makePSOCKcluster`, `makeForkCluster`.)
2. `clusterExport` takes a character vector or object names and exports the objects corresponding to the names to the cluster.
3. `clusterEvalQ` performs some operation (e.g. command, loading library, function call etc.) on the cluster and returns the results as a list.

There are several others for you to peruse at your leisure. Also, some of the them make more sense in `multicore` and are not hyped in the documentation.

`snow`: Simple Network of Workstations

A couple of other functions worth seeing.

1. `parRapply` a parallel row `apply`.
2. `parLapply` We will return to this one...
3. Despite its obvious name, `parApply` is rarely used.

An Example: Random Subset Cross Validation (Spam Detection)

```
1  #Common setup
2  library(rpart)
3  library(parallel)
4
5  spam.data <- read.table("spam.data", header=FALSE, sep='
       ')
6  ...
7  fold <- function() {
8    train <- sample(c(0,1), prob=c(0.1, 0.9), replace=TRUE,
         size=nrow(spam.data))
9    trained.tree <- rpart(SPAM ~ ., data=spam.data[train ==
         1, ])
10   test <- predict(trained.tree, spam.data[train == 0, ])
11   predictions <- ifelse(test > 0.5, 1, 0)
12   true <- spam.data$SPAM[train == 0]
13   sum(true == predictions) / length(true)
14 }
```

An Example: Random Subset Cross Validation (Spam Detection)

Serial Train the model on a random 90% of the data, test on the other 10%. Measure the test error. Do this 10 times.

```
res <- vector(length=10)
system.time({
         for (i in 1:10) {
                  res[i] <- fold()
         }
})

#    user  system elapsed
# 12.450   0.040  12.514
```

An Example: Random Subset Cross Validation (Spam Detection)

Explicit Parallelism

```
1  system.time({
2    cl <- makePSOCKcluster(10)
3    clusterExport(cl, c("spam.data","fold"))
4    junk <-clusterEvalQ(cl, library(rpart))
5    clusterSetRNGStream(cl, 123)
6    res <- clusterEvalQ(cl, fold())
7    stopCluster(cl)
8  })
9
10 #    user  system elapsed
11 #    0.35    0.08    4.88
```

Which is faster, but not as fast as it should be... Why not?

A Snake in the Grass

What gives?

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). *Of course, the method also matters.*

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

Since

$$T(\texttt{CV}) \not< \not< T(\text{cluster setup})$$

so we do not achieve the expected boost in performance.

So, is parallelization **always** a good idea? **NO!**

An Example: Random Subset Cross Validation (Spam Detection)

Another important fact worth noting: forking n processes is faster than creating n socket connections.

Using `makeForkCluster`

```
#Entire process (FORK)
#   user  system elapsed
#  0.280   0.050   3.299
```

Using `makePSOCKcluster`

```
#Entire process (PSOCK)
#   user  system elapsed
#   0.35    0.08    4.88
```

Rule of Thumb: *Fork* on local host, *sockets* for distributed

parLapply

There is another (more useful) way to use the cluster by using `parLapply` which has the usual `apply` syntax but runs jobs across a cluster.

```
parLapply(cl, X, fun, ...)
```

1. `cl` is the cluster object.
2. `X` is some data or parameters to pass to the analysis function `fun`.

10-fold Cross Validation with parLapply

A new function for test/train on a particular fold. First randomly assign data to folds.

```
1  fold <- sample(seq(1, 10), size=nrow(spam.data), replace=
       TRUE)
2  fold.cv <- function(i) {
3           trainset <- spam.data[fold == i, ]
4           testset <- spam.data[fold != i, ]
5           trained.tree <- rpart(SPAM ~ ., data=trainset)
6           test <- predict(trained.tree, testset)
7           predictions <- ifelse(test > 0.5, 1, 0)
8           true <- testset$SPAM
9           sum(true == predictions) / length(true)
10 }
```

10-fold Cross Validation with parLapply

```
system.time({
  cl <- makeForkCluster(24)
  clusterSetRNGStream(cl, 123)
  res <- do.call(c, parLapply(cl, seq_len(mc), fold.cv))
  stopCluster(cl)
})
#    user  system elapsed
#   0.010   0.010   1.238
```

Note: since we fork the current process, no need to export variables to the cluster!

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.

`parallel`'s `multicore` Functionality

Disclaimer: No adaptation of `multicore` is distributed for Windows currently (ever?). Unlike `snow`, the `parallel` version of `multicore` remains almost entirely intact.

parallel's multicore Functionality

multicore provides functions for parallel execution of R code on systems with multiple cores or multiple CPUs.

Important! Unlike other parallelization packages, all subjobs started by multicore share the same state!

parallel's multicore Functionality

The main functions in multicore are

- mclapply, a parallel version of lapply.
- mcparallel, do something in a separate process.
- mccollect, 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 *a priori* 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: Random Subset CV with mclapply

Let's use mclapply to perform the same operation as in the previous example.

```
1  system.time({
2  acc <- do.call(c, mclapply(seq_len(10), fold, mc.
       cores = 10) )
3  })  #If I specify 24, I get \texttt{NA} for each
       core that was not scheduled.
4
5  #user  system elapsed
6  #16.280   0.390   1.699
7  #Remember, 12.5s serial!
```

seq_len is similar to a foreach from 1 to 10 or range.

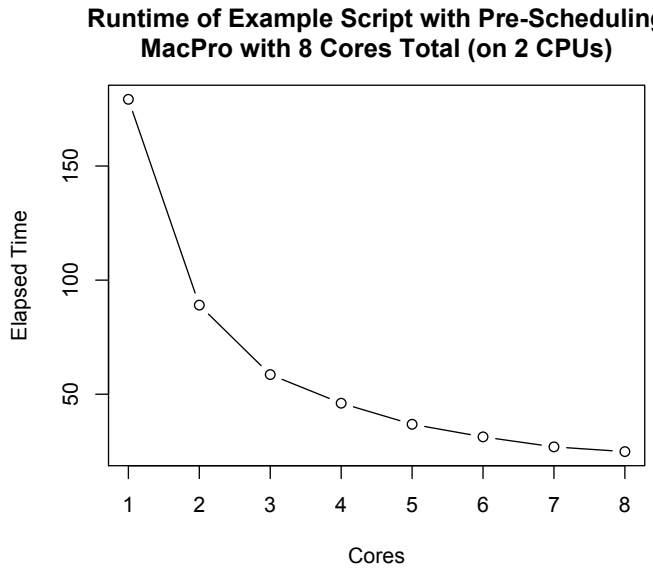
Example: 10-fold CV with `mclapply`

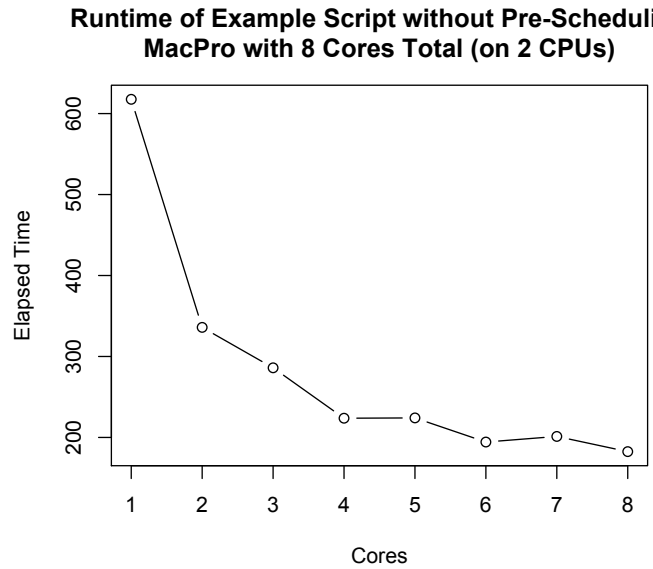
By modifying the original function, we can now do 10-fold CV easily.

```
system.time({
          acc <- do.call(c, mclapply(seq_len(10), fold.cv,
              mc.cores = 24) )
})
```

Example: `mclapply` Performance

For a text mining example (previous incarnation of this talk) on an 8-core system, processing time improves dramatically using all 8-cores.

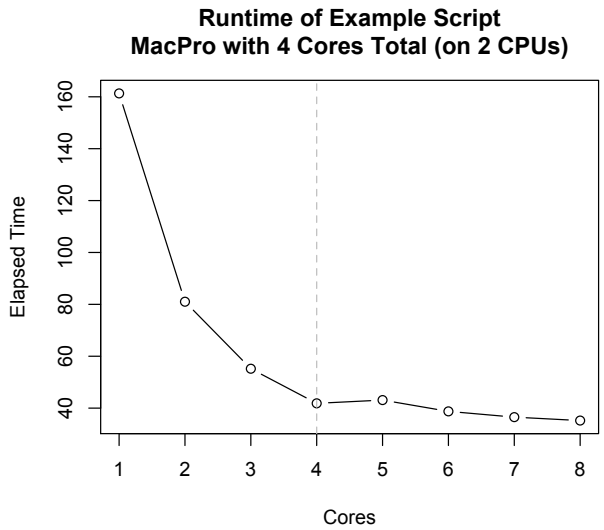


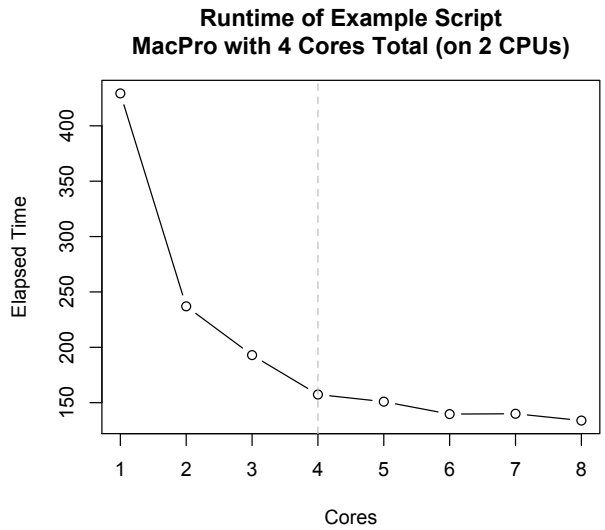


Notice that using pre-scheduling yields superior performance here, decreasing processing time from 3 minutes to 25 seconds.

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.

mcparallel and mccollect

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

```
mcparallel(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.

mcparallel and mccollect

mccollect allows us to retrieve the results of the spawned processes.

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

This function takes as parameters:

- jobs:
  - a list of process objects that were bound using mcparallel.
  - 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 and return.
- a function, intermediate, to execute while waiting for results.

Demonstration: `mcparallel` and `mccollect`

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 <- mcparallel(my.silly.loop(1, 10000000))
10 silly.2 <- mcparallel(my.silly.loop(2, 5000000))
11 mccollect(list(silly.1, silly.2))
```

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

Demonstration: `mcparallel` and `mccollect`

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

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

Demonstration: `mcparallel` and `mccollect`

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 <- mcparallel(my.silly.loop(1, 10000000))
6  silly.2 <- mcparallel(my.silly.loop(2, 5000000))
7  results <- mccollect(list(silly.1, silly.2), intermediate
       =status)
```

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

parallel multicore-Functionality 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...

Differences from `multicore` Standalone

There are two minor differences between `parallel` and the original `multicore` package

1. low-level functions in my previous slides are no longer exported in the namespace because they should never need to be used.
2. several functions (i.e. `parallel` and `collect`) now have `mc` added at the beginning of their names to prevent masking.

Switching Gears

The new `parallel` package distributed with R 2.14 provides

1. a modified version of `snow` allowing socket clusters and forking processes for explicit parallelism.
2. the `multicore` package for intra-host implicit parallelism.

Other packages originally from Revolution Computing offer other important parallelization capabilities.

1. `foreach` to iterate over a set of values in parallel.
2. `RHadoop` to interface R with Hadoop[1].

[1] `RHIPE`, `http://www.rhipe.org` is another package to integrate R and Hadoop, and is still actively developed

R and Hadoop: RHadoop

Hadoop is an open-source implementation of Map-Reduce, a concept that originated in functional languages such as Lisp and was made popular by Google[2].

In order to understand and use RHadoop, it is important to first understand MapReduce and Hadoop. Unfortunately, Hadoop can be quite complex, so we will skim the basics and see some examples.

[2] MapReduce: Simplified Data Processing on Large Clusters

MapReduce

MapReduce is a paradigm for processing huge amounts of data on disk (petabytes are a piece of cake at Yahoo). It consists of two steps:

1. **Map phase**: compute some transformation function on each piece of data independently (like a row from a log file, a web page, etc.) and output a key-value pair. The map phase is typically used for extracting fields from data, transforming or parsing data, and filtering data.
2. **Reduce phase**: the output from map phase is sorted and grouped by key. Some aggregate function is computed over the values associated with the key, and this is output to disk.

Hadoop MapReduce

Hadoop jobs are typically written in Java. For the full experience, and for its full power, it is generally suggested to write jobs in Java.

However, as long as your jobs can read from `STDIN` and write to `STDOUT`, you can use Hadoop Streaming (distributed with Hadoop) to write Hadoop jobs in any language, including R.

There are many limitations with Streaming however. `RHadoop` is based on Streaming.

A Boring but Useful Example

Suppose we have TBs of logs containing daily statistics about the number of times a particular ad was shown on a particular website. We want to provide an analysis of the number of times a particular ad was shown on a particular website *during the month of February.*

Input: log files for 28 days in February (the number of logs is likely > 28).

1. **Map phase:** for each line, output a *compound key* like `url:ad_id` and the the number of times the ad displayed on the page at URL. Note this is a key-value pair.
2. **Between Map and Reduce:** all values are placed into groups by key, and then distributed to reducers.
3. **Reduce phase:** Compute the sum over all of the values associated with the compound key.

How Does it Work?

Like the relationship status: It's Complicated.

There are several pieces to the framework.

- The **NameNode** is the queen bee – it maintains the index of the distributed filesystem HDFS. There is only one namenode per cluster, and it runs on the master node. It is also the single point of failure.
- The **SecondaryNameNode** maintains a snapshot of the namenode's memory structure *to reduce filesystem corruption and data loss.* It is **not** a failsafe for a failed namenode.

How Does it Work?

Like the relationship status: It's Complicated.

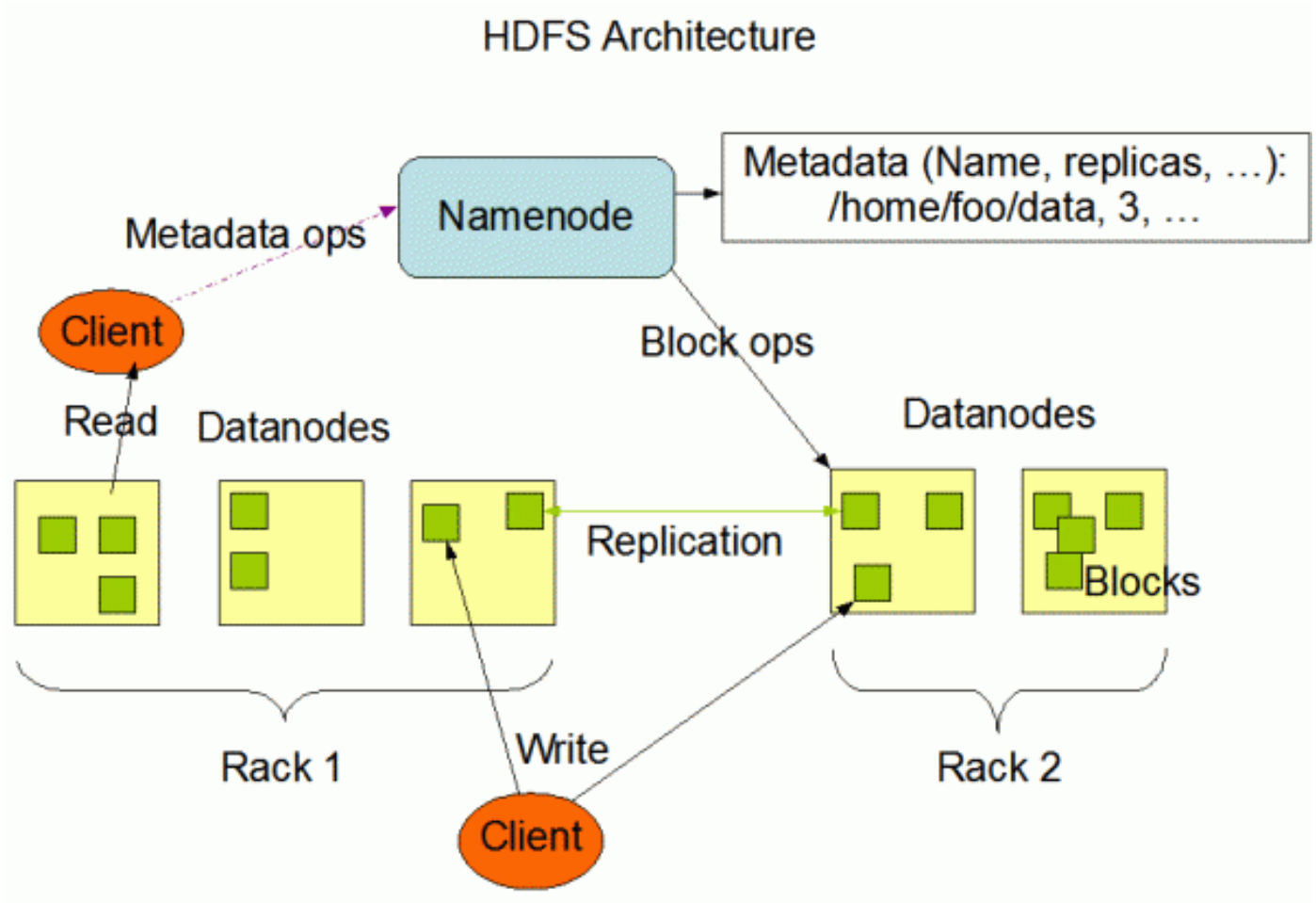
There are several pieces to the framework.

- The **JobTracker** maintains job scheduling information and also provides the main web portal for information about running, completed and failed jobs. Every job consists of several tasks.
- The **TaskTrackers** actually do the work and communicate with the JobTracker about tasks completed, in progress and failed.
- The **DataNode** is more low-level and performs the main I/O of moving bytes etc. for HDFS.

What Lives Where?

- The NameNode lives on the master node, always.
- The SecondaryNameNode should live on a separate node than the worker node, but not required.
- The JobTracker may be on the master node, or on a separate node.
- Every node hosts TaskTrackers.
- Every node is a DataNode.

What Lives Where?



HDFS: Hadoop Distributed FileSystem

HDFS is the filesystem that forms the backbone of Hadoop on a cluster.

- allows easy reading/writing of data from systems connected to the HDFS.
- stores data throughout the cluster, in chunks.
- provides data redundancy (if configured).
- NameNode stores metadata, DataNode stores data.
- a kinda-sorta "meta-filesystem". On most installations, sits atop another filesystem.

Do I Need a Hadoop Cluster?

There are several options for running Hadoop jobs that do not require $10,000+ of hardware.

1. Can run jobs locally with or without HDFS. (easiest option)
2. Can run jobs on a cluster or any size.
3. Can setup and configure Hadoop to run on EC2 using either one machine, or multiple with low or high-end hardware.
4. Can use Amazon Elastic MapReduce (EMR) to run jobs written in Java or Streaming with multiple options. (easier option)

Words of Wisdom

"When first beginning with Hadoop, you may feel overwhelmed. It takes a month or so of solid continuous use to get entirely comfortable with the framework, and to learn how all of the pieces move together and fit together. Don't get discouraged."

-Me

Various Flavors of Hadoop

Hadoop has undergone a commercialization over the past few years.

(a)

(b)

(c)

(d)

`Cloudera : Hadoop :: Revolution : R :: Canonical :: Ubuntu`

Where to Hadoop?

Hadoop can be downloaded from Apache's website.

`http://hadoop.apache.org/`

For another flavor or Hadoop, check the vendor's website.

RHadoop Contents

RHadoop consists of three packages:

1. `rhdfs` for file management and I/O in HDFS.
2. `rmr` for executing MapReduce jobs.
3. `rhbase` for interfacing R with HBase.[3]

[3] We will not get here.

RHadoop Contents

RHadoop's three packages can be downloaded as stable archives, or as a git repository from

https://github.com/RevolutionAnalytics/RHadoop

The Missing Slide

Due to time constraints, and preventing the audience from getting frustrated, I defer to the documentation on how to install and set up Hadoop and how to install RHadoop.

Installing just requires from Linux shell experience and patience.

The Missing Slide

Since Hadoop is typically used on commodity hardware, it is common to see Linux on these machines (since it is cheap... free) instead of Windows.

Most configuration guides are tailored towards Linux. My favorite set of directions are provided by Michael Noll[4][5]. Although they are for Ubuntu, they should work with most Linux distributions and Mac OS X with some labor.

[4] http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/

[5] http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/

How to Proceed

As I mentioned, installation and configuration is outside the scope of this talk. However, you will need to research (or already know) how to do the following before continuing:

1. Install **Sun** (Oracle) Java (not OpenJDK).
2. Install and configure Hadoop on your local machine.
3. Optionally: Install and configure Hadoop for multiple machines in a cluster.
4. Set environment variables for Hadoop and the JVM.
5. Allow the `hadoop` user (or whomever) to login without a password.
6. Install an R package from source.
7. Format the namenode to initialize HDFS.
8. Start and stop the Hadoop cluster.

Getting Started with RHadoop

First, we load the `rhdfs` package, and then the `rmr` package.
Note all of the prerequisites.

```
> library(rhdfs)
Loading required package: rJava
This is rhdfs 1.0.1. For overview type ?rhdfs.
HADOOP_HOME=/home/ryan/usr/local/hadoop #Must be set beforehand
HADOOP_CONF=/home/ryan/usr/local/hadoop/conf
> library(rmr)
Loading required package: RJSONIO
Loading required package: itertools
Loading required package: iterators
Loading required package: digest
```

Hello World

As with all of these parallel libraries, we use lists and apply functions to them. This is natural in MapReduce. Suppose we want to compute the squares of a bunch of numbers, say 10,000.

```
1  my.ints <- 1:10000
2  out = lapply(my.ints, function(x) x^2)    #almost instant
3  out
4  [[1]]
5  [1] 1
6
7  [[2]]
8  [1] 4
9
10 [[3]]
11 [1] 9
12 ...
```

Hello World

Now let's use Hadoop. I must warn that this is just a small example, so don't be disappointed by the result.

```
1  my.ints = to.dfs(1:10000)
2  out = mapreduce(input = my.ints, map = function(k,v
       ) keyval(v, v^2), reduce = function(k,v) keyval
       (k, v))
3  stuff = from.dfs(out)
```

Watch what happens...

Hello World

We've created a monster! Hadoop is very verbose. The main line of interest is **Job complete.**

```
> my.ints = to.dfs(1:10000)
12/03/14 22:42:33 INFO util.NativeCodeLoader: Loaded the native-hadoop library
12/03/14 22:42:33 INFO zlib.ZlibFactory: Successfully loaded & initialized native-zlib library
12/03/14 22:42:33 INFO compress.CodecPool: Got brand-new compressor
> out = mapreduce(input = my.ints, map = function(k,v) keyval(v, v^2))
packageJobJar: [/tmp/RtmpxwZWJx/rmr-local-env, /tmp/RtmpxwZWJx/rmr-global-env, /tmp/RtmpxwZWJx/rhstr.map5
12/03/14 22:42:36 INFO mapred.FileInputFormat: Total input paths to process : 1
12/03/14 22:42:36 INFO streaming.StreamJob: getLocalDirs(): [/home/ryan/tmp/mapred/local]
12/03/14 22:42:36 INFO streaming.StreamJob: Running job: job_201203142235_0003
12/03/14 22:42:36 INFO streaming.StreamJob: To kill this job, run:
12/03/14 22:42:36 INFO streaming.StreamJob: /home/ryan/usr/local/hadoop/bin/hadoop job  -Dmapred.job.trac
12/03/14 22:42:36 INFO streaming.StreamJob: Tracking URL: http://localhost:50030/jobdetails.jsp?jobid=job_
12/03/14 22:42:37 INFO streaming.StreamJob:  map 0%  reduce 0%
12/03/14 22:42:45 INFO streaming.StreamJob:  map 49%  reduce 0%
12/03/14 22:42:47 INFO streaming.StreamJob:  map 100%  reduce 0%
12/03/14 22:42:54 INFO streaming.StreamJob:  map 100%  reduce 17%
12/03/14 22:42:55 INFO streaming.StreamJob:  map 100%  reduce 100%
12/03/14 22:42:56 INFO streaming.StreamJob: Job complete: job_201203142235_0003
12/03/14 22:42:56 INFO streaming.StreamJob: Output: /tmp/RtmpxwZWJx/file590e24aa7be1
```

This example has a lot of overhead, and takes about 30 seconds to complete.

Dude... What Just Happened?

1. We create a vector from 1 to 10000 and store it in HDFS.[6] We get back a *handle* to the file in HDFS.
2. We call the `mapreduce` function to run the Hadoop job. We specify the handle from the previous step as the input. We also specify a `map` function and a reduce function. We get a handle to the output in HDFS.
3. We convert the data in HDFS into an R object.

[6] a temporary file is created in HDFS, but let's abstract it away.

Dude... What Just Happened?

Abbreviated output from the Hadoop execution:

```
1  my.ints = to.dfs(1:10000)
2  out = mapreduce(input = my.ints, map = function(k,v)
       keyval(v, v^2), reduce = function(k,v) keyval(k, v))
3  stuff = from.dfs(out)
4  > stuff[1]
5  [[5]]
6  [[5]]$key
7  [1] 5
8
9  [[5]]$val
10 [1] 25
11 ...
```

In typical MapReduce fashion, we get back a list of keys and values.

The `map` Parameter

The `map` parameter specifies the map function.

- It accepts two parameters: `key` and `value`.[7]
- The value `v` contains a piece of data that we are going to process. For this example, it is an element of the list `1:10000`.
- We make a call to the `keyval` function which tells Hadoop to return `k` and `v` as a key-value pair from the map phase.[8].

Note: the map function can also return a list of key/value calls, or `NULL`.

---

[7] The input key to the mapper is usually thrown away in the map phase. Depends on the `InputFormat`.

[8] like `collect` or `write` in the Java API.

Our `map` Function

```
map = function(k,v) keyval(v, v**2)
```

- read in a key `k` (thrown out), and a value `v`.
- return a key/value pair where the key is the original value `v` and the value is the exponentiation `v**2`.

The `reduce` Parameter

The `reduce` parameter specifies the reduce function.

- It accepts two parameters: `key` and `value` which come from the output of the map function.
- The key `k` specifies a group of data that contains one or more values.
- The value `v` contains one specific value with the key specified by `k`, a squared element.
- We make a call to the `keyval` function which tells Hadoop to return `k` and `v` as a key-value pair from the map phase.[9].

[9] like `collect` or `write` in the Java API.

Our `reduce` Function

```
reduce = function(k,v) keyval(k, v)
```

- read in a key `k` (a group identifier), and a value `v`.
- return a key/value pair where the key is the same, and the value is also the same (we do not do any processing in the reducer for this example).

This was an example of a map-only job, or a job with an *identity reducer*. Many data transformation and text parsing jobs are map-only.

Use MapReduce for k-Means

Most of the work can be done locally. To compute the new centers at each iteration, we use MapReduce.

```
1  kmeans =
2    function(points, ncenters, iterations = 10, distfun =
         NULL) {
3      if(is.null(distfun))
4        distfun =
5          function(a,b) norm(as.matrix(a-b), type = 'F')
6      newCenters =
7        kmeans.iter(
8          points,
9          distfun,
10         ncenters = ncenters)
11     for(i in 1:iterations) {
12       newCenters = kmeans.iter(points, distfun, centers =
             newCenters)}
13     newCenters}
```

Note that MapReduce will be invoked by `kmeans.iter`.

Better Example: *k*-Means

```
1  kmeans.iter =
2   function(points, distfun, ncenters = dim(centers)[1],
        centers = NULL) {
3    from.dfs(mapreduce(input = points,
4     map =
5      if (is.null(centers)) {
6       function(k,v) keyval(sample(1:ncenters,1),v)}
7      else {
8       function(k,v) {
9        distances = apply(centers, 1, function(c) distfun(
             c,v))
10       keyval(centers[which.min(distances),], v)}},
11     reduce = function(k,vv) keyval(NULL, apply(do.call(
           rbind, vv), 2, mean))),
12   to.data.frame = T)}
```

The map Function

```
1  map =
2    if (is.null(centers)) {
3      function(k,v) keyval(sample(1:ncenters,1),v)}
4    else {
5      function(k,v) {
6        distances = apply(centers, 1, function(c) distfun(
             c,v))
7        keyval(centers[which.min(distances),], v)}
8    }
```

- If no centers have been computed, randomly pick a cluster for each value `v`. Return the cluster ID as `key`.
- Otherwise, apply the distance function to the list of centers and the values `v`. Pick the center with the minimal distance. Return the new center as the key `k` and the data value `v`.

The reduce Function

```
reduce = function(k,vv) keyval(NULL,
      apply(do.call(rbind, vv), 2, mean)))
```

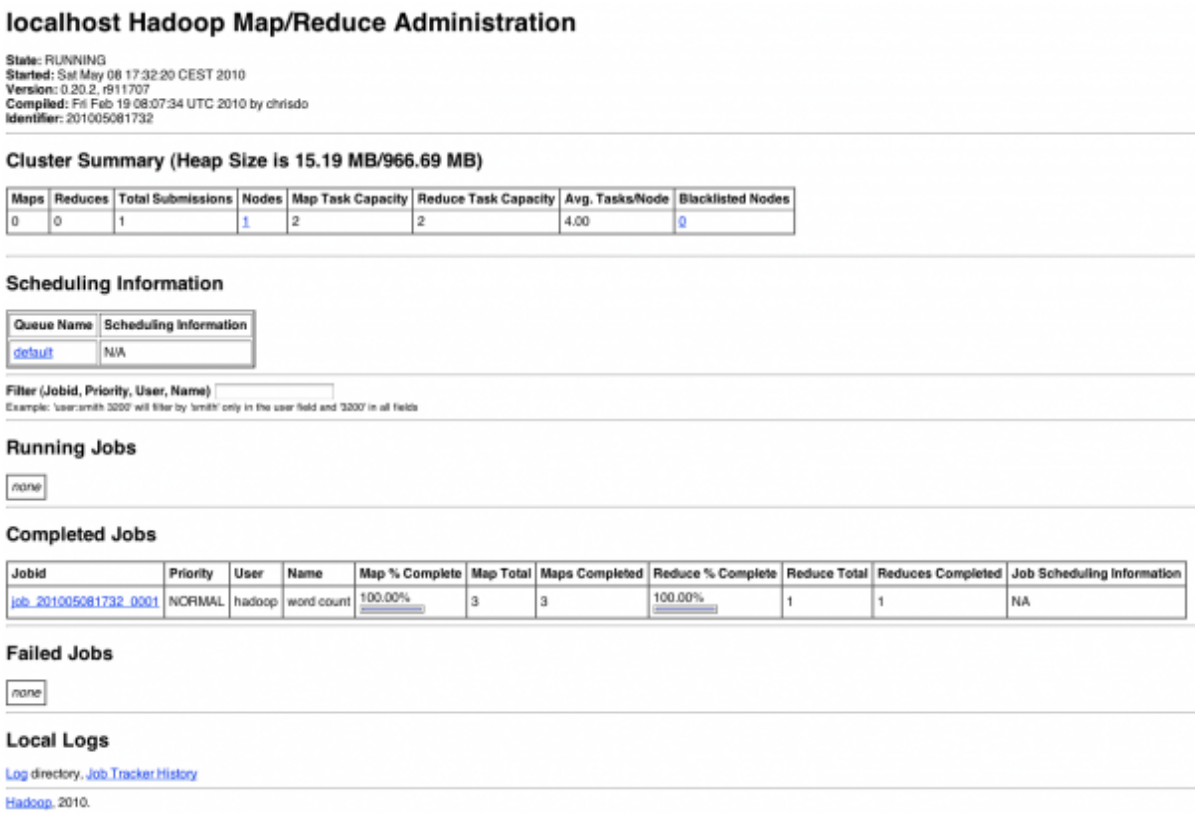
- The input is a key (group), in this case, a center ID. The other input is the list of values vv for that key k.
- We row bind all of the data points for this cluster, take the mean, and return it as a value, but toss the cluster ID (key) because it is meaningless.

Diagnostic Information

The JobTracker has a web interface, typically available at `http://master_ip:50030`, that provides information about running, completed and failed jobs. This is a good place to look for errors if a job fails.

Understanding what the Java exceptions mean takes a lot of practice. The URL to the diagnostic page for your job is reported to you when the job starts running.

Diagnostic Information



localhost Hadoop Map/Reduce Administration

State: RUNNING
Started: Sat May 08 17:32:20 CEST 2010
Version: 0.20.2, r911707
Compiled: Fri Feb 19 08:07:34 UTC 2010 by chrisdo
Identifier: 201005081732

Cluster Summary (Heap Size is 15.19 MB/966.69 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
0	0	1	1	2	2	4.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

none

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_201005081732_0001	NORMAL	hadoop	word count	100.00%	3	3	100.00%	1	1	NA

Failed Jobs

none

Local Logs

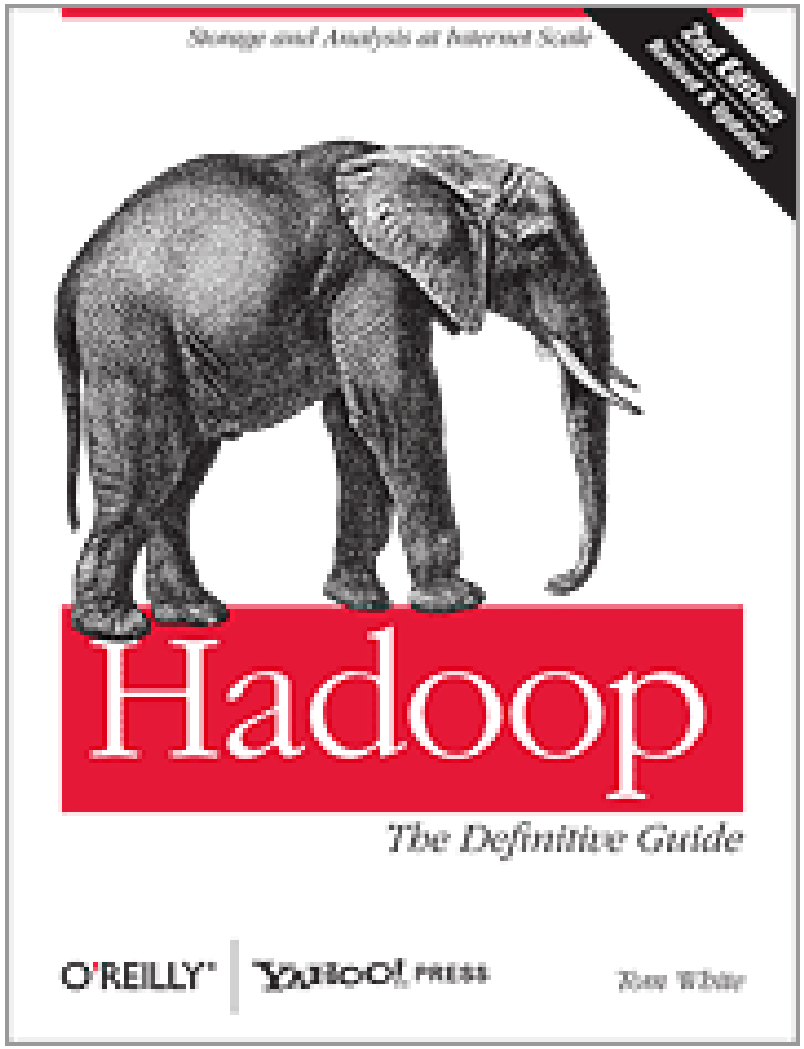
Log directory, Job Tracker History

Hadoop, 2010.

Other Hadoop Friends

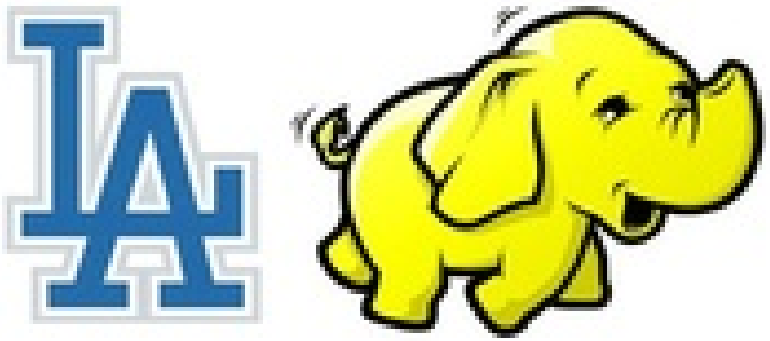
1. **HBase** is a column-oriented data store.
2. **Hive** is a data warehouse for ad hoc querying.
3. **Pig** for ad-hoc analysis similar to R.
4. **Mahout** for scalable machine learning (partially atop Hadoop).
5. **Cascading, Azkaban, Oozie** for chaining jobs and workflow.

A Good Hadoop Reference



A Good Hadoop Reference

Los Angeles Hadoop Users Group (LA-HUG)
`http://www.meetup.com/LA-HUG/`



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 3.

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 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!

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).

What I Hope to See

I look forward to the future. I hope to see:

1. more, and easier to use GPU packages for R.
2. an interface to Mahout.
3. interfaces to GraphLab and Spark.
4. Integration with other Hadoop subprojects (Pig etc.)

In Conclusion

1. R provides several ways to parallelize tasks.
2. They are pretty easy to use.
3. It is important to know when *not* to parallelize.
4. They do not fall victim to the Swiss Cheese Phenomenon under typical usage.
5. Worth learning as more and more packages begin to suggest them (`tm` and some other NLP packages).
6. Hadoop is the go-to standard for big data a la cheap.

My Previous Slides

For (much) more (detailed) information about the standalone `snow` family of packages, `foreach` and `multicore`, see the materials from my previous talks:

`Parallelism:`
`http://www.bytemining.com/`
`2010/07/taking-r-to-the-limit-part-i-parallelization-in-r/`

`Big Data:`
`http://www.bytemining.com/`
`2010/08/taking-r-to-the-limit-part-ii-large-datasets-in-r/`

For More Information

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!

Appendix A

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?

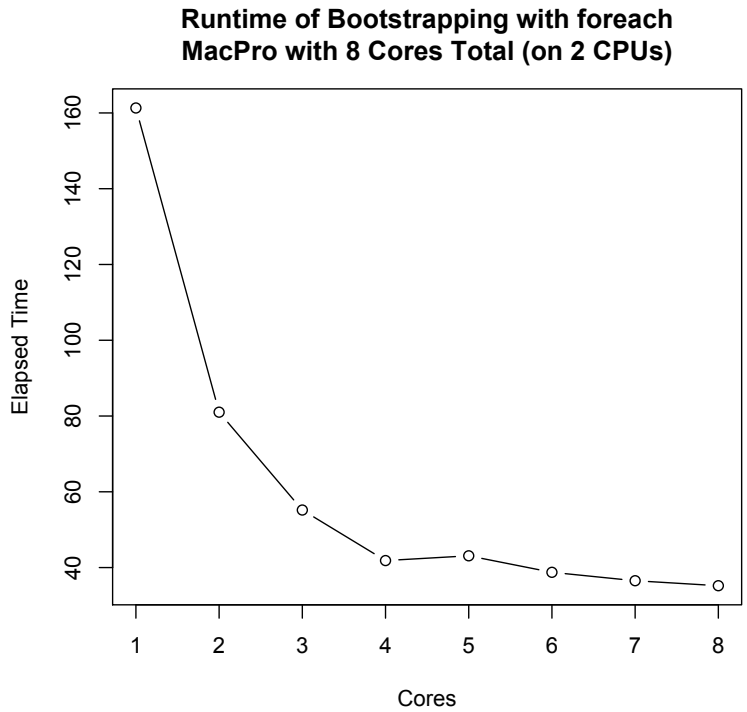
```
1  data(iris)
2
3  iris.sub <- iris[which(iris[, 5] != "setosa"), c(1,5)]
4  trials <- 10000
5  result <- foreach(icount(trials), .combine=cbind) %dopar%
       {
6      indices <- sample(100, 100, replace=TRUE)
7      glm.result <- glm(iris.sub[indices, 2]~iris.sub[
           indices, 1], family=binomial("logit"))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