# *Parallel Options for R*

**Glenn K. Lockwood**
**SDSC User Services**

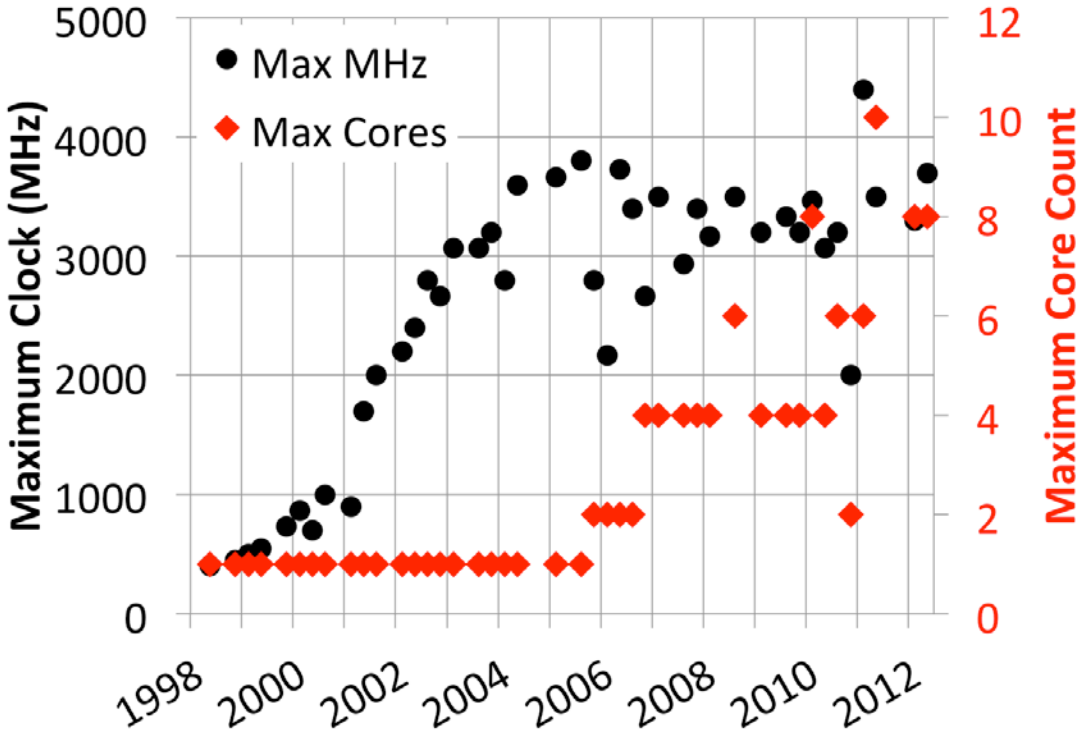
**glock@sdsc.edu**

# *Motivation*

**"I just ran an intensive R script [on the supercomputer]. It's not much faster than my own machine."**

# Motivation

**"I just ran an intensive R script [on the supercomputer]. It's not much faster than my own machine."**



# *Outline/Parallel R Taxonomy*

- **lapply-based parallelism**
  - multicore library
  - snow library
- **foreach-based parallelism**
  - doMC backend
  - doSNOW backend
  - doMPI backend
- **Map/Reduce- (Hadoop-) based parallelism**
  - Hadoop streaming with R mappers/reducers
  - Rhadoop (rmr, rhdfs, rhbase)
  - RHIPE

# *Outline/Parallel R Taxonomy*

- **Poor-man's Parallelism**
  - lots of Rs running
  - lots of input files
- **Hands-off Parallelism**
  - OpenMP support compiled into R build
  - Dangerous!

## Parallel Options for R

# *RUNNING R ON GORDON*

# *R with the Batch System*

- **Interactive jobs**

```
qsub -I
     -l nodes=1:ppn=16:native,walltime=01:00:00
     -q normal
```

- **Non-interactive jobs**

```
qsub myrjob.qsub
```

- **Run it on the login nodes instead of using qsub**

**DO NOT DO THIS**

# Serial / Single-node Script

```
#!/bin/bash
#PBS -N Rjob
#PBS -l nodes=1:ppn=16:native
#PBS -l walltime=00:15:00
#PBS -q normal

### Special R/3.0.1 with MPI/Hadoop libraries
source /etc/profile.d/modules.sh
export MODULEPATH=/home/glock/gordon/modulefiles:$MODULEPATH
module swap mvapich2_ib openmpi_ib
module load R/3.0.1
export OMP_NUM_THREADS=1

cd $PBS_O_WORKDIR
R CMD BATCH ./myrscript.R
```

# MPI / Multi-node Script

```
#!/bin/bash
#PBS -N Rjob
#PBS -l nodes=2:ppn=16:native
#PBS -l walltime=00:15:00
#PBS -q normal

### Special R/3.0.1 with MPI/Hadoop libraries
source /etc/profile.d/modules.sh
export MODULEPATH=/home/glock/gordon/modulefiles:$MODULEPATH
module swap mvapich2_ib openmpi_ib
module load R/3.0.1
export OMP_NUM_THREADS=1

cd $PBS_O_WORKDIR
mpirun -n 1 R CMD BATCH ./myrscript.R
```

# *Follow Along Yourself*

- **Download sample scripts**
  - Copy /home/diag/SI2013-R/parallel_r.tar.gz
  - See Piazza site for link to all on-line material
- **Serial and multicore samples can run on your laptop**
- **snow (and multicore) will run on Gordon with two files:**
  - gordon-mc.qsub - for single-node (serial or multicore)
  - gordon-snow.qsub – for multi-node (snow)
  - Just change the ./kmeans-*.R file in the last line of the script

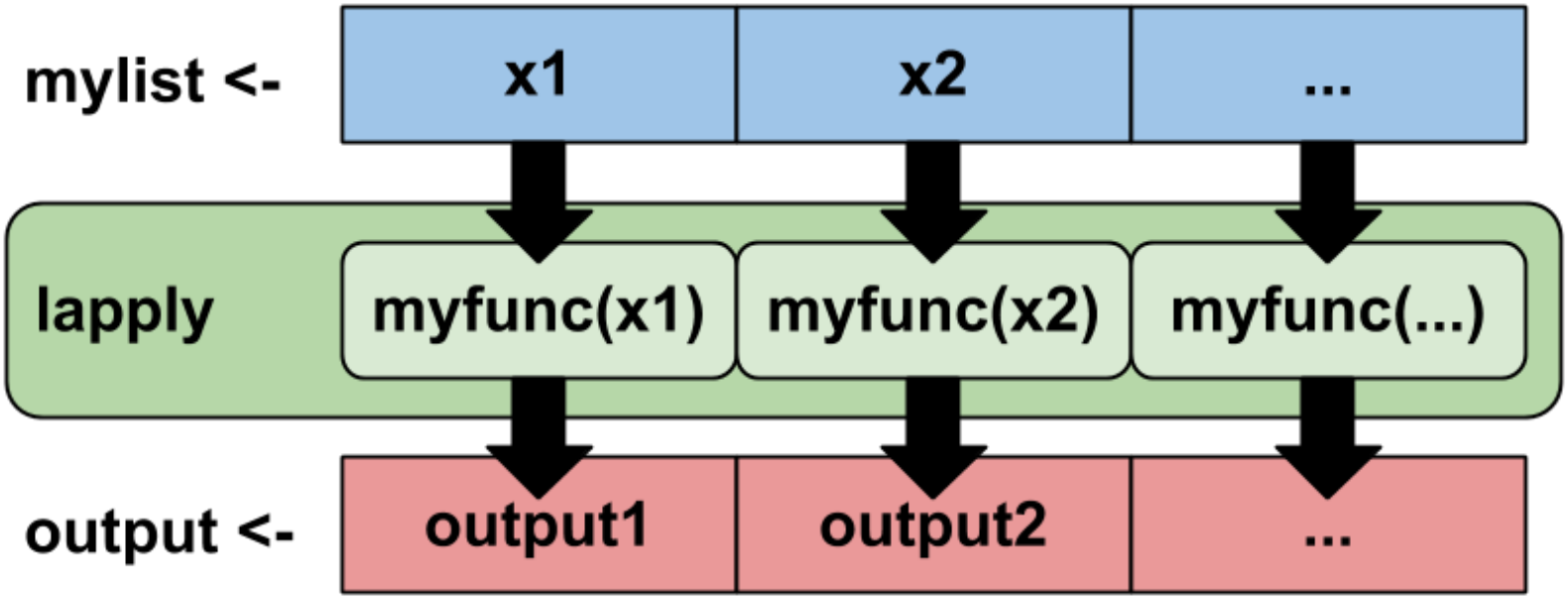
## Parallel Options for R – Conventional Parallelism

# *K-MEANS EXAMPLES*

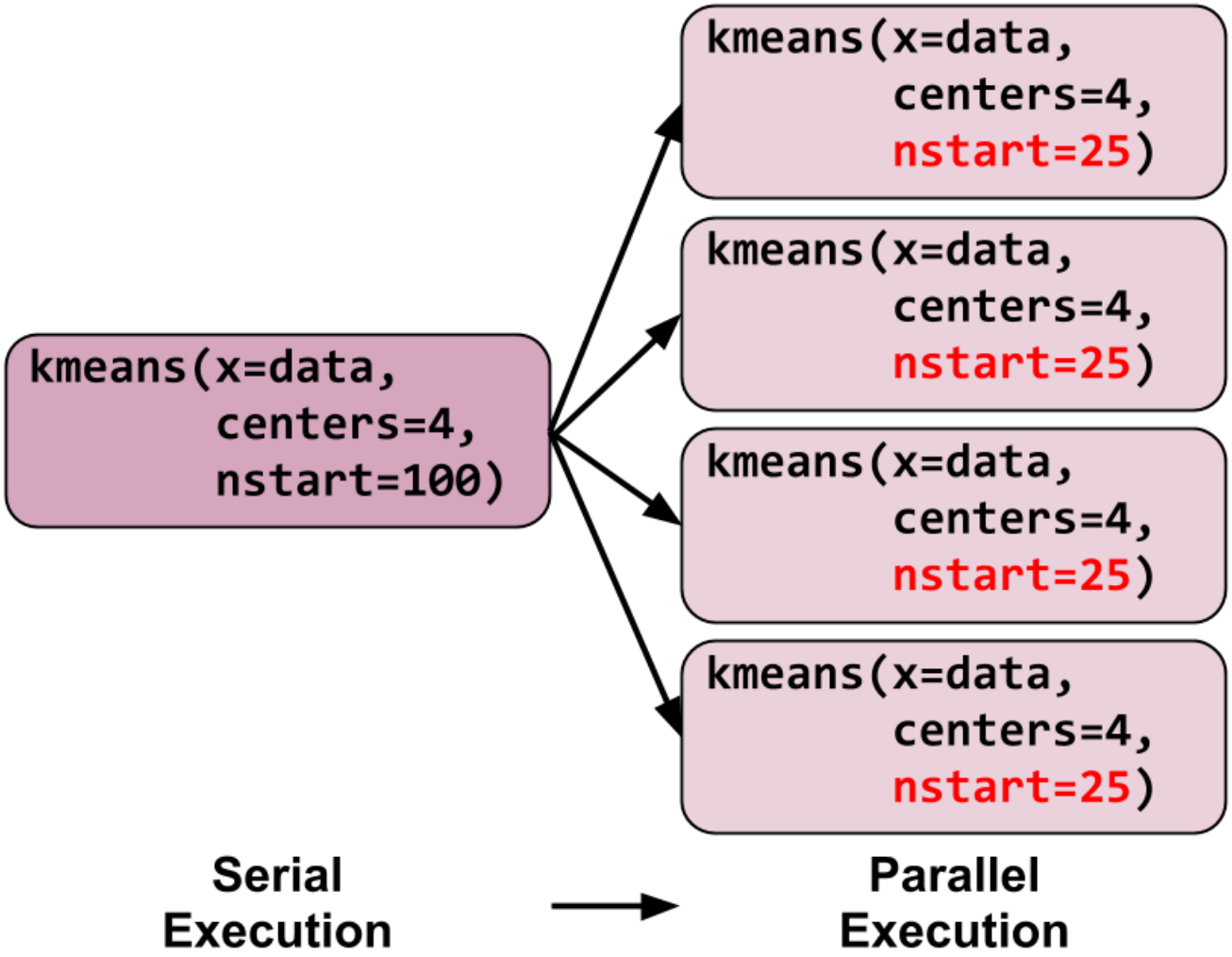
# *lapply-based Parallelism*

- **lapply: apply a function to every element of a list, e.g.,**

output <- lapply(X=mylist, FUN=myfunc)



# k-means: The lapply Version



# Example: k-means clustering

- Iteratively approach solutions from random starting position
- More starts = better chance of getting "most correct" solution
- Simplest (serial) example:

```
data <- read.csv('dataset.csv')
result <- kmeans(x=data,
                 centers=4,
                 nstart=100)
print(result)
```

# k-means: The lapply Version

```
data <- read.csv('dataset.csv')

parallel.function <- function(i) {
    kmeans( x=data, centers=4, nstart=i )
}

results <- lapply( c(25, 25, 25, 25),
                   FUN=parallel.function )
temp.vector <- sapply( results,
    function(result) { result$tot.withinss }
)
result <- results[[which.min(temp.vector)]]
print(result)
```

# *k-means: The lapply Version*

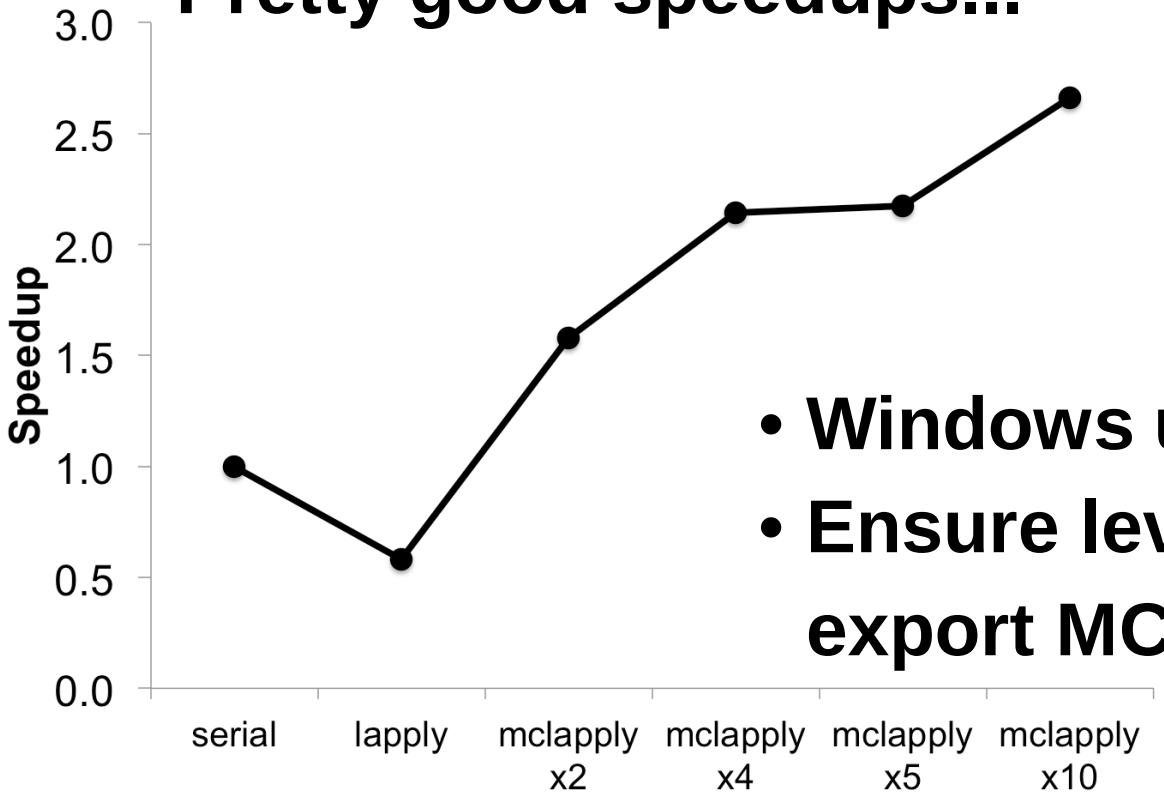
- ***Identical*** **results to simple version**
- ***Significantly*** **more complicated (>2x more lines of code)**
- **55%** ***slower*****(!)**
- **What was the point?**

# k-means: The mclapply Version

```
library(parallel)
data <- read.csv('dataset.csv')

parallel.function <- function(i) {
    kmeans( x=data, centers=4, nstart=i )
}

results <- mclapply( c(25, 25, 25, 25),
                     FUN=parallel.function )
temp.vector <- sapply( results,
    function(result) { result$tot.withinss } )
result <- results[[which.min(temp.vector)]]
print(result)
```

# *k-means: The mclapply Version*

- ***Identical*** **results to simple version**
- **Pretty good speedups...**




- **Windows users out of luck**
- **Ensure level of parallelism: export MC_CORES=4**

# k-means: The ClusterApply Version

```
library(parallel)
data <- read.csv('dataset.csv')

parallel.function <- function(i) {
  kmeans( x=data, centers=4, nstart=i )
}
cl <- makeCluster( mpi.universe.size(), type="MPI" )
clusterExport(cl, c('data'))
results <- ClusterApply( c(25, 25, 25, 25),
    FUN=parallel.function )
temp.vector <- sapply( results,
    function(result) { result$tot.withinss } )
result <- results[[which.min(temp.vector)]]
print(result)
stopCluster(cl)
mpi.exit()
```

# *k-means: The ClusterApply Version*

- **Scalable beyond a single node's cores**
- **...but memory of a single node still is bottleneck**
- **makeCluster( ..., type="XYZ") where XYZ is**
  - FORK – essentially mclapply with snow API
  - PSOCK – uses TCP; useful at lab scale
  - MPI – native support for Infiniband**

** requires snow and Rmpi libraries. Installation not for faint of heart; tips on how to do this are on my website

# *foreach-based Parallelism*

- **foreach: evaluate a for loop and return a list with each iteration's output value**

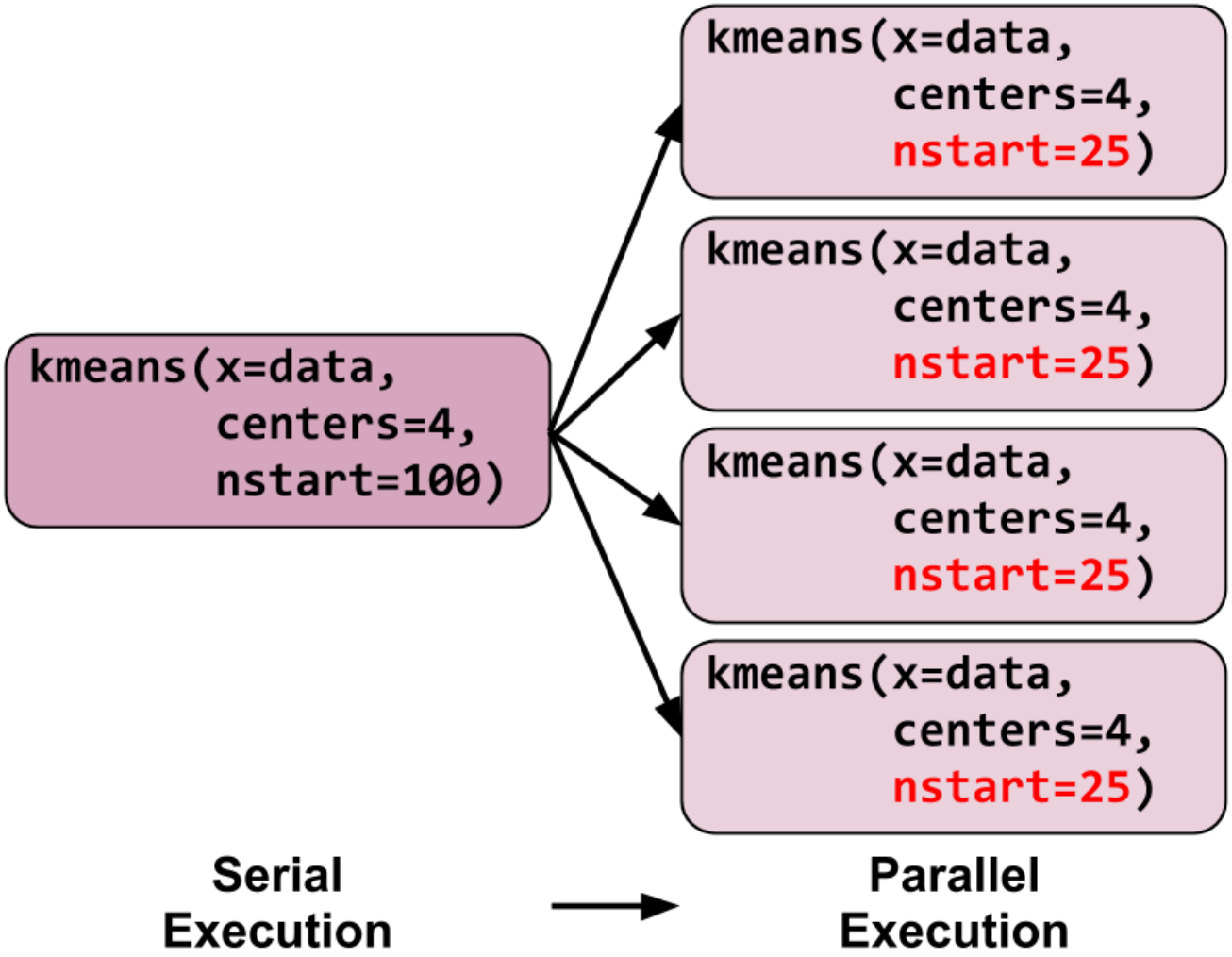
  output <- foreach(i = mylist) %do% { *mycode* }

- **similar to lapply BUT**
  - do not have to evaluate a function on each input object
  - relationship between mylist and mycode is not prescribed
  - same API for different parallel backends
  - **assumption is that mycode's side effects are *not important***

# Example: k-means clustering

```
data <- read.csv('dataset.csv')
result <- kmeans(x=data,
                 centers=4,
                 nstart=100)
print(result)
```

# k-means: The foreach Version



# k-means: The foreach Version

```
library(foreach)
data <- read.csv('dataset.csv')

results <- foreach( i = c(25,25,25,25) ) %do% {
    kmeans( x=data, centers=4, nstart=i )
}

temp.vector <- sapply( results, function(result)
    { result$tot.withinss } )
result <- results[[which.min(temp.vector)]]

print(result)
```

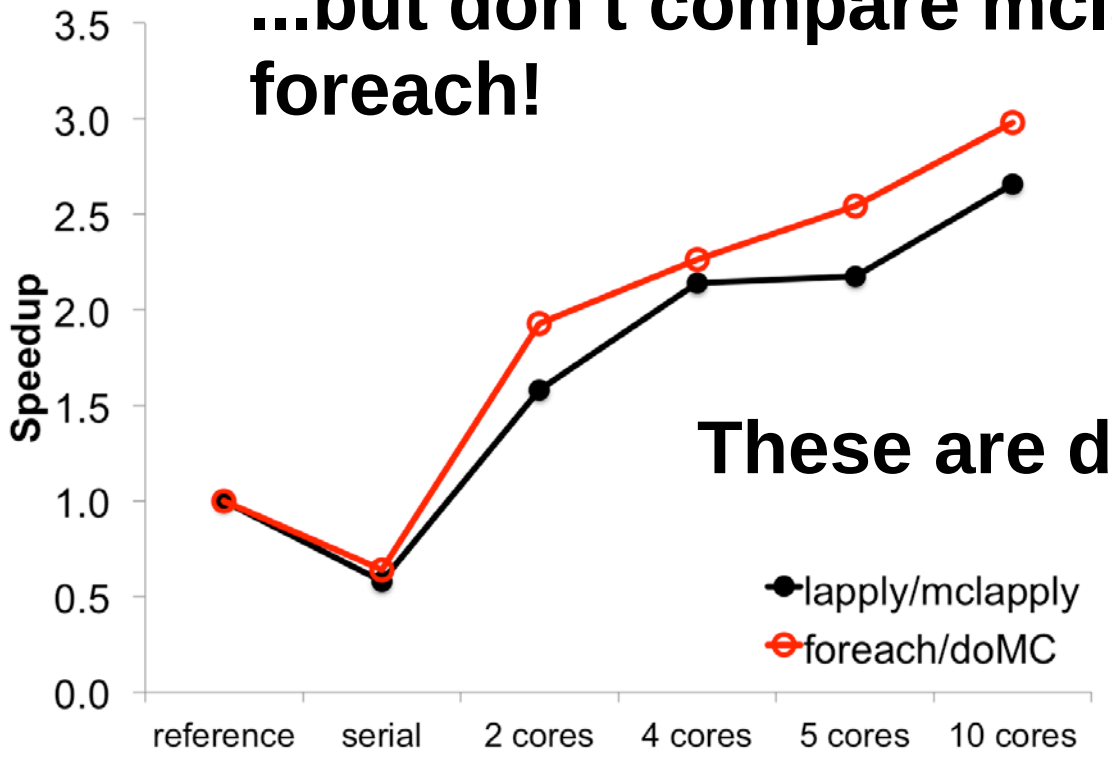
# k-means: The foreach/doMC Version

```
library(foreach)
library(doMC)

data <- read.csv('dataset.csv')
registerDoMC(4)
results <- foreach( i = c(25,25,25,25) ) %dopar% {
    kmeans( x=data, centers=4, nstart=i )
}

temp.vector <- sapply( results, function(result)
    { result$tot.withinss } )
result <- results[[which.min(temp.vector)]]

print(result)
```

# *foreach/doMC: Sanity Check*

- ***Identical*** **results to simple version**
- **Pretty good speedups...**

**...but don't compare mclapply to foreach!**




**These are dumb examples!**

# k-means: The doSNOW Version

```
library(foreach)
library(doSNOW)
data <- read.csv('dataset.csv')
cl <- makeCluster( mpi.universe.size(), type="MPI" )
clusterExport(cl,c('data'))
registerDoSNOW(cl)
results <- foreach( i = c(25,25,25,25) ) %dopar% {
    kmeans( x=data, centers=4, nstart=i )
}

temp.vector <- sapply( results, function(result)
    { result$tot.withinss } )
result <- results[[which.min(temp.vector)]]

print(result)
stopCluster(cl)
mpi.exit()
```

# *lapply/foreach Summary*

- **designed for trivially parallel problems**
- **similar code for multi-core and multi-node**

- **many bottlenecks remain**
  - file I/O is serial
  - limited by the RAM on the master node (64GB on SDSC Gordon and Trestles)

- **suitable for compute-intensive or big-ish data problems**

## Parallel Options for R – Map/Reduce-based Methods

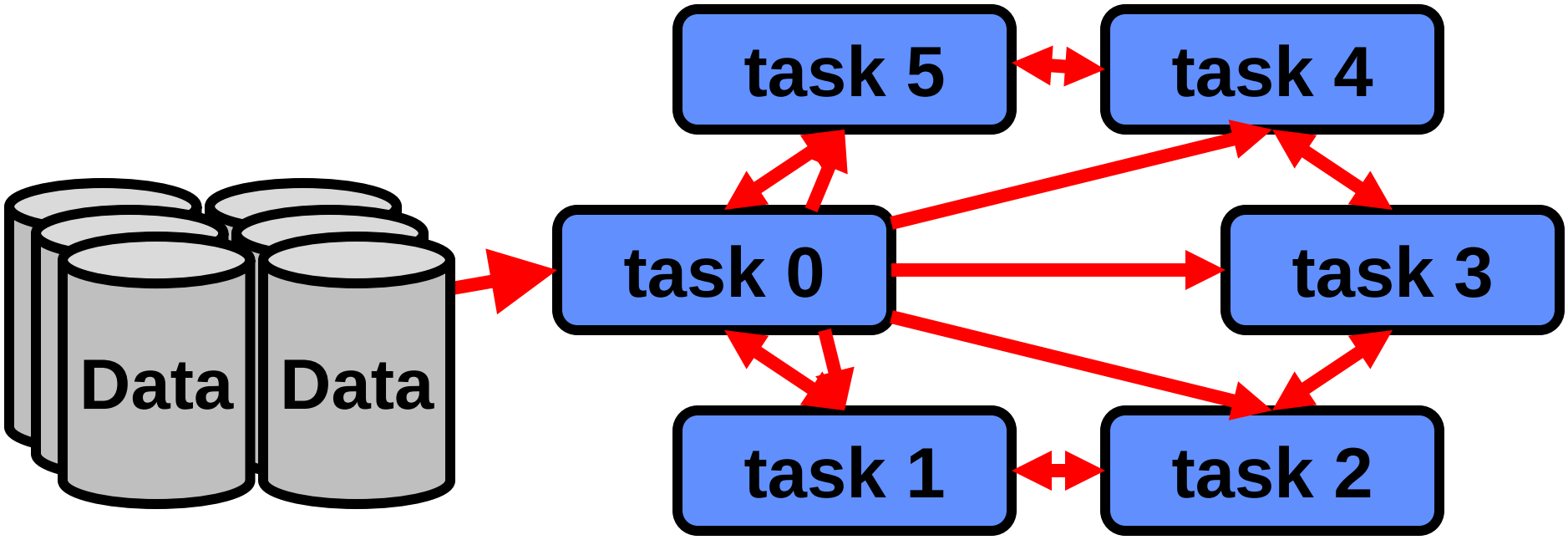
# *30,000-FT OVERVIEW*

# *Map/Reduce vs. Traditional Parallelism*

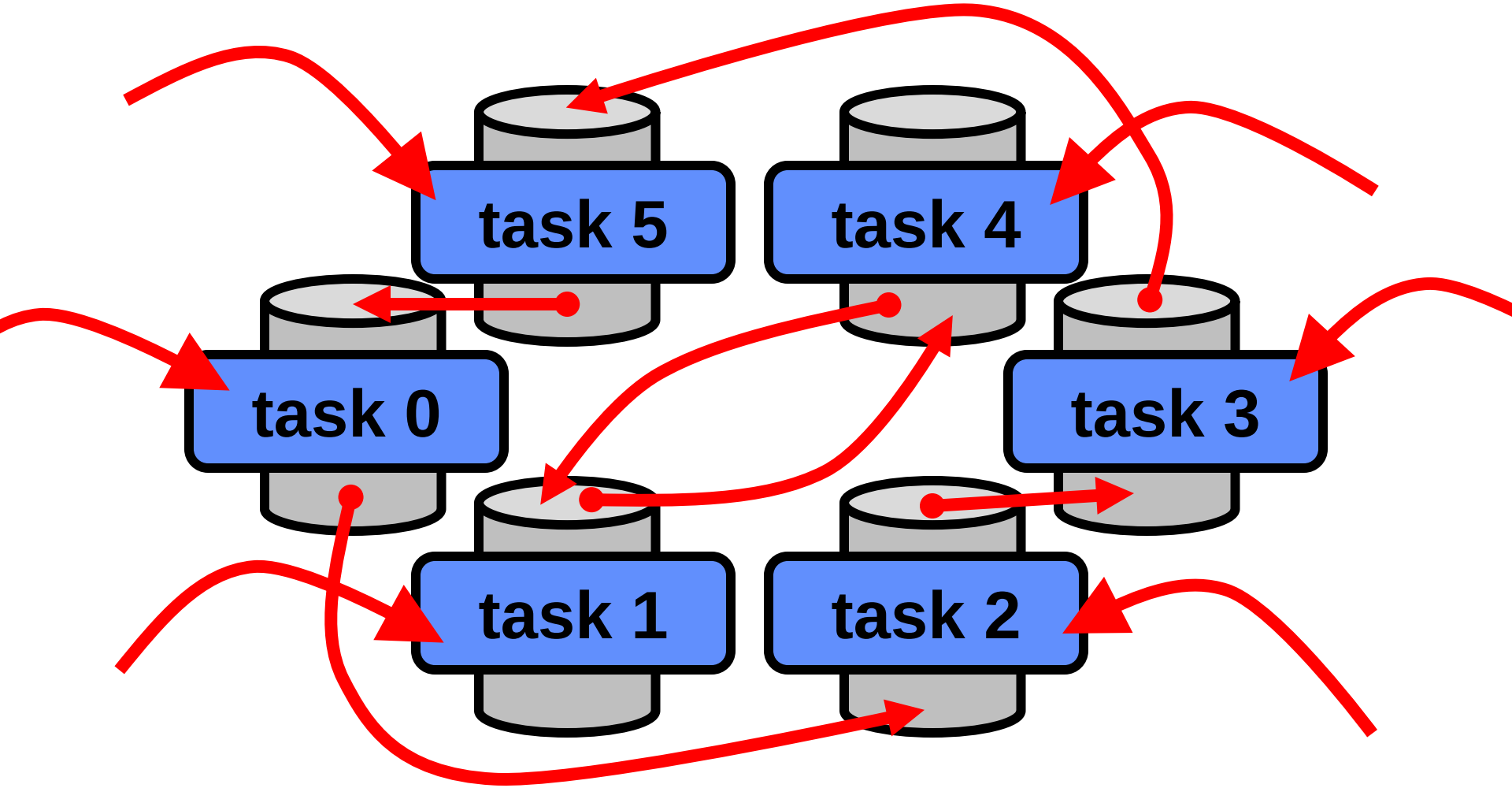
- **Traditional problems**
  - Problem is CPU-bound
  - Input data is gigabyte-scale
  - Speed up with MPI (multi-node), OpenMP (multi-core)

- **Data-intensive problems**
  - Problem is I/O-bound
  - Input data is tera-, peta-, or exa-byte scale
  - Speed up with ???

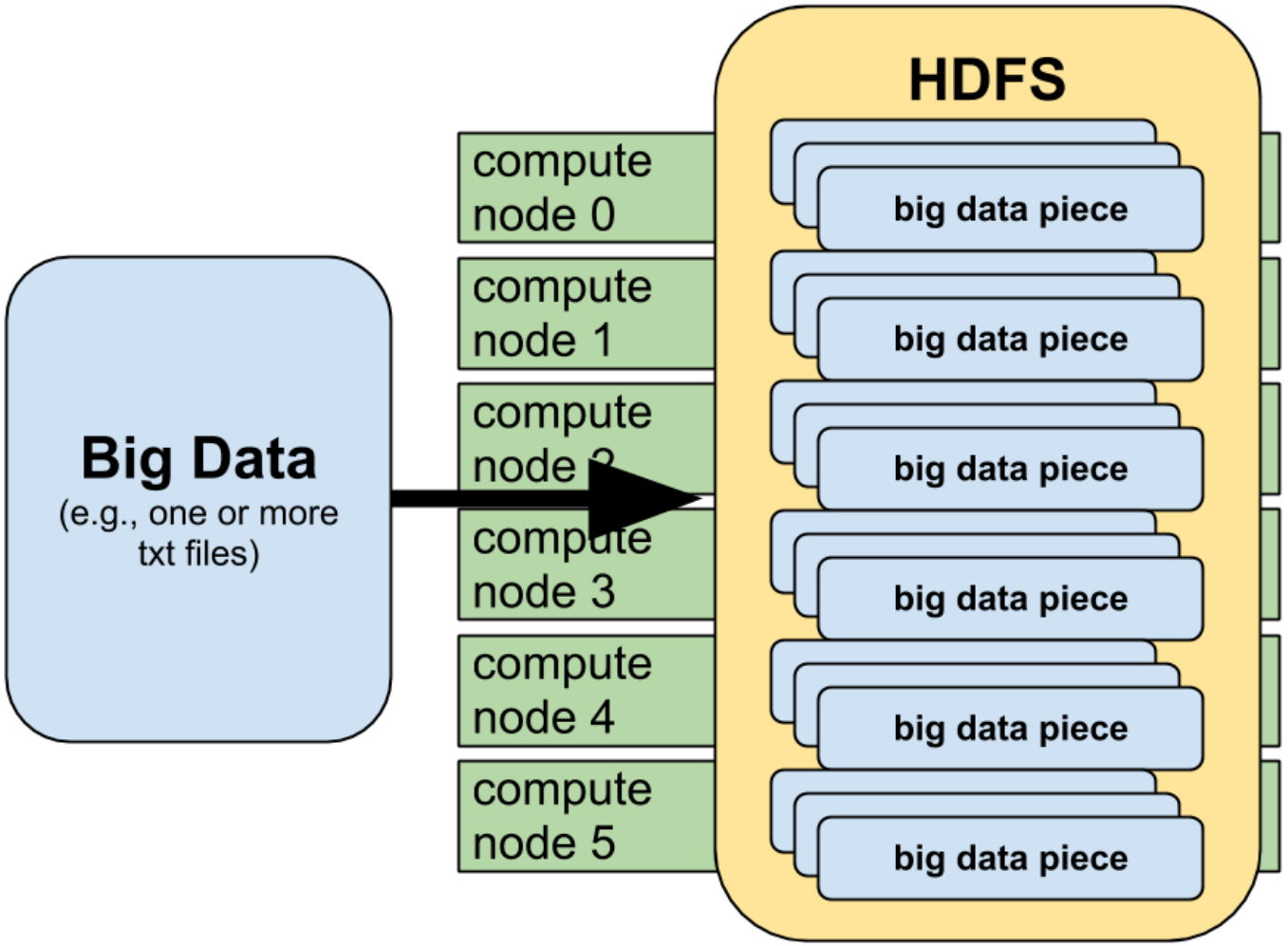
# Traditional Parallelism



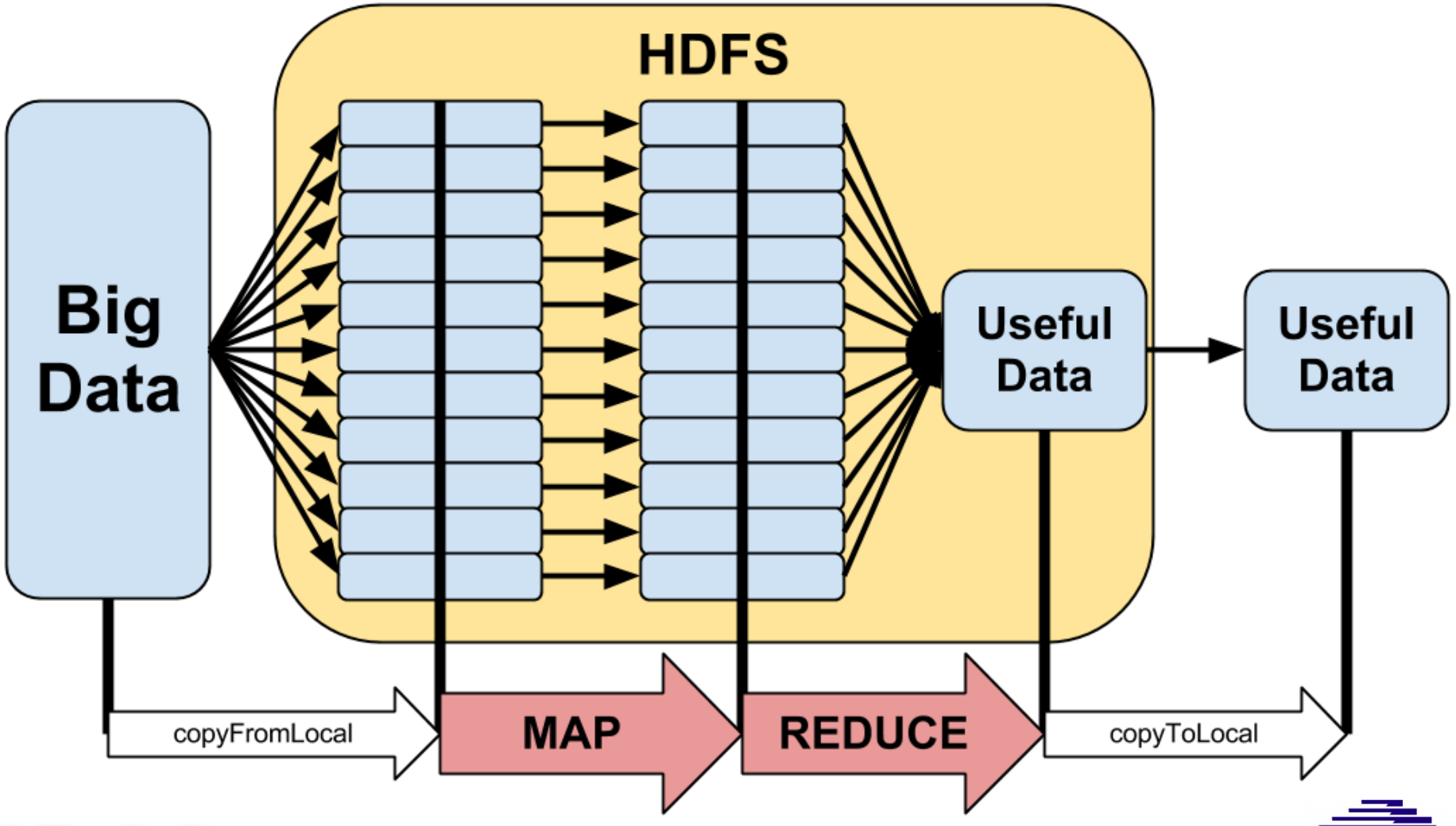
# Map/Reduce Parallelism



# Magic of HDFS



# Hadoop Workflow



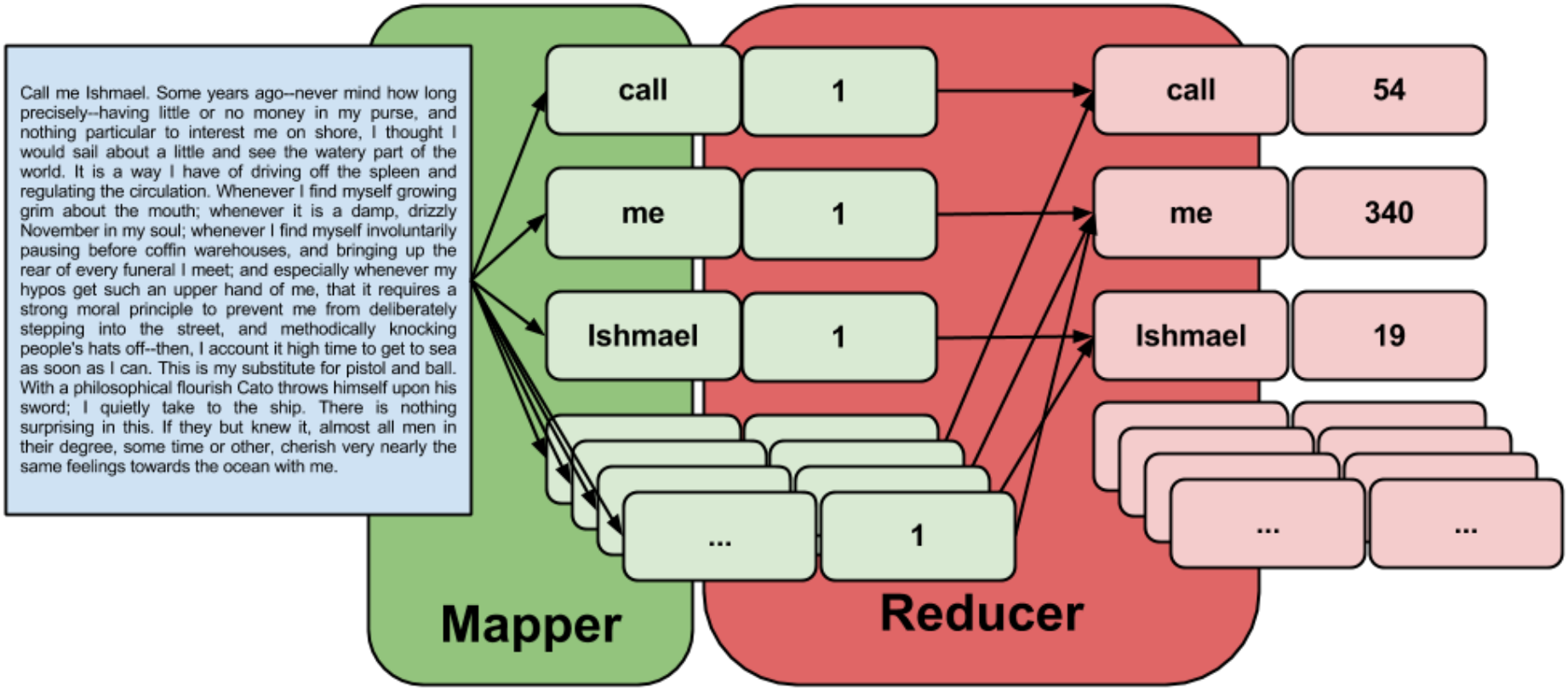
## Parallel Options for R – Map/Reduce-based Methods

# *WORDCOUNT EXAMPLES*

# *Map/Reduce (Hadoop) and R*

- **Hadoop streaming w/ R mappers/reducers**
  - most portable
  - most difficult (or least difficult)
  - you are the glue between R and Hadoop
- **Rhipe (**hree-***pay*****)**
  - least portable
  - comprehensive integration
  - R interacts with native Java application
- **RHadoop (rmr, rhdfs, rhbase)**
  - comprehensive integration
  - R interface to Hadoop streaming

# Wordcount Example



# *Hadoop with R - Streaming*

- **"Simplest" (most portable) method**
- **Uses R, Hadoop – you are the glue**

```
cat input.txt | mapper.R | sort | reducer.R > output.txt
```

provide these two scripts; Hadoop does the rest

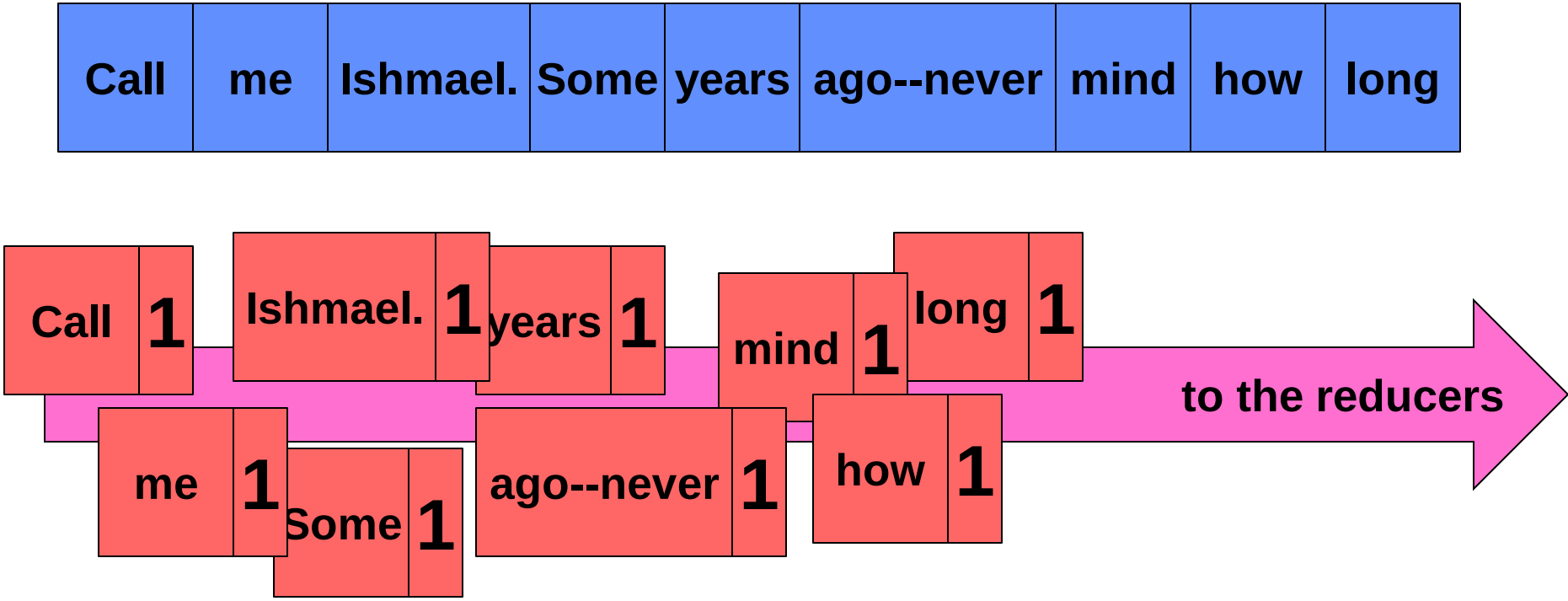
- **generalizable to any language you want**

# Wordcount: Hadoop streaming mapper

```
 1 emit.keyval = function(key, value) {
 2     cat(key,'\t',value,'\n',sep='')
 3 }
 4
 5 stdin <- file('stdin', open='r')
 6 while ( length(line <- readLines(stdin, n=1)) > 0 ) {
 7     line <- gsub('(^\\s+|\\s+$)', '', line)
 8     keys <- unlist(strsplit(line, split='\\s+'))
 9     value <- 1
10     lapply(keys, FUN=emit.keyval, value=value)
11 }
12 close(stdin)
```

# What One Mapper Does

Call me Ishmael.  Some years ago—never mind how long



# Reducer Loop

- **If this key is the same as the previous key,**
  - add this key's value to our running total.
- **Otherwise,**
  - print out the previous key's name and the running total,
  - reset our running total to 0,
  - add this key's value to the running total, and
  - "this key" is now considered the "previous key"

# Wordcount: Streaming Reducer (1/2)

```
1  last_key <- ""
2  running_total <- 0
3
4  stdin <- file('stdin', open='r')
5  while ( length(line <- readLines(stdin, n=1)) > 0 ) {
6      line <- gsub('(^\\s+)|(\\s+$)', '', line)
7      keyvalue <- unlist(strsplit(line, split='\t', fixed=TRUE))
8      this_key <- keyvalue[[1]]
9      value <- as.numeric(keyvalue[[2]])
10
11     if ( last_key == this_key ) {
12         running_total <- running_total + value
13     }
14     else {
```

# Wordcount: Streaming Reducer (2/2)

```
14      else {
15          if ( last_key != "" ) {
16              cat( paste(last_key,'\t',running_total,'\n',sep='') )
17          }
18          running_total <- value
19          last_key = this_key
20      }
21  }
22
23  if ( last_key == this_key ) {
24              cat( paste(last_key,'\t',running_total,'\n',sep='') )
25  }
26  close(stdin)
```

# Testing Mappers/Reducers

- 

```
$ head -n100 pg2701.txt |
  ./wordcount-streaming-mapper.R | sort |
  ./wordcount-streaming-reducer.R
...
with  5
word, 1
world.      1
www.gutenberg.org 1
you   3
You   1
```

# Launching Hadoop Streaming

```
$ hadoop dfs -copyFromLocal ./pg2701.txt mobydick.txt

$ hadoop jar \
 /opt/hadoop/contrib/streaming/hadoop-streaming-1.0.3.jar \
    -D mapred.reduce.tasks=2 \
    -mapper "Rscript $PWD/wordcount-streaming-mapper.R" \
    -reducer "Rscript $PWD/wordcount-streaming-reducer.R" \
    -input mobydick.txt \
    -output output

$ hadoop dfs -cat output/part-* > ./output.txt
```

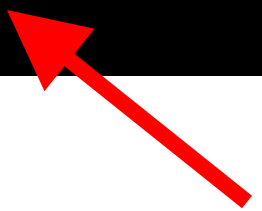
# *Hadoop with R - RHIPE*

- **Mapper, reducer written as** expression()**s**
- **Reads/writes R objects to HDFS natively**
- **All HDFS commands specified in R**
- **Running is easy:**

```
$ R CMD BATCH ./wordcount-rhipe.R
```

# RHIPE - Mapper

```
17 mapper <- expression( {
18     # 'map.values' = list containing each line of input file
19     lines <- gsub('(^\\s+|\\s+$)', '', map.values)
20     keys <- unlist(strsplit(lines, split='\\s+'))
21     value <- 1
22     lapply(keys, FUN=rhcollect, value=value)
23 } )
```



rhcollect **"emits" key/value pairs**

# RHIPE - Reducer

```
25 reducer <- expression(
26     # 'reduce.key' is equivalent to this_key and set by Rhipe
27     # 'reduce.values' is a list of values corresponding to this_key
28     # 'pre' is executed before we process a new reduce.key
29     # 'reduce' is executed for 'reduce.values'
30     # 'post' is executed once all reduce.values are processed
31     pre = {
32         running_total <- 0
33     },
34     reduce = {
35         running_total <- sum(running_total, unlist(reduce.values))
36     },
37     post = {
38         rhcollect(reduce.key, running_total)
39     }
40 )
```

# RHIPE – Job Launch

```
46 rhipe.results <- rhwatch(
47                          map=mapper, reduce=reducer,
48                          input=rhfmt(input.file.hdfs, type="text"),
49                          output=output.dir.hdfs,
50                          jobname='Wordcount',
51                          mapred=list(mapred.reduce.tasks=2))
52 results <- rhread(paste("/user/glock/output/part-*", sep = ""))
```

# RHIPE - Submit Script

```
$ hadoop dfs -copyFromLocal ./pg2701.txt mobydick.txt
$ hadoop jar \
 /opt/hadoop/contrib/streaming/hadoop-streaming-1.0.3.jar \
    -D mapred.reduce.tasks=2 \
    -mapper "Rscript $PWD/wordcount-streaming-mapper.R" \
    -reducer "Rscript $PWD/wordcount-streaming-reducer.R" \
    -input mobydick.txt \
    -output output
$ hadoop dfs -cat output/part-* > ./output
```

## Hadoop streaming vs. RHIPE

```
$ R CMD BATCH ./wordcount-rhipe.R
```

# *Hadoop with R - RHadoop*

- **Mapper, reducer written as** function()**s**
- **Reads/writes R objects to HDFS natively**
- **All HDFS commands specified in R**
- **Running is easy:**

```
$ R CMD BATCH ./wordcount-rhadoop.R
```

# RHadoop – Mapper

```
mapper <- function( keys, lines ) {
    lines <- gsub('(^\\s+|\\s+$)', '', lines)
    keys <- unlist(strsplit(lines, split='\\s+'))
    value <- 1
    lapply(keys, FUN=keyval, v=value)
}
```

# RHadoop – Reducer

```
reducer <- function( key, values ) {
    running_total <- sum( unlist(values) )
    keyval(key, running_total)
}
```

# RHadoop – Job Launch

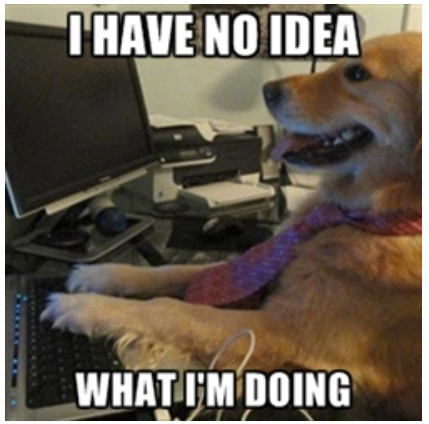
```
rmr.results <- mapreduce(
    map=mapper, reduce=reducer,
    input=input.file.hdfs, input.format = "text",
    output=output.dir.hdfs,
    backend.parameters=list("mapred.reduce.tasks=2"))
```

# R and Hadoop - PROs

- File I/O no longer serial so it is not a bottleneck
- RAM on master node is no longer limiting
- Mappers/reducers can use all R libraries
- Hadoop foundation provides huge scalability

# RHadoop/Rhipe - CONs

- **APIs are changing**
  - We must use older RHadoop to work on Gordon, but
    - its API is inconsistent with modern documentation
    - mapreduce(..., input.format =...) became mapreduce(..., textinputformat =...)
  - Rhipe developers don't seem to care about compatibility
    - rhmr() turned into rhwatch() sometime between 0.69 and 0.72
    - examples, documentation mostly for rhmr()
- **If you have to install these yourself...**



# Additional Resources


## Parallel R

McCallum and Weston (2011)

O'Reilly Media

## Us here at SDSC

- Official Hadoop on Gordon Guide: http://www.sdsc.edu/us/resources/gordon/gordon_hadoop.html
- Unofficial Parallel R and Hadoop Streaming: http://users.sdsc.edu/~glockwood/comp
- help@xsede.org comes to us (for XSEDE users)