High Performance Computing with R

Drew Schmidt

April 16 and 23, 2014

http://r-pbd.org/CS505

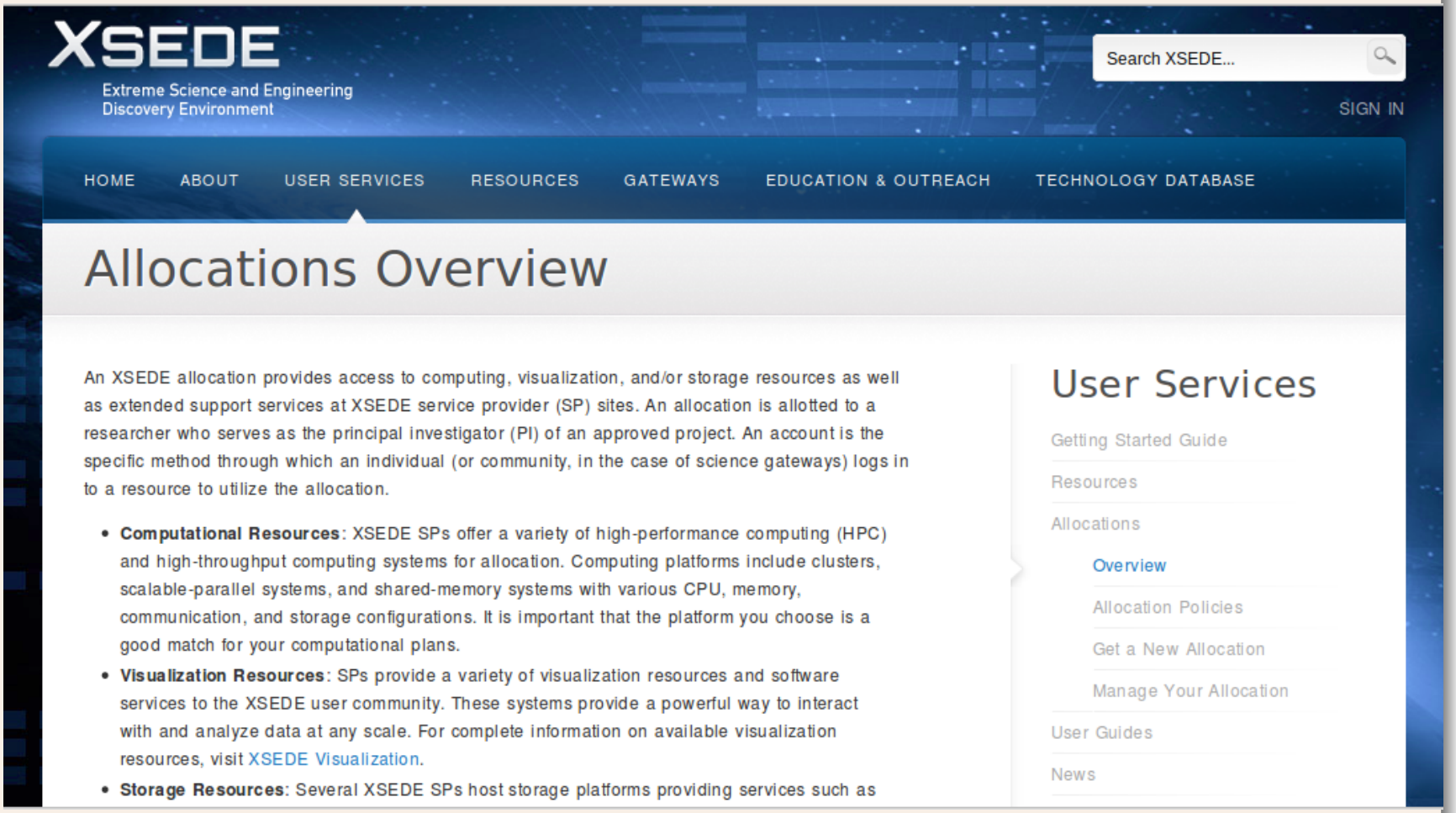
Contents

1. Introduction
   - Compute Resources
   - Data Science
   - Why R?

Compute Resources

- Your laptop.
- Your own server.
- The cloud.
- NSF resources.

XSEDE



https://www.xsede.org/allocations

XSEDE

- (+) Free*!!!
- (+) Access to *massive* compute resources.
- (+) OS image and software managed by others.
- (-) 1-3 month turnaround for new applications.
- (-) Application consists of more than "here's my credit card."
- (-) Some restrictions apply.

Computational Science vs Data Science

- *Computational science* generally refers to traditional (mostly simulation) sciences.
- *Data science*: Statistics, machine learning, data mining, knowledge discovery, ...

Data Science

- Real data is a mess.
- I/O is expensive.
- Analytics algorithms are slow and $O(n^3)$.

What is R?

- *lingua franca* for data analytics and statistical computing.
- Part programming language, part data analysis package.
- Free (GPL).
- Syntax designed for data.

History

- Dialect of S (Bell Labs), John Chambers.
- S: May 5, 1976.
- R: Ross Ihaka, Robert Gentleman, 1996.
- *R: A Language for Data Analysis and Graphics*

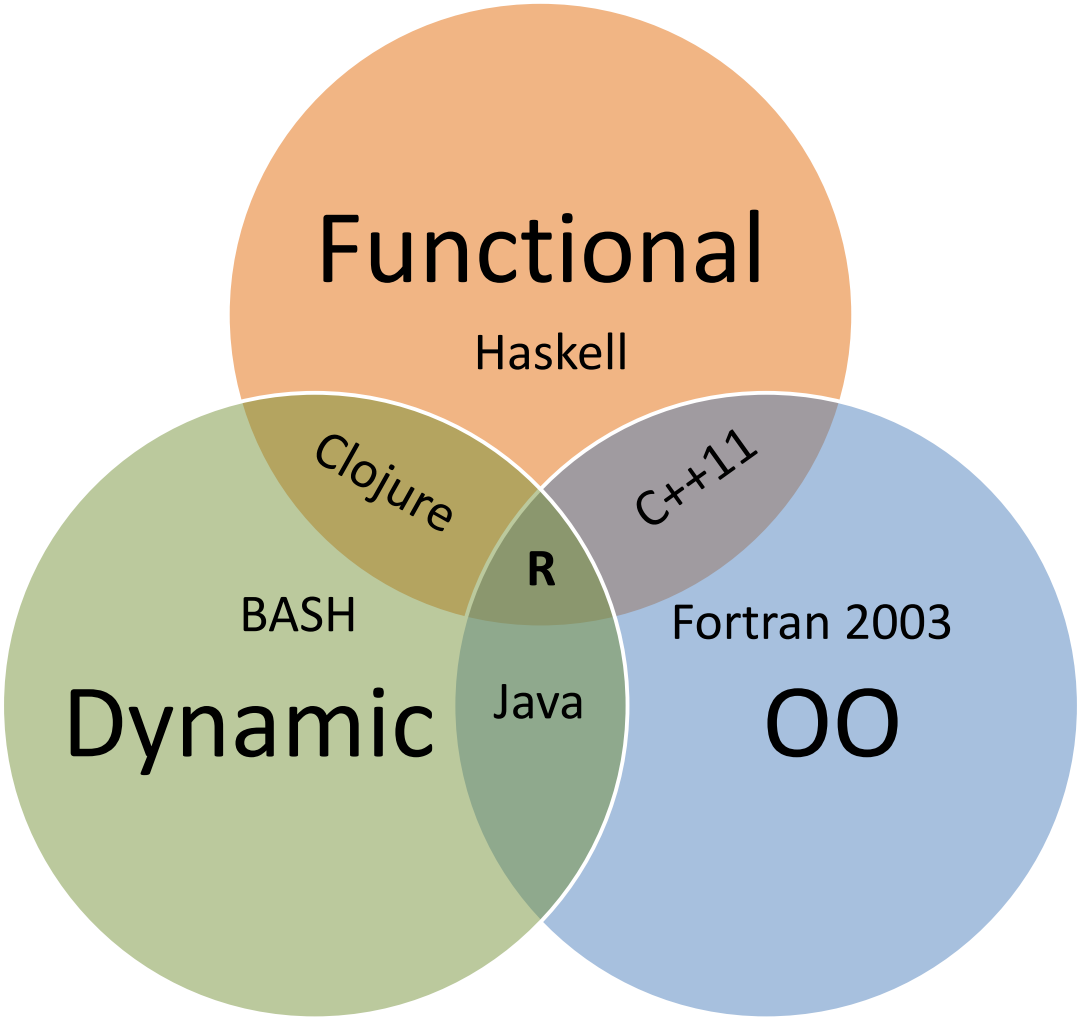
The CRAN

- Comprehensive R Archive Network
- Very high standard of quality.
- Over 5000 packages.
- Implemented by analytics experts.

Who uses R?


ORACLE
MERCK
KICKSTARTER
Bank of America
okcupid
The New York Times
Shell
Pfizer
ORBITZ
mozilla FOUNDATION
LLOYD'S
Google
ebay
bing
Ford

Language Paradigms



Data Types

- Storage: logical, int, double, double complex, character
- Structures: vector, matrix, array, list, dataframe
- Caveats: (Logical) `TRUE`, `FALSE`, `NA`

For the remainder of the tutorial, we will restrict ourselves to real number matrix computations.

2. R Basics
   - R Basics
   - Basic Numerical Operations in R
   - R Syntax for Data Science: Not A Matlab Clone!

Basics (1 of 2)

- The default method is to print:

```
R> sum
function (..., na.rm = FALSE)  .Primitive("sum")
```

- Use <- for assignment:

```
R> x <- 1
R> x+1
[1] 2
```

- Naming rules: mostly like C.
- R is case sensitive.
- We use . the way most languages use _, e.g., `La.svd()` instead of `La_svd()`.
- We use $ (sometimes @) the way most languages use .

Basics (2 of 2)

- Use ? or ?? to search help

```
R> ?set.seed
R> ?comm.set.seed
No documentation for  comm.set.seed  in
    specified packages and libraries:
you could try    ??comm.set.seed
R> ??comm.set.seed
```

Addons and Extras

R has the Comprehensive R Archive Network (CRAN), which is a package repository like CTAN and CPAN.

From R

```
install.packages("pbdMPI") # install
library(pbdMPI)            # load
```

From Shell

```
R CMD INSTALL pbdMPI_0.1-6.tar.gz
```

Lists (1 of 1)

```
R> l <- list(a=1, b="a")
R> l
$a
[1] 1

$b
[1] "a"

R> l$a
[1] 1

R> list(x=list(a=1, b="a"), y=TRUE)
$x
$x$a
[1] 1

$x$b
[1] "a"


$y
[1] TRUE
```

Vectors and Matrices (1 of 2)

```
R> c(1, 2, 3, 4, 5, 6)
[1] 1 2 3 4 5 6

R> matrix(1:6, nrow=2, ncol=3)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

R> x <- matrix(1:6, nrow=2, ncol=3)

R> x[, -1]
     [,1] [,2]
[1,]    3    5
[2,]    4    6

R> x[1, 1:2]
[1] 1 3
```

Vectors and Matrices (2 of 2)

```
R> dim(x)
[1] 2 3

R> dim(x) <- NULL
R> x
[1] 1 2 3 4 5 6

R> dim(x) <- c(3,2)
R> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Vector and Matrix Arithmetic (1 of 2)

```
R> 1:4 + 4:1
[1] 5 5 5 5

R> x <- matrix(0, nrow=2, ncol=3)

R> x + 1
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1

R> x + 1:3
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    2    1    3
```

Vector and Matrix Arithmetic (2 of 2)

```
R> x <- matrix(1:6, nrow=2)

R> x*x
     [,1] [,2] [,3]
[1,]    1    9   25
[2,]    4   16   36

R> x %*% x
Error in x %*% x : non-conformable arguments

R> t(x) %*% x
     [,1] [,2] [,3]
[1,]    5   11   17
[2,]   11   25   39
[3,]   17   39   61

R> crossprod(x)
     [,1] [,2] [,3]
[1,]    5   11   17
[2,]   11   25   39
[3,]   17   39   61
```

Linear Algebra (1 of 2): Matrix Inverse

$$x_{n\times n} \text{ invertible} \iff \exists y_{n\times n}\,(xy = yx = Id_{n\times n})$$

```
R> x <- matrix(rnorm(5*5), nrow=5)
R> y <- solve(x)

R> round(x %*% y)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

Linear Algebra (2 of 2): Singular Value Decomposition

$$x = U\Sigma V^T$$

```
R> x <- matrix(rnorm(2*3), nrow=3)
R> svd(x)
$d
[1] 2.4050716 0.3105008

$u
           [,1]       [,2]
[1,] 0.8582569 -0.1701879
[2,] 0.2885390  0.9402076
[3,] 0.4244295 -0.2950353

$v
             [,1]        [,2]
[1,] -0.05024326 -0.99873701
[2,] -0.99873701  0.05024326
```

More than just a Matlab clone...

- Data science (machine learning, statistics, data mining, ...) is mostly matrix algebra.

  So what about Matlab/Python/Julia/...?
- The one you prefer depends more on your "religion" rather than differences in capabilities.
- As a *data analysis* package, R is king.

Simple Statistics (1 of 2): Summary Statistics

```
R> x <- matrix(rnorm(30, mean=10, sd=3), nrow=10)

R> mean(x)
[1] 9.825177

R> median(x)
[1] 9.919243

R> sd(as.vector(x))
[1] 3.239388

R> colMeans(x)
[1]  9.661822 10.654686  9.159025

R> apply(x, MARGIN=2, FUN=sd)
[1] 2.101059 3.377347 4.087131
```

Simple Statistics (2 of 2): Sample Covariance

$$cov(x_{n\times p}) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu_x)(x_i - \mu_x)^T$$

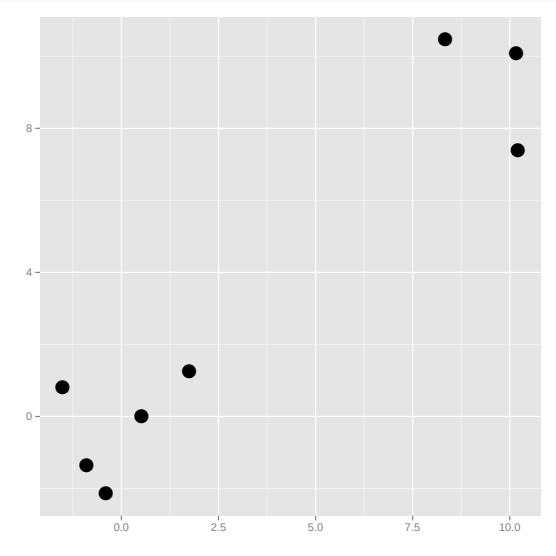
```
x <- matrix(rnorm(30), nrow=10)

# least recommended
cm <- colMeans(x)
crossprod(sweep(x, MARGIN=2, STATS=cm))

# less recommended
crossprod(scale(x, center=TRUE, scale=FALSE))

# recommended
cov(x)
```

Advanced Statistics (1 of 2): Principal Components

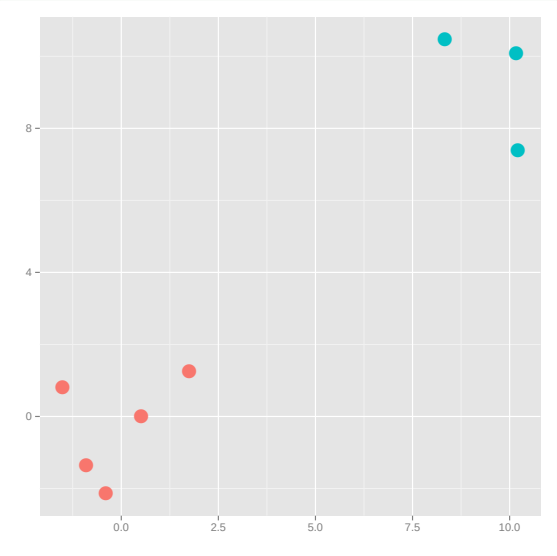
PCA = centering + scaling + rotation (via SVD)

```
R> x <- matrix(rnorm(30), nrow=10)

R> prcomp(x, retx=TRUE, scale=TRUE)
Standard deviations:
[1] 1.1203373 1.0617440 0.7858397

Rotation:
              PC1        PC2        PC3
[1,]  0.71697825 -0.3275365  0.6153552
[2,] -0.03382385  0.8653562  0.5000147
[3,]  0.69627447  0.3793133 -0.6093630
```

Advanced Statistics (2 of 2): k-Means Clustering



→




```
R> x <- rbind(matrix(rnorm(5*2, mean=0), ncol=2),
              matrix(rnorm(3*2, mean=10), ncol=2))
```

Advanced Statistics (2 of 2): k-Means Clustering

```
R> kmeans(x, centers=2)
K-means clustering with 2 clusters of sizes 5, 3

Cluster means:
        [,1]       [,2]
1 -0.1080612 -0.2827576
2  9.5695365  9.3191892

Clustering vector:
[1] 1 1 1 1 1 2 2 2

Within cluster sum of squares by cluster:
[1] 14.675072  7.912641
 (between_SS / total_SS =  93.9 %)

Available components:

[1] "cluster"      "centers"      "totss"
    "withinss"     "tot.withinss"
[6] "betweenss"    "size"
```

3. Nice Things R Does
   - CRAN Packages
   - Plotting

CRAN

- Packages developed by the community.
- Solve all sorts of problems.
- Finding *a* solution usually not the problem...

Bayesian	Bayesian Inference
ChemPhys	Chemometrics and Computational Physics
ClinicalTrials	Clinical Trial Design, Monitoring, and Analysis
Cluster	Cluster Analysis & Finite Mixture Models
DifferentialEquations	Differential Equations
Distributions	Probability Distributions
Econometrics	Computational Econometrics
Environmetrics	Analysis of Ecological and Environmental Data
ExperimentalDesign	Design of Experiments (DoE) & Analysis of Experimental Data
Finance	Empirical Finance
Genetics	Statistical Genetics
Graphics	Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization
HighPerformanceComputing	High-Performance and Parallel Computing with R
MachineLearning	Machine Learning & Statistical Learning
MedicalImaging	Medical Image Analysis
MetaAnalysis	Meta-Analysis
Multivariate	Multivariate Statistics
NaturalLanguageProcessing	Natural Language Processing
NumericalMathematics	Numerical Mathematics
OfficialStatistics	Official Statistics & Survey Methodology
Optimization	Optimization and Mathematical Programming
Pharmacokinetics	Analysis of Pharmacokinetic Data
Phylogenetics	Phylogenetics, Especially Comparative Methods
Psychometrics	Psychometric Models and Methods
ReproducibleResearch	Reproducible Research
Robust	Robust Statistical Methods
SocialSciences	Statistics for the Social Sciences
Spatial	Analysis of Spatial Data
SpatioTemporal	Handling and Analyzing Spatio-Temporal Data
Survival	Survival Analysis
TimeSeries	Time Series Analysis
WebTechnologies	Web Technologies and Services
gR	gRaphical Models in R

CRAN Taskviews: http://cran.r-project.org/web/views/

Plotting Data in R

1. Base Graphics: Easy for simple things, hard for complex things
2. Grid: "Assembly language for graphics"
3. ggplot2: Powerful and flexible, takes some time to learn
4. Lattice: "Like" ggplot2 in scope, but very different

   ...
5. And about 20 other packages: http://cran.r-project.org/web/views/Graphics.html

ggplot2

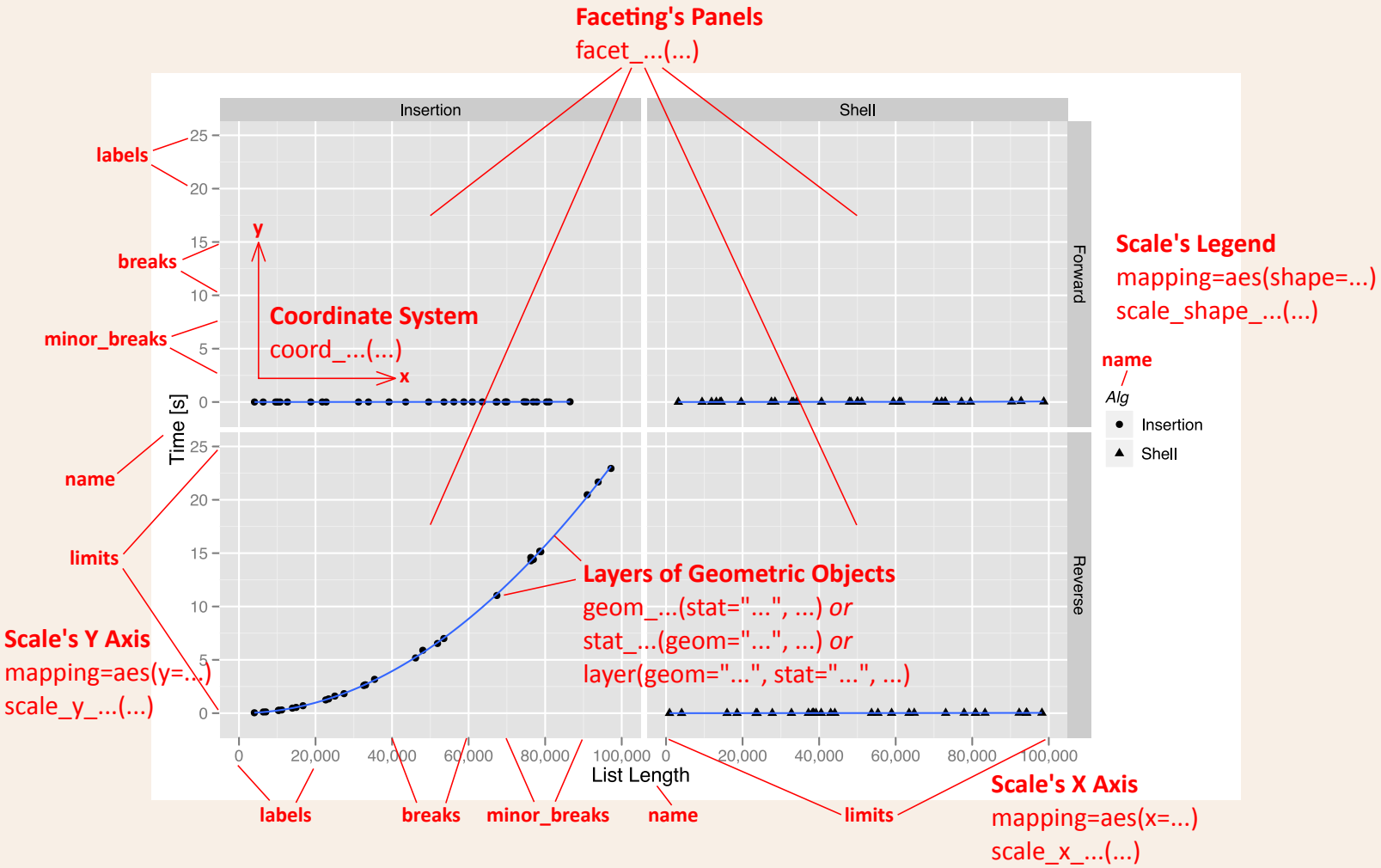
- Package written by Hadley Wickham in 2005 for R
- Implementation of Lee Wilkinson's Grammar of Graphics
- Highly abstract and powerful way of plotting data
- Written entirely in R

The Four Components to a Graphic in the Grammar of Graphics Schema

1. Geoms: "Physical components", e.g. point, path, line, polygon, ...
2. Aesthetics: "Visual cues", e.g. size, rotation, thickness, gradient, shape, color, ...
3. Coordinates: Just what it sounds like: rectangular, polar, ...
4. Faceting: Coplotting — more on this later

Customization

Everything in a ggplot2 plot can be customized:



Geom and Stat Functions

There are many "layering" functions:

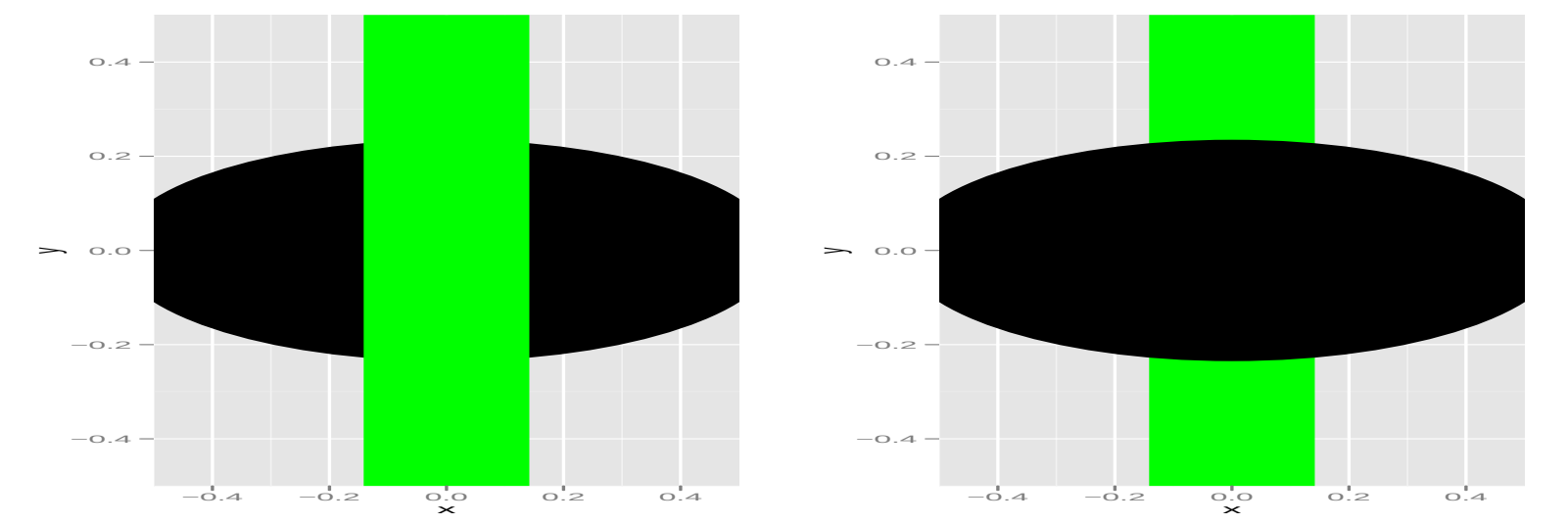
Geom Functions				
geom_abline	geom_bar	geom_blank	geom_contour	geom_density
geom_errorbar	geom_freqpoly	geom_histogram	geom_jitter	geom_linerange
geom_point	geom_polygon	geom_rect	geom_rug	geom_smooth
geom_text	geom_vline	geom_area	geom_bin2d	geom_boxplot
geom_crossbar	geom_density2d	geom_errorbarh	geom_hex	geom_hline
geom_line	geom_path	geom_pointrange	geom_quantile	geom_ribbon
geom_segment	geom_step	geom_tile		

Stat Functions				
stat_abline	stat_bin2d	stat_boxplot	stat_density	stat_function
stat_identity	stat_quantile	stat_spoke	stat_summary	stat_vline
stat_bin	stat_binhex	stat_contour	stat_density2d	stat_hline
stat_qq	stat_smooth	stat_sum	stat_unique	

For explanations and examples, see the ggplot2 reference manual http://had.co.nz/ggplot2/

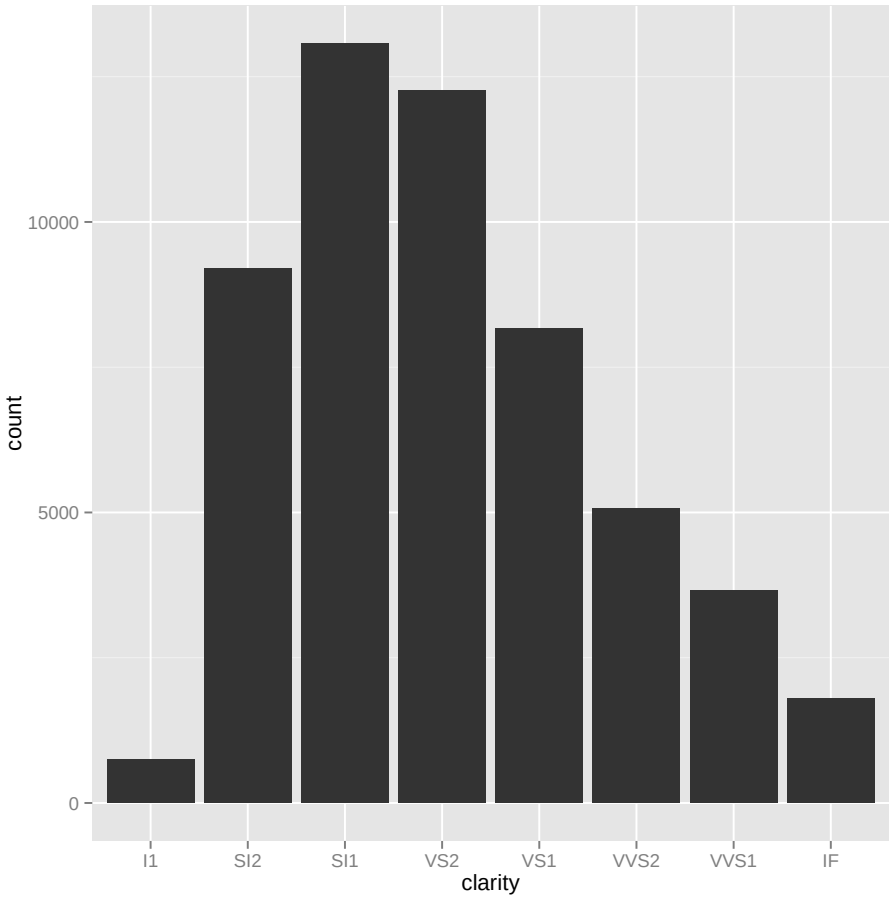
Adding Layers is Not Necessarily Commutative

```
# Plot points layer then lines layer on top
g + geom_point() + geom_line()

# Plot lines layer then points layer on top
g + geom_line() + geom_point()
```



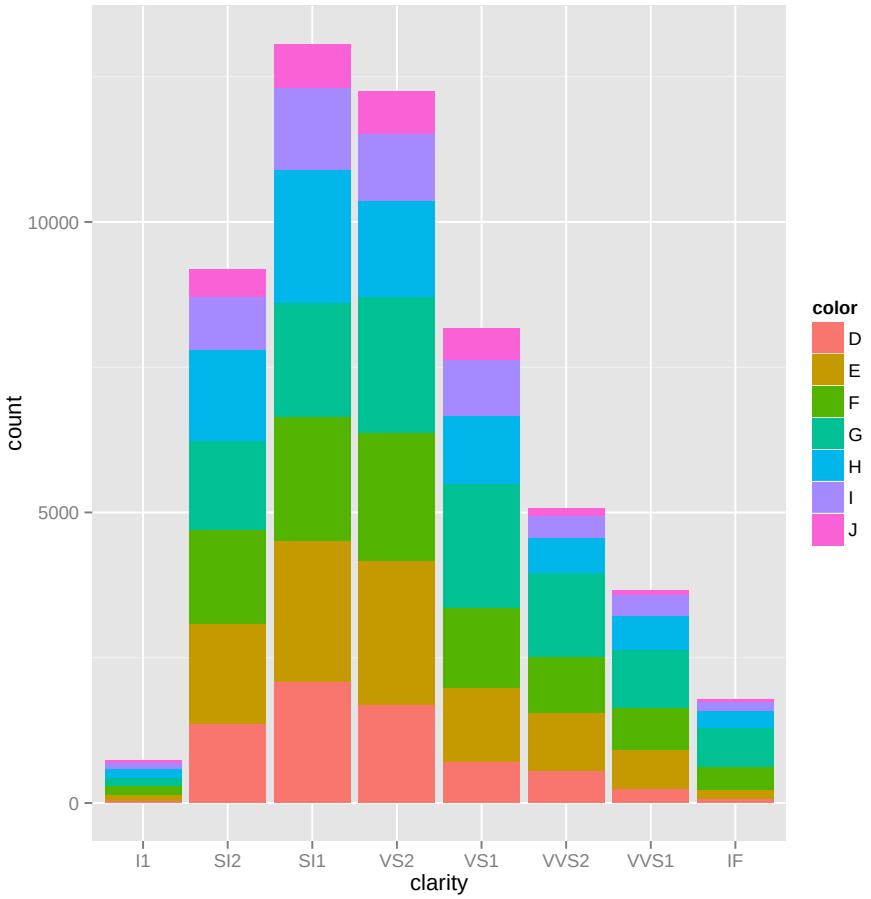
Simple Barplot

```
library(ggplot2)
data(diamonds)

ggplot(data=diamonds,
    aes(x=clarity) )
    + geom_bar()
```



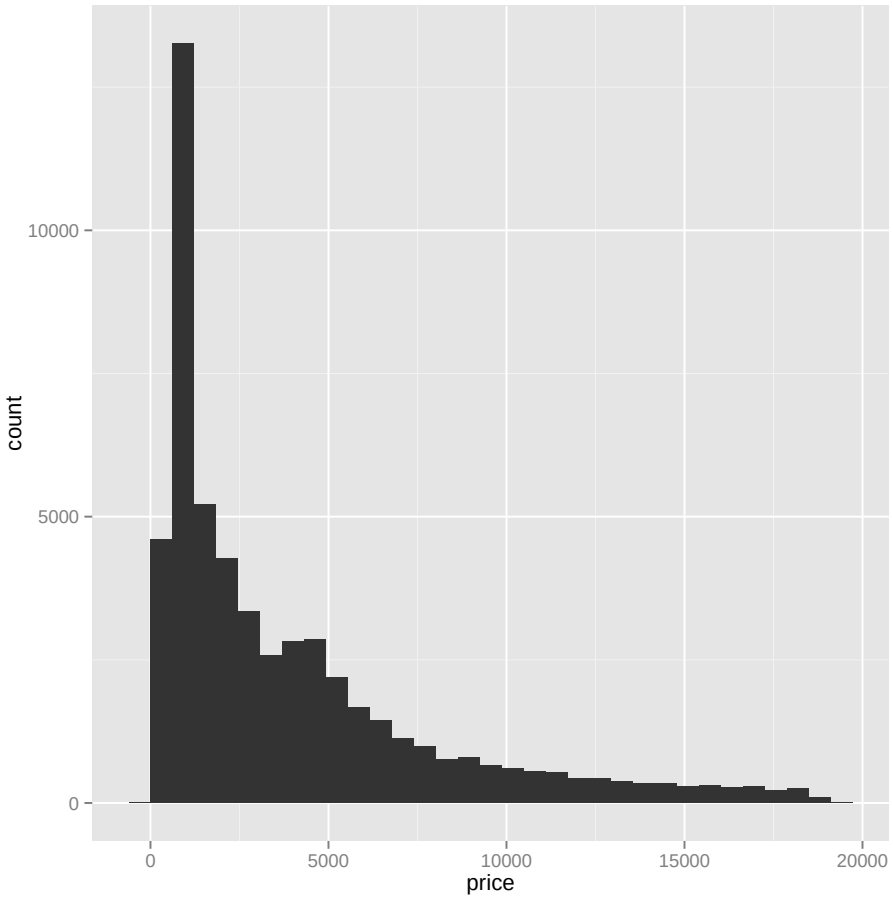
Color-by-group Barplot

```
library(ggplot2)
data(diamonds)

ggplot(data=diamonds,
    aes(x=clarity,
    fill=color) ) +
    geom_bar()
```



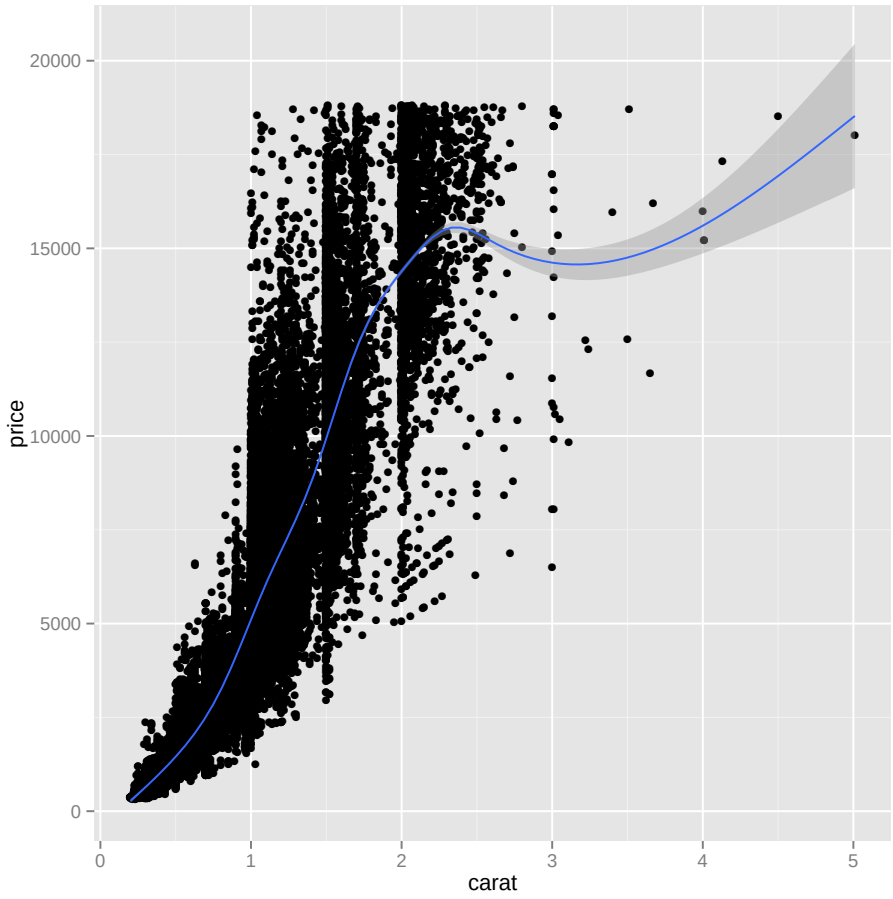
Histogram

```
library(ggplot2)
data(diamonds)

ggplot(data=diamonds,
    aes(x=price) ) +
    geom_histogram()
```



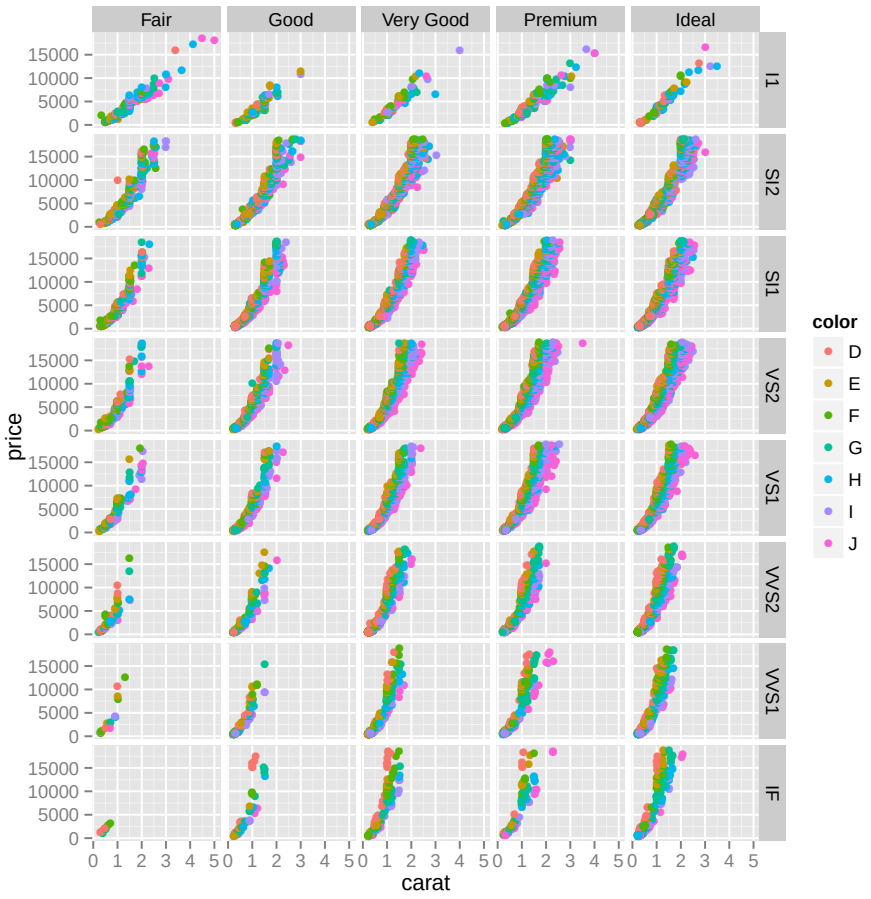
Scatterplot with LOESS Fit

```
library(ggplot2)
data(diamonds)

g <- ggplot(data =
    diamonds, aes(x
    = clarity, y =
    carat))

g + geom_point() +
    geom_smooth()
```



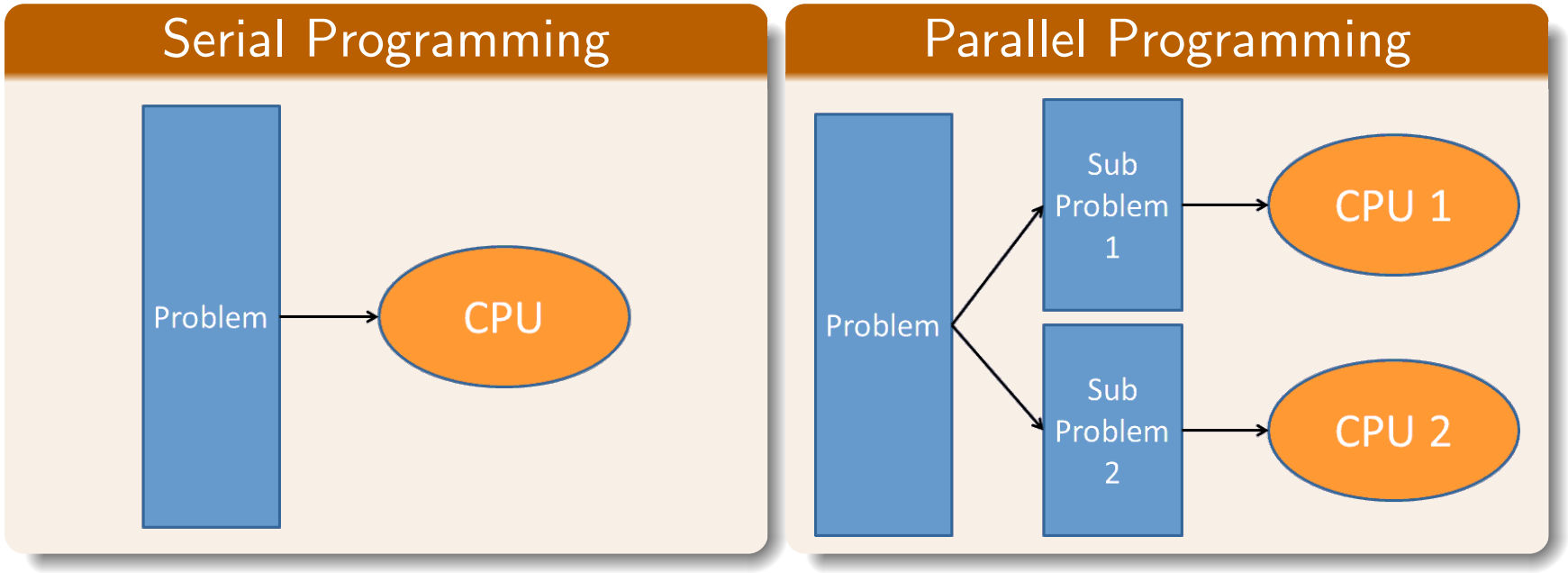
Faceting by Cut

```
library(ggplot2)
data(diamonds)

g <- ggplot(data =
    diamonds, aes(x
    = clarity, y =
    carat))

g +
    geom_point(aes(color
    = color)) +
    facet_grid(clarity
    ~ cut)
```



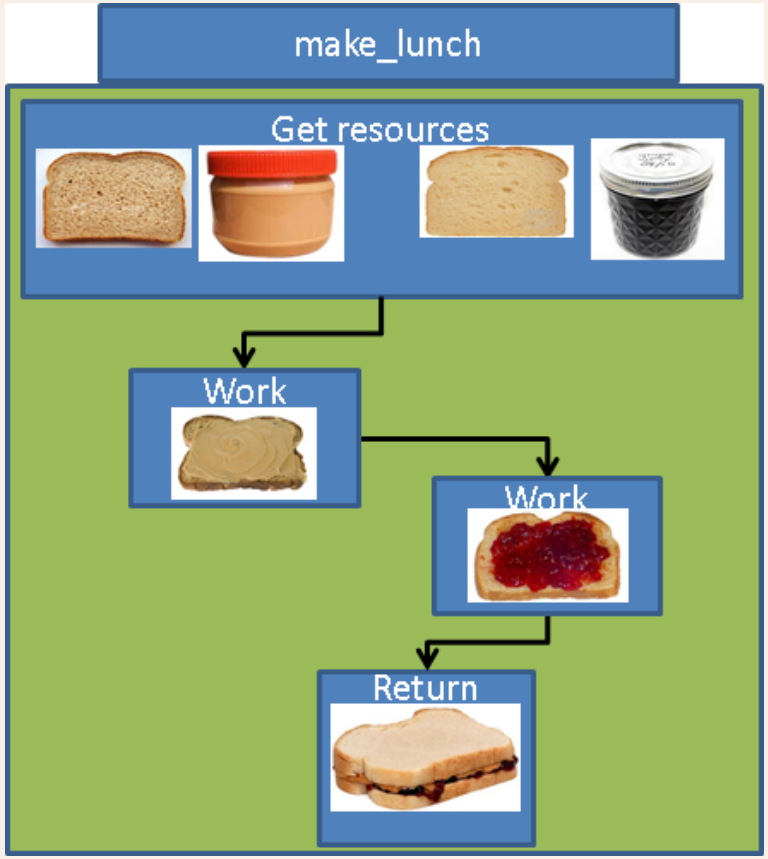
4. An Overview of Parallelism
   - Terminology: Parallelism
   - Choice of BLAS Library
   - Guidelines

Parallelism

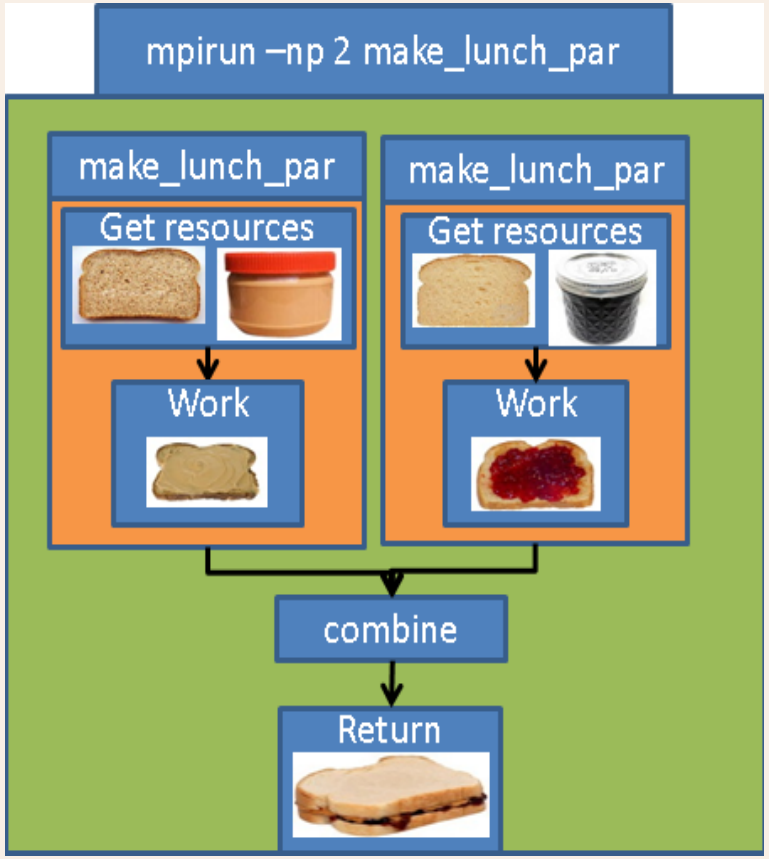


Parallelism

Serial Programming



Parallel Programming



Parallel Programming Vocabulary: Difficulty in Parallelism

1. *Implicit parallelism*: Parallel details hidden from user
   Example: Using multi-threaded BLAS
2. *Explicit parallelism*: Some assembly required...
   Example: Using the `mclapply()` from the **parallel** package
3. *Embarrassingly Parallel* or *loosely coupled*: Obvious how to make parallel; lots of independence in computations.
   Example: Fit two independent models in parallel.
4. *Tightly Coupled*: Opposite of embarrassingly parallel; lots of dependence in computations.
   Example: Speed up model fitting for one model.

Speedup

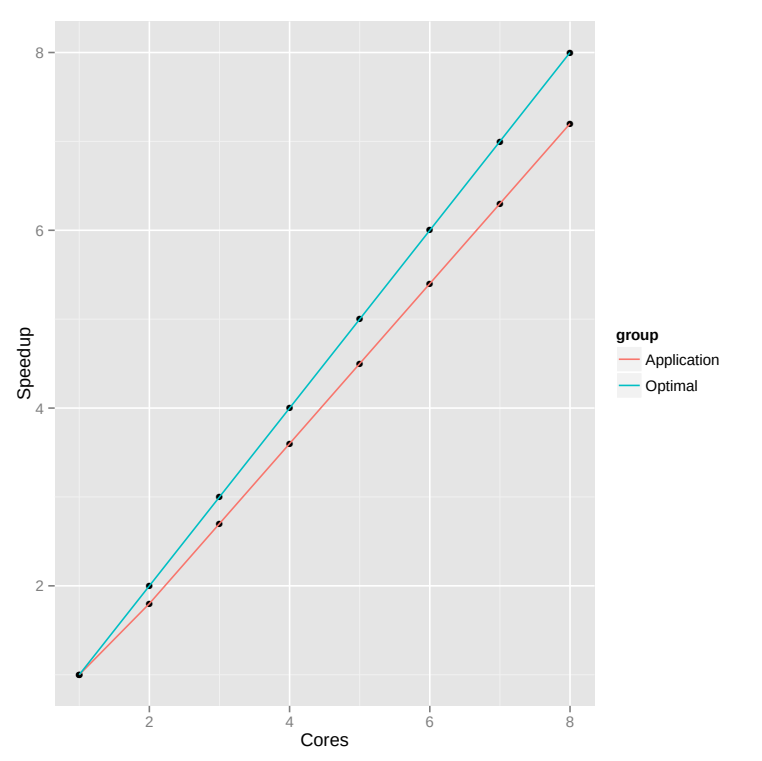
- *Wallclock Time*: Time of the clock on the wall from start to finish
- *Speedup*: unitless measure of improvement; more is better.

$$S_{n_1,n_2} = \frac{\text{Time for } n_1 \text{ cores}}{\text{Time for } n_2 \text{ cores}}$$

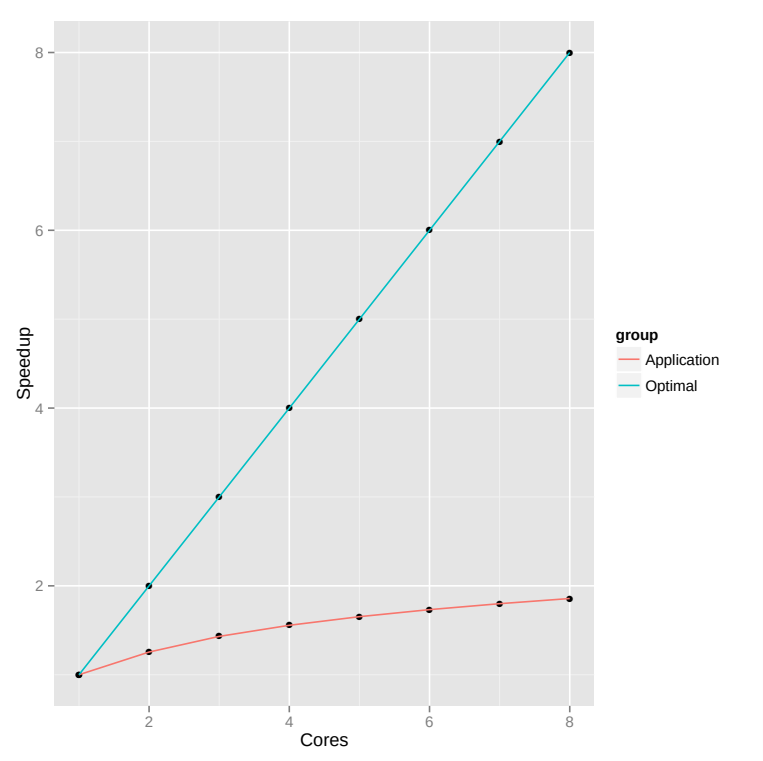
  - n_1 is often taken to be 1
  - In this case, comparing parallel algorithm to serial algorithm

Speedup

Good Speedup



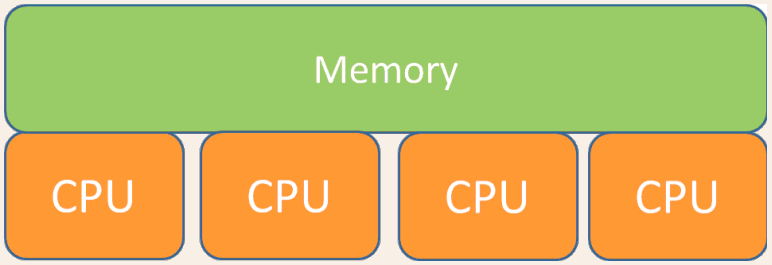
Bad Speedup



Shared and Distributed Memory Machines

Shared Memory

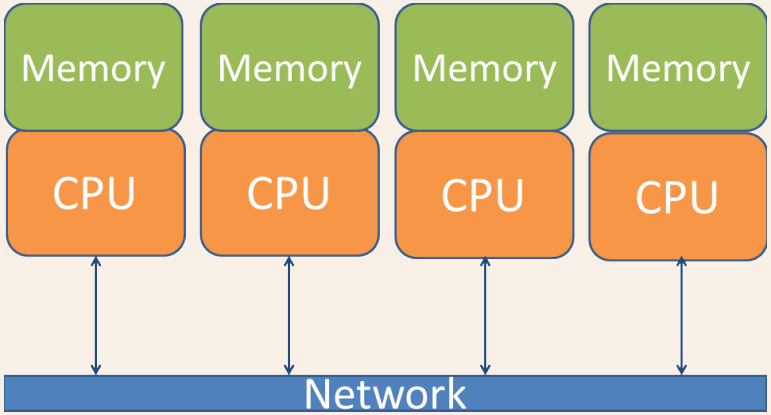
Direct access to read/change memory (one node)



Examples: laptop, GPU, MIC

Distributed

No direct access to read/change memory (many nodes); requires communication



Examples: cluster, server, supercomputer

Shared and Distributed Memory Machines

Shared Memory Machines

Thousands of cores

Nautilus, University of Tennessee
1024 cores
4 TB RAM

Distributed Memory Machines

Hundreds of thousands of cores

Kraken, University of Tennessee
112,896 cores
147 TB RAM

Shared and Distributed Programming from R

Shared Memory

Examples: **parallel**, **snow**, **foreach**, **gputools**, **HiPLARM**

Distributed

Examples: **pbdR**, **Rmpi**, **RHadoop**, **RHIPE**

CRAN HPC Task View

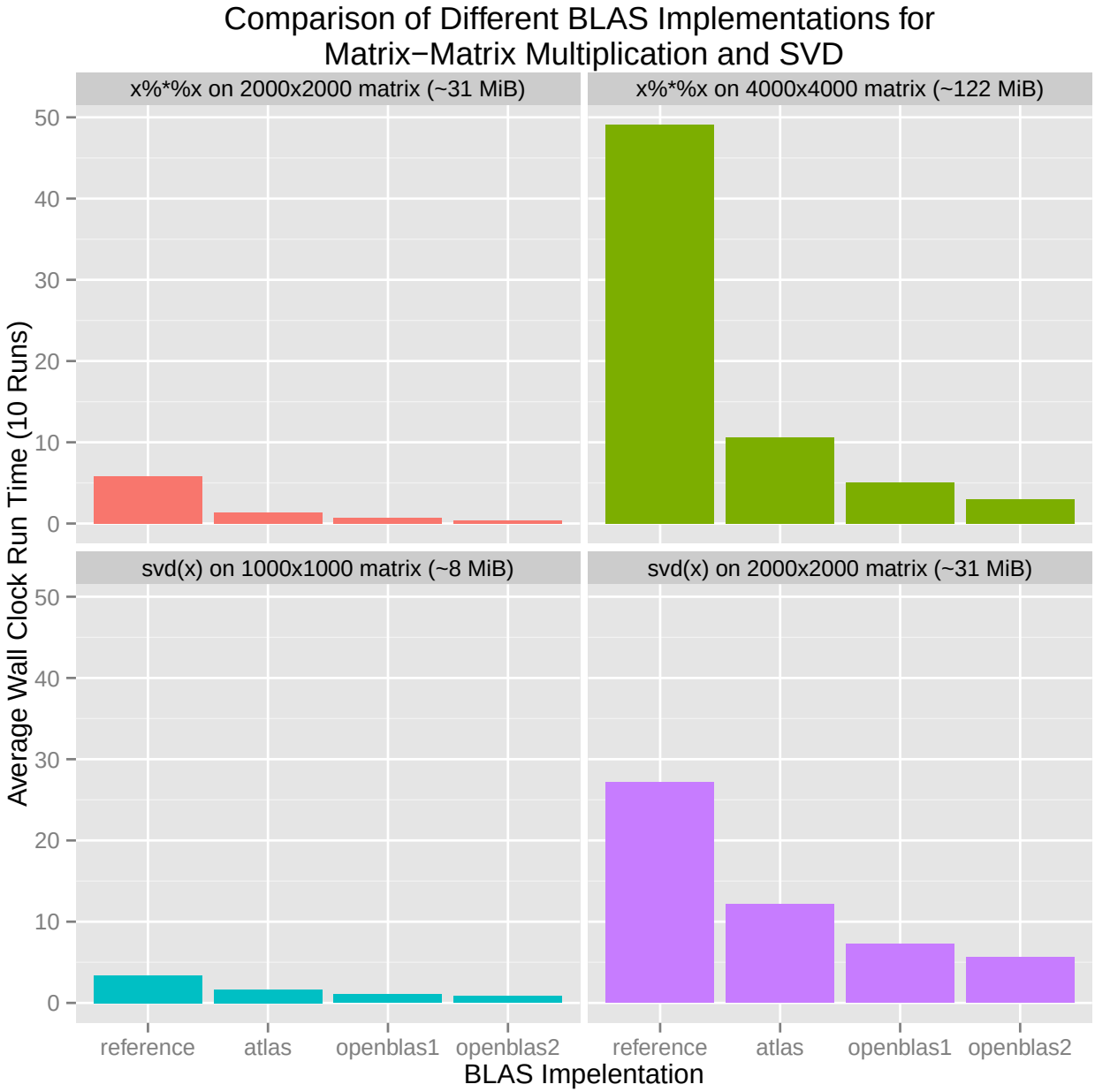
For more examples, see: http://cran.r-project.org/web/views/HighPerformanceComputing.html

The BLAS

- Basic Linear Algebra Subprograms.
- Simple vector-vector (level 1), matrix-vector (level 2), and matrix-matrix (level 3).
- R uses BLAS (and LAPACK) for most linear algebra operations.
- There are different implementations available, with massively different performance.
- Several multithreaded BLAS libraries exist.

Benchmark

```
set.seed(1234)
m <- 2000
n <- 2000
x <- matrix(
  rnorm(m*n),
  m, n)

object.size(x)

library(rbenchmark)

benchmark(x%*%x)
benchmark(svd(x))
```



Using openblas

On Debian and derivatives:

```
sudo apt-get install libopenblas-dev
sudo update-alternatives --config libblas.so.3
```

Warning: doesn't play nice with the **parallel** package!

Portability

- Not all packages (or methods within a package) support all OS's.
- In the HPC world, that usually means "doesn't work on Windows".

RNG's in Parallel

- Be careful!
- Aided by **rlecuyer**, **rsprng**, and **doRNG** packages.

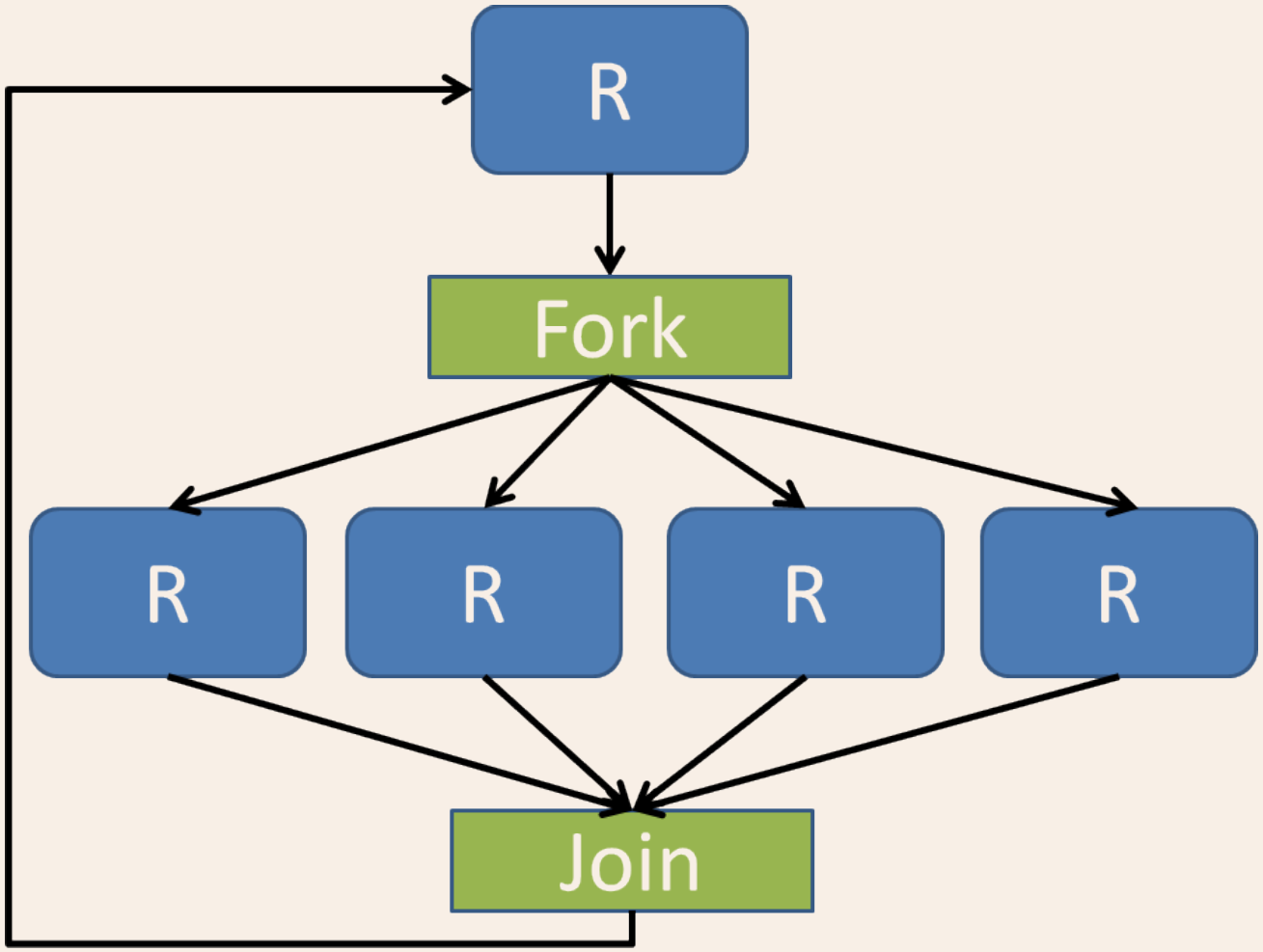
5. Shared Memory Parallelism in R
   - The parallel Package
   - The foreach Package

The parallel Package

- Comes with R $\geq$ 2.14.0
- Includes **multicore** + most of **snow**.
- As such, has 2 disjoint interfaces.

The parallel Package: multicore

Operates on fork/join paradigm.



The parallel Package: multicore

- (+) Data copied to child on write (handled by OS)
- (+) Very efficient.
- (-) No Windows support.
- (-) Not as efficient as threads.

The parallel Package: multicore

```
mclapply(X, FUN, ...,
  mc.preschedule=TRUE, mc.set.seed=TRUE,
  mc.silent=FALSE, mc.cores=getOption("mc.cores", 2L),
  mc.cleanup=TRUE, mc.allow.recursive=TRUE)
```

```
x <- lapply(1:10, sqrt)

library(parallel)
x.mc <- mclapply(1:10, sqrt)

all.equal(x.mc, x)
# [1] TRUE
```

The parallel Package: multicore

```
simplify2array(mclapply(1:10, function(i) Sys.getpid(),
    mc.cores=4))
# [1] 27452 27453 27454 27455 27452 27453 27454 27455
    27452 27453

simplify2array(mclapply(1:2, function(i) Sys.getpid(),
    mc.cores=4))
# [1] 27457 2745
```

The parallel Package: snow

- Uses sockets.
- (+) Works on all platforms.
- (-) More fiddley than `mclapply()`.
- (-) Not as efficient as forks.

The parallel Package: multicore

```
### Set up the worker processes
my.cl <- makeCluster(detectCores())
my.cl
# socket cluster with 4 nodes on host   localhost

parSapply(cl, 1:5, sqrt)

stopCluster(my.cl)
```

The parallel Package: Summary

All

- `detectCores()`
- `splitIndices()`

multicore

- `mclapply()`
- `mcmapply()`
- `mcparallel()`
- `mccollect()`
- and others...

snow

- `makeCluster()`
- `stopCluster()`
- `parLapply()`
- `parSapply()`
- and others...

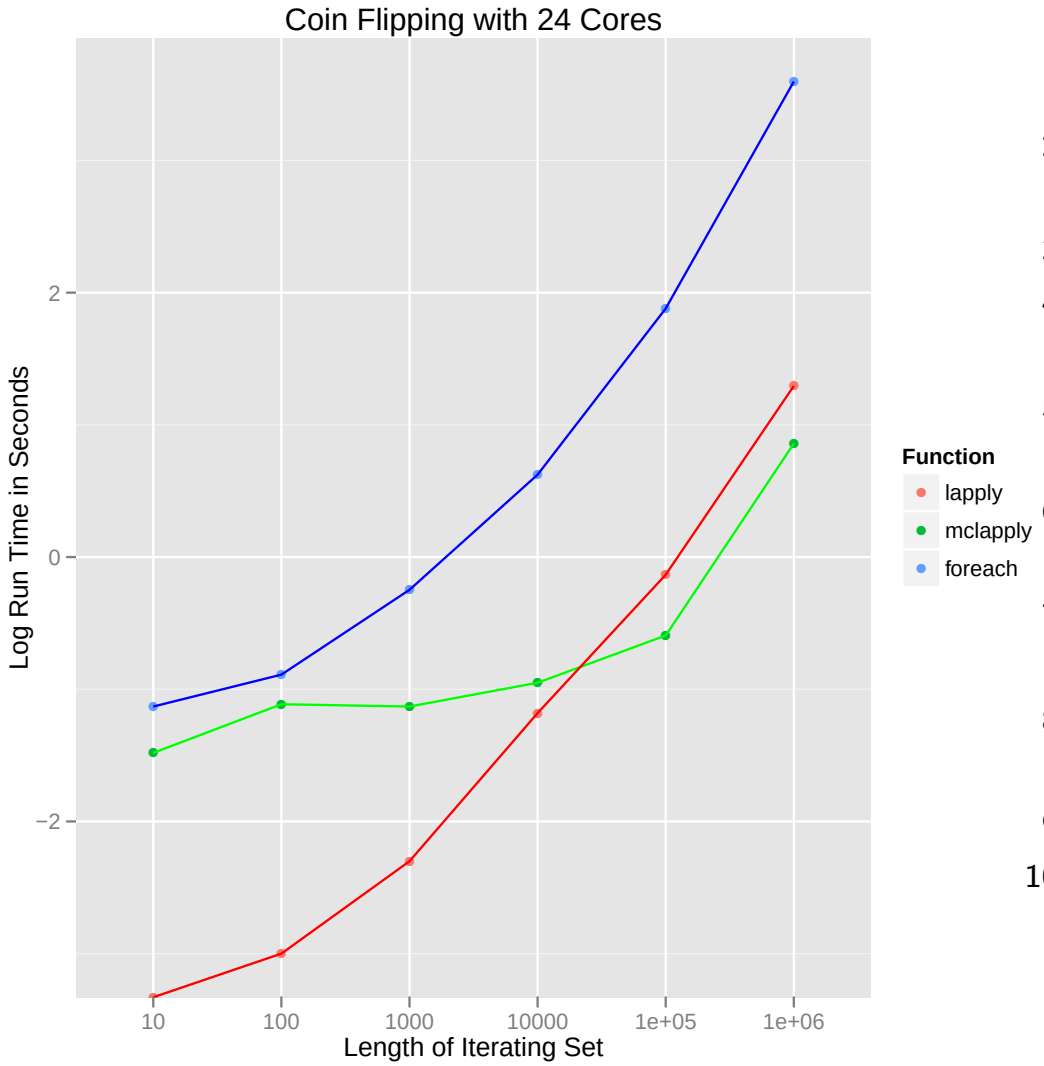
The foreach Package

- On Cran (Revolution Analytics).
- Main package is **foreach**, which is a single interface for a number of "backend" packages.
- Backends: **doMC**, **doMPI**, **doParallel**, **doRedis**, **doRNG**, **doSNOW**.

The foreach Package

- (+) Works on all platforms (with correct backend).
- (+) Can even work serial with minor notational change.
- (+) Write the code once, use whichever backend you prefer.
- (-) Really bizarre, non-R-ish synatx.
- (-) Efficiency issues ???

Efficiency Issues ???




```
### Bad performance
foreach(i=1:len)
    %dopar% tinyfun(i)

### Expected
    performance
foreach(i=1:ncores)
    %dopar% {
  out <-
      numeric(len/ncores)
  for (j in
      1:(len/ncores))
    out[i] <-
        tinyfun(j)
  out
}
```

The foreach Package: General Procedure

- Load **foreach** and your backend package.
- Register your backend.
- Call `foreach`

Using foreach: serial

```
library(foreach)

### Example 1
foreach(i=1:3) %do% sqrt(i)

### Example 2
n <- 50
reps <- 100

x <- foreach(i=1:reps) %do% {
  sum(rnorm(n, mean=i)) / (n*reps)
}
```

Using foreach: Parallel

```
library(foreach)
library(<mybackend>)

register<MyBackend>()

### Example 1
foreach(i=1:3) %dopar% sqrt(i)

### Example 2
n <- 50
reps <- 100

x <- foreach(i=1:reps) %dopar% {
  sum(rnorm(n, mean=i)) / (n*reps)
}
```

foreach backends

multicore

```
library(doParallel)
registerDoParallel(cores=ncores)
foreach(i=1:2) %dopar% Sys.getpid()
```

snow

```
library(doParallel)
cl <- makeCluster(ncores)
registerDoParallel(cl=cl)

foreach(i=1:2) %dopar% Sys.getpid()
stopCluster(cl)
```

foreach Summary

- Make sure to register your backend.
- Different backends may have different performance.
- Use `%dopar%` for parallel foreach.
- `%do%` and `%dopar%` *must* appear on the same line as the `foreach()` call.

6 Distributed Memory Parallelism with R

- Distributed Memory Parallelism
- Rmpi
- pbdMPI vs Rmpi
- Summary

Why Distribute?

- Nodes only hold so much ram.
- Commodity hardware: $\approx 32 - 64$ gib.
- With a few exceptions (**ff**, **bigmemory**), R does computations in memory.
- If your problem doesn't fit in the memory of one node...

Packages for Distributed Memory Parallelism in R

- **Rmpi**, and **snow** via **Rmpi**
- **RHIPE** and **RHadoop** ecosystem
- **pbdR** ecosystem (Monday)

Hasty Explanation of MPI

- We will return to this...
- MPI = Message Passing Interface
- Recall: Distributed machines can't directly manipulate memory of other nodes.
- Can *indirectly* manipulate them, however...
- Distinct nodes collaborate by passing messages over network.

Rmpi Hello World

```
mpi.spawn.Rslaves(nslaves=2)
#          2 slaves are spawned successfully. 0 failed.
# master (rank 0, comm 1) of size 3 is running on:
    wootabega
# slave1 (rank 1, comm 1) of size 3 is running on:
    wootabega
# slave2 (rank 2, comm 1) of size 3 is running on:
    wootabega

mpi.remote.exec(paste("I
    am",mpi.comm.rank(),"of",mpi.comm.size()))
# $slave1
# [1] "I am 1 of 3"
#
# $slave2
# [1] "I am 2 of 3"

mpi.exit()
```

Using Rmpi from snow

```
library(snow)
library(Rmpi)

cl <- makeCluster(2, type = "MPI")
clusterCall(cl, function() Sys.getpid())
clusterCall(cl, runif, 2)
stopCluster(cl)
mpi.quit()
```

Rmpi Resources

- **Rmpi** tutorial: http://math.acadiau.ca/ACMMaC/Rmpi/
- **Rmpi** manual: http://cran.r-project.org/web/packages/Rmpi/Rmpi.pdf

pbdMPI vs Rmpi

- **Rmpi** is interactive; **pbdMPI** is exclusively batch.
- **pbdMPI** is easier to install.
- **pbdMPI** has a simpler interface.
- **pbdMPI** integrates with other pbdR packages.

Example Syntax

Rmpi

```
# int
mpi.allreduce(x, type=1)
# double
mpi.allreduce(x, type=2)
```

pbdMPI

```
allreduce(x)
```

Types in R

```
> is.integer(1)
[1] FALSE
> is.integer(2)
[1] FALSE
> is.integer(1:2)
[1] TRUE
```

Summary

- Distributed parallelism is necessary when computations no longer fit in ram.
- Several options available; most go beyond the scope of this talk.
- More on pbdR Monday!

7 The pbdR Project

Recall: Parallel R Packages

Shared Memory

1. **foreach**
2. **parallel**
3. **snow**
4. **multicore**

Distributed

1. **Rmpi**
2. **RHIPE, RHadoop**
3. **pbdR**

(and others. . .)

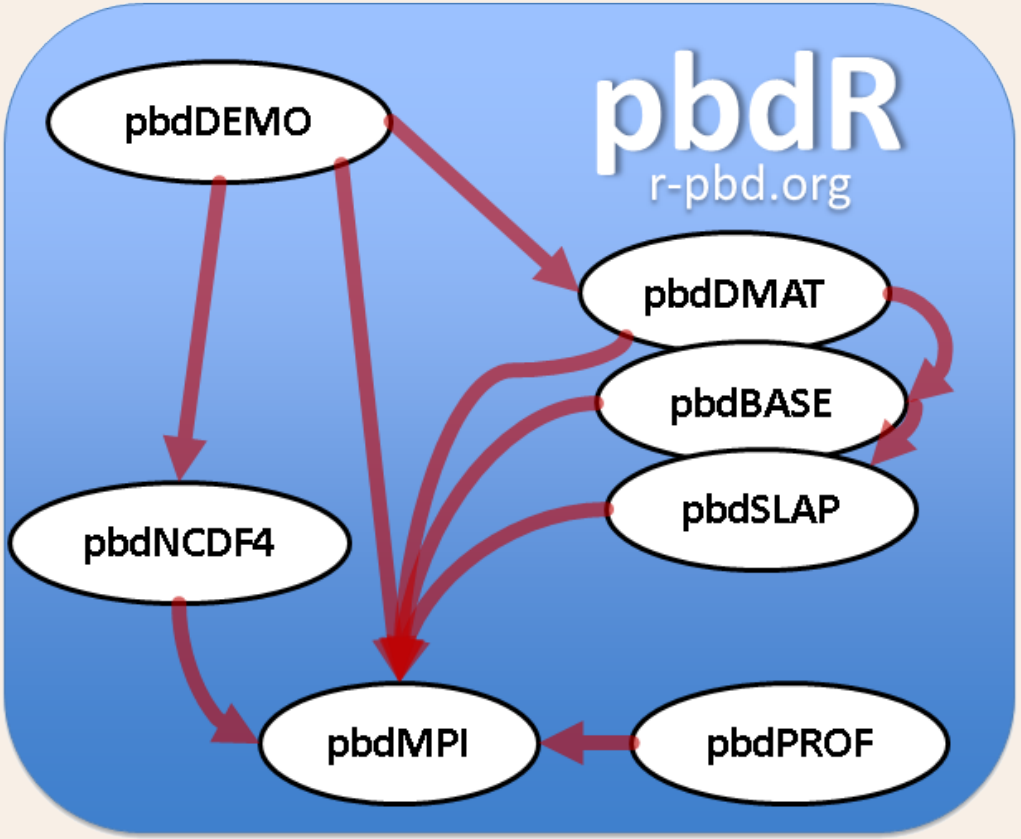
Programming with Big Data in R (pbdR)

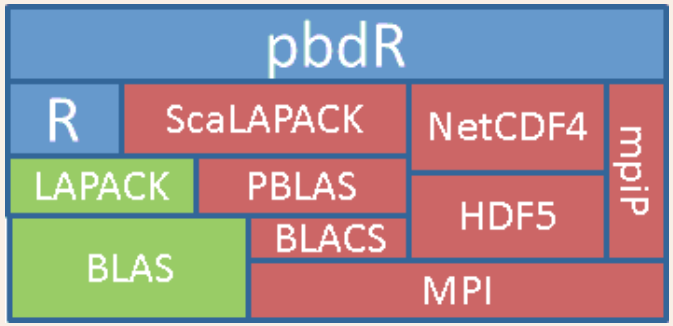
Striving for *Productivity, Portability, Performance*

- *Free*[a] R packages.
- Bridging high-performance compiled code with high-productivity of R
- Scalable, big data analytics.
- Offers implicit and explicit parallelism.
- Methods have syntax *identical* to R.

[a] MPL, BSD, and GPL licensed

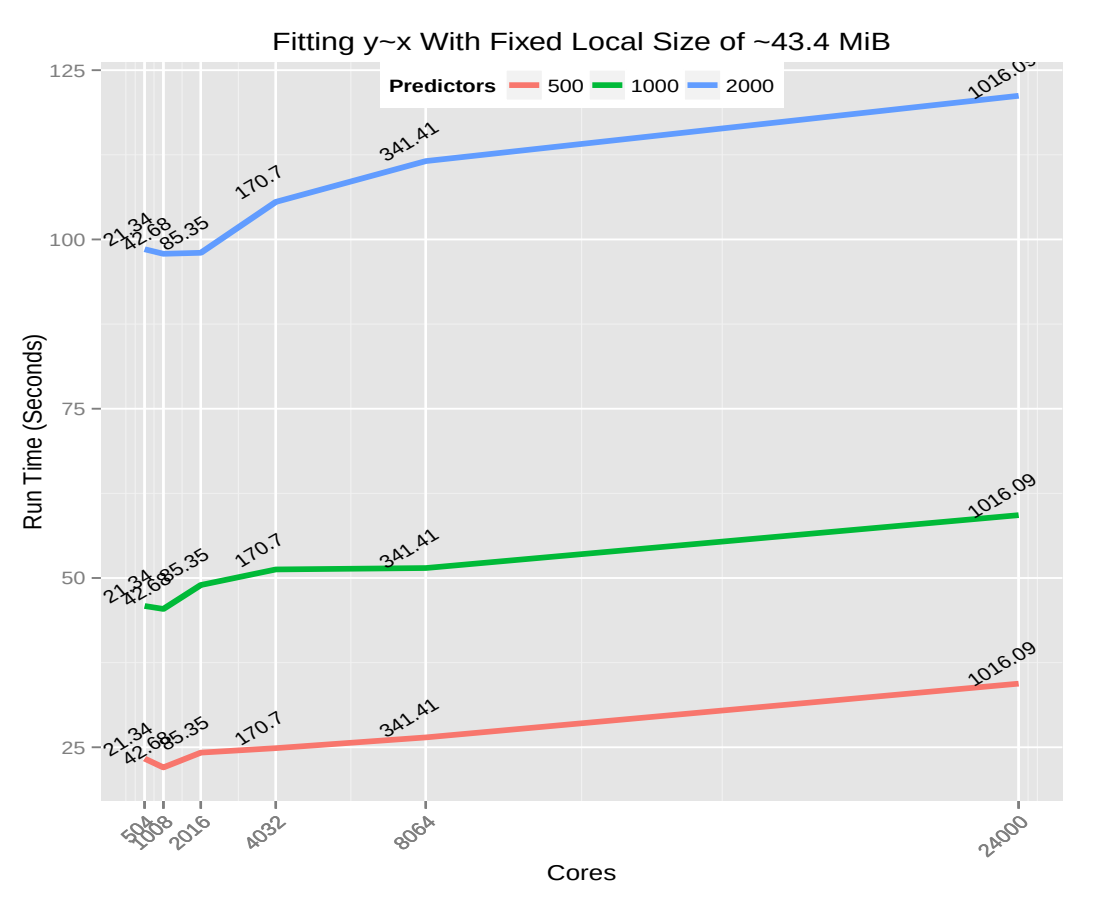
pbdR Packages





Distributed Matrices and Statistics with **pbdDMAT**

Least Squares Benchmark




```
x <- ddmatrix("rnorm", nrow=m, ncol=n)
y <- ddmatrix( "rnorm", nrow=m, ncol=1)
mdl <- lm.fit(x=x, y=y)
```

Profiling with **pbdPROF**

1. Rebuild **pbdR** packages

```
R CMD INSTALL
    pbdMPI_0.2-1.tar.gz \
    --configure-args= \
    "--enable-pbdPROF"
```

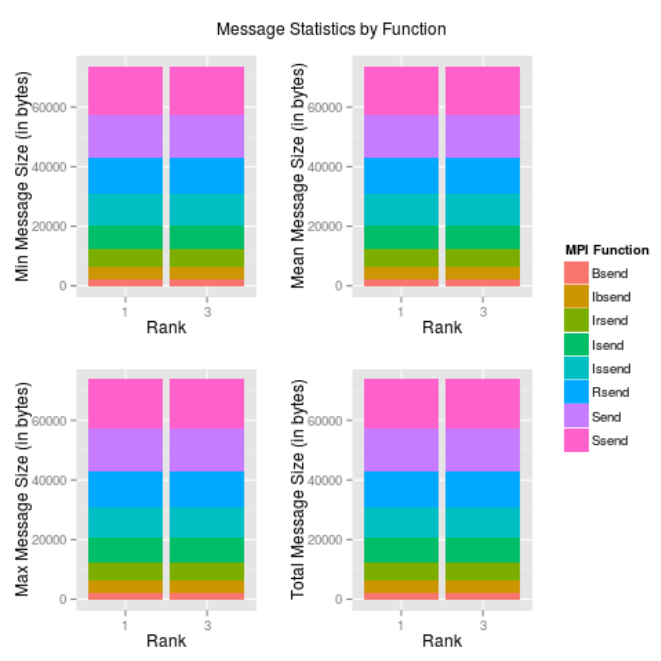
2. Run code

```
mpirun -np 64 Rscript
    my_script.R
```

3. Analyze results

```
library(pbdPROF)
prof <- read.prof(
    "profiler_output.mpiP")
plot(prof)
```

Publication-quality graphs



pbdR Basics

- **pbdR** programs are R programs
- Batch execution (non-interactive).
- Parallel code utilizes Single Program/Multiple Data (SPMD) style
- Emphasizes data parallelism.

Batch Execution

- Running a serial R program in batch:

```
Rscript my_script.r
```

or

```
R CMD BATCH my_script.r
```

- Running a parallel (with MPI) R program in batch:

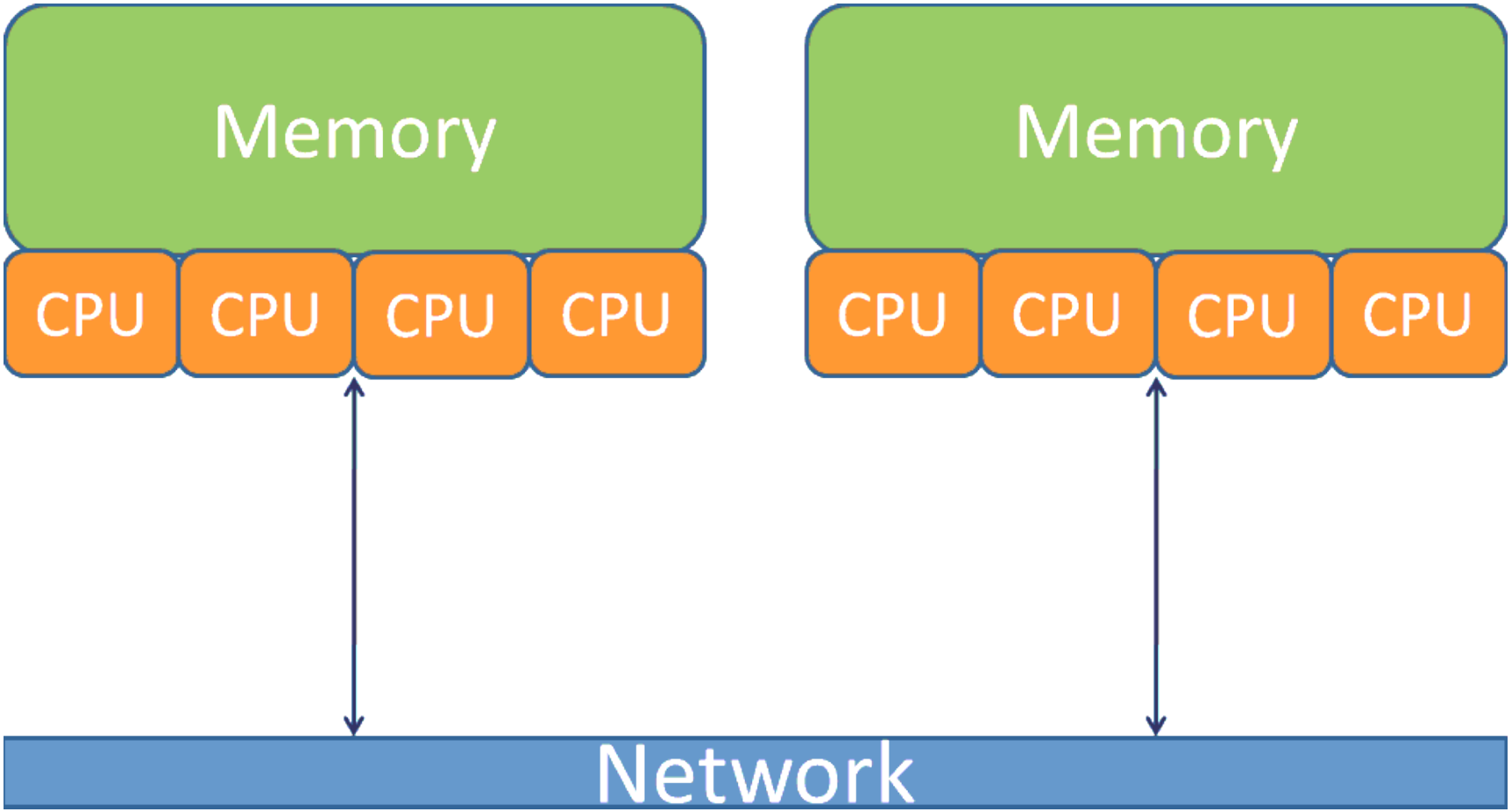
```
mpirun -np 2 Rscript my_par_script.r
```

8 A Hasty Introduction to MPI

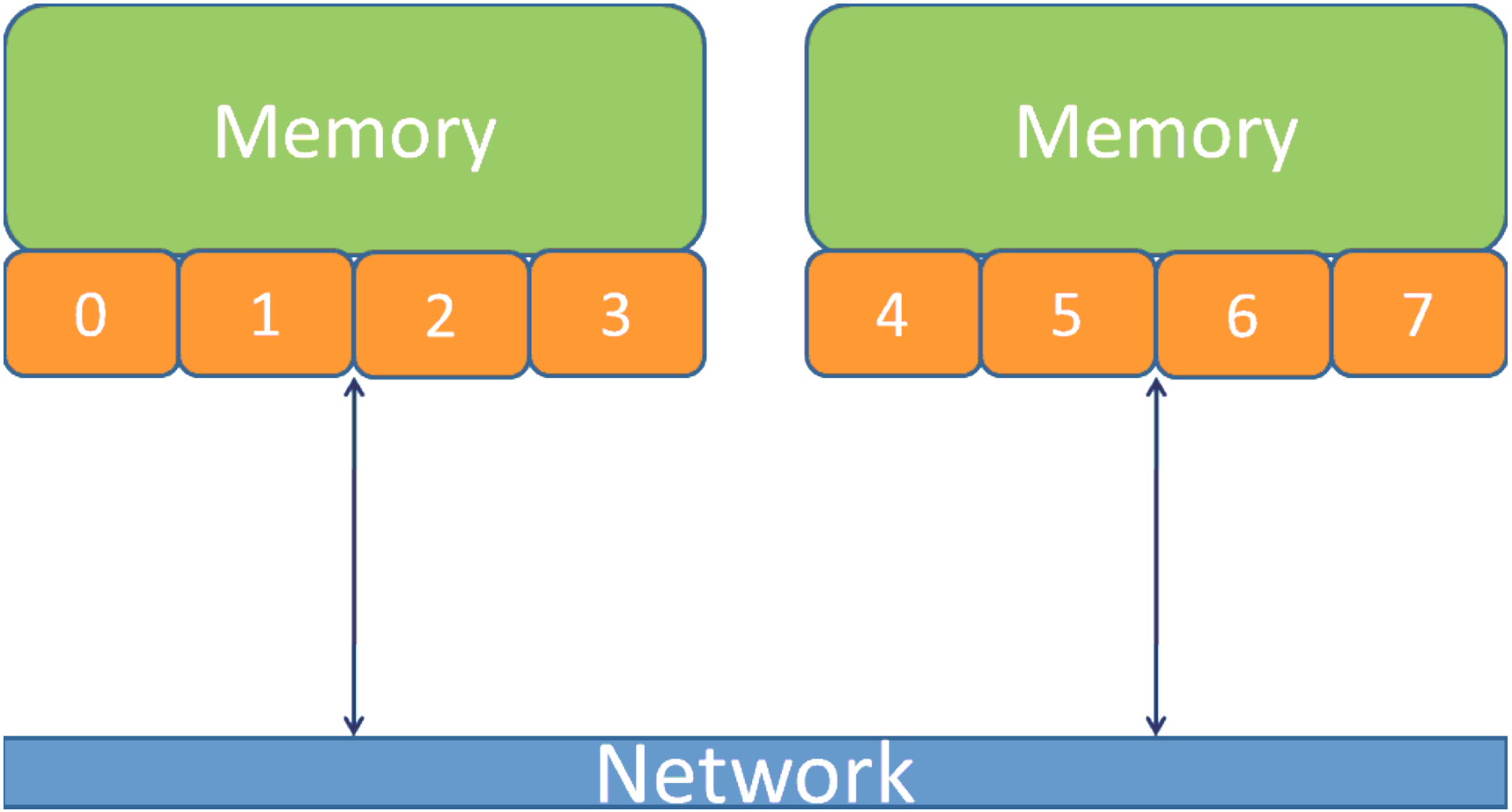
Message Passing Interface (MPI)

- *MPI*: Standard for managing communications (data and instructions) between different nodes/computers.
- *Implementations*: OpenMPI, MPICH2, Cray MPT, ...
- Enables parallelism (via communication) on distributed machines.
- *Communicator*: manages communications between processors.

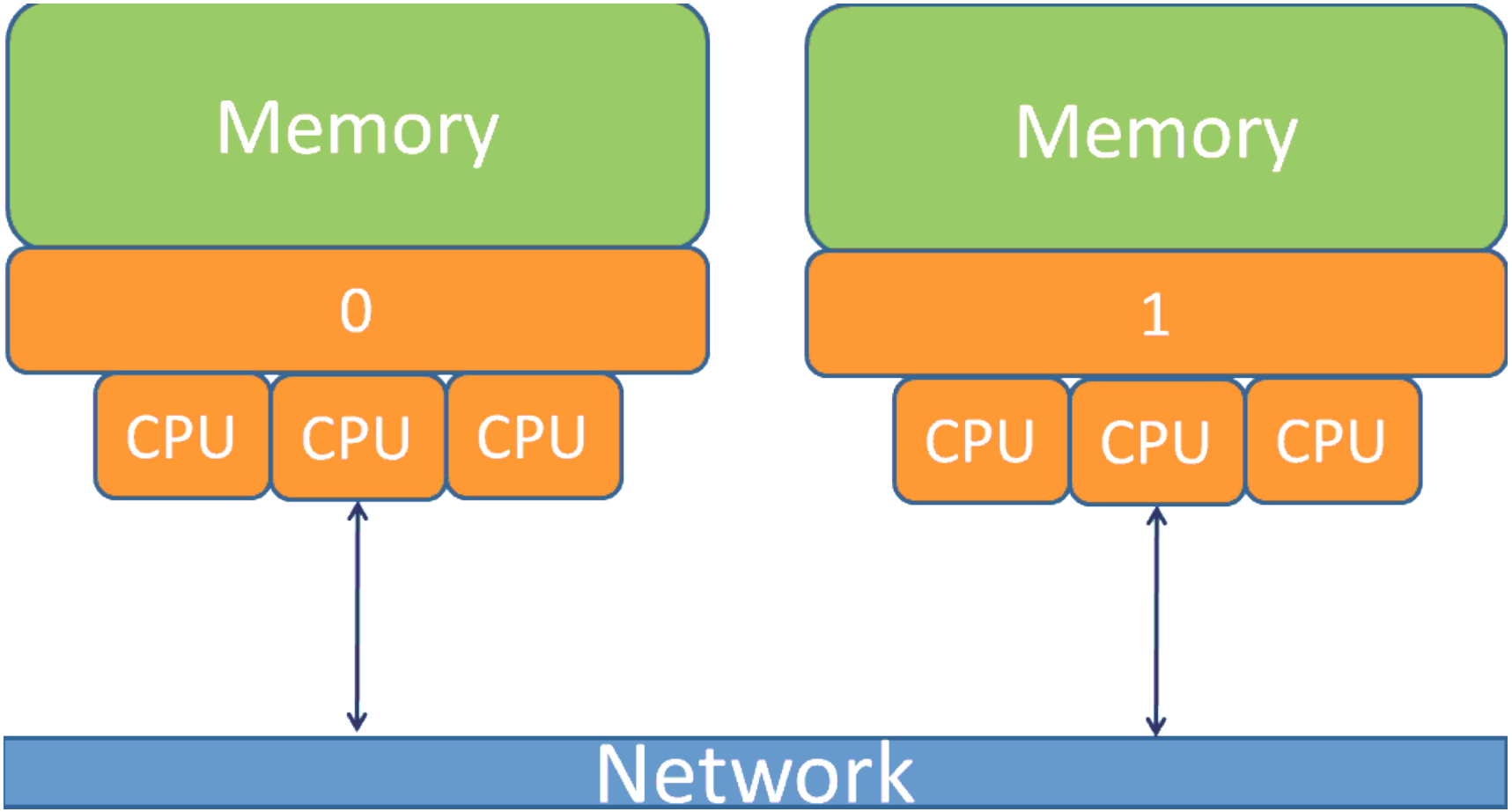
MPI Communicators



MPI Communicators



MPI Communicators



MPI Operations (1 of 2)

- **Managing a Communicator**: Create and destroy communicators.
  `init()` — initialize communicator
  `finalize()` — shut down communicator(s)

- **Rank query**: determine the processor's position in the communicator.
  `comm.rank()` — "who am I?"
  `comm.size()` — "how many of us are there?"

- **Printing**: Printing output from various ranks.
  `comm.print(x)`
  `comm.cat(x)`
  WARNING: only use these functions on *results*, never on yet-to-be-computed things.

Quick Example 1

Rank Query: 1_rank.r

```
library(pbdMPI, quietly = TRUE)
init()

my.rank <- comm.rank()
comm.print(my.rank, all.rank=TRUE)

finalize()
```

Execute this script via:

```
mpirun -np 2 Rscript 1_rank.r
```

Sample Output:

```
COMM.RANK = 0
[1] 0
COMM.RANK = 1
[1] 1
```

Quick Example 1: pbdinline

```
library(pbdinline)
body <- "
  my.rank <- comm.rank()
  comm.print(my.rank, all.rank=TRUE)
"

pbdRscript(body, cores=2)
```

Quick Example 2

Hello World: 2_hello.r

```
library(pbdMPI, quietly=TRUE)
init()

comm.print("Hello, world")

comm.print("Hello again", all.rank=TRUE, quietly=TRUE)

finalize()
```

Execute this script via:

```
mpirun -np 2 Rscript 2_hello.r
```

Sample Output:

```
COMM.RANK = 0
[1] "Hello, world"
[1] "Hello again"
[1] "Hello again"
```

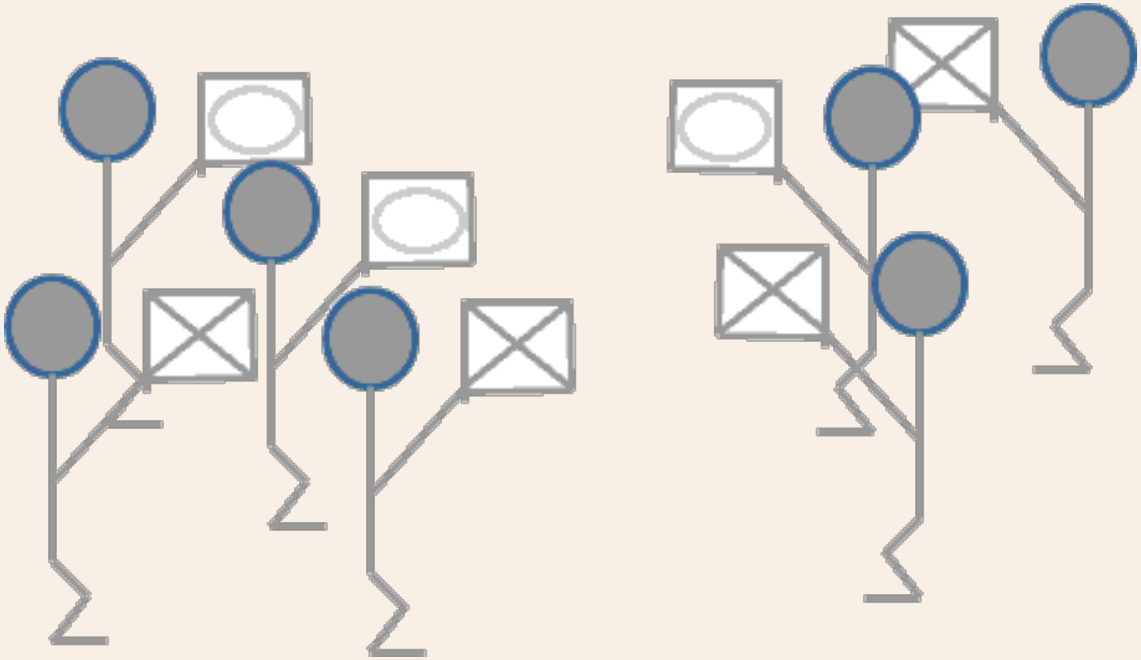
Quick Example 2: pbdinline

```
library(pbdinline)
body <- "
  comm.print("Hello, world")

  comm.print("Hello again", all.rank=TRUE, quietly=TRUE)
"

pbdRscript(body, cores=2)
```

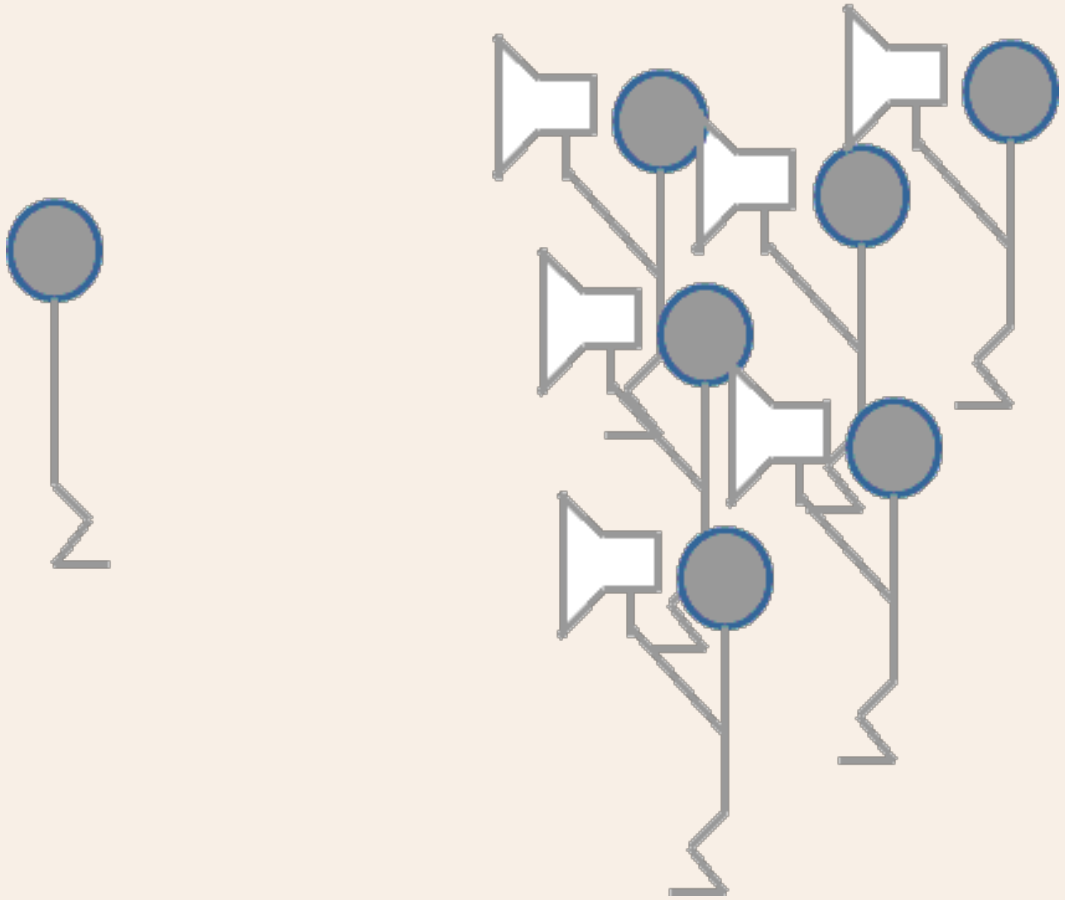
MPI Operations

1. Reduce
2. Gather
3. Broadcast
4. Barrier

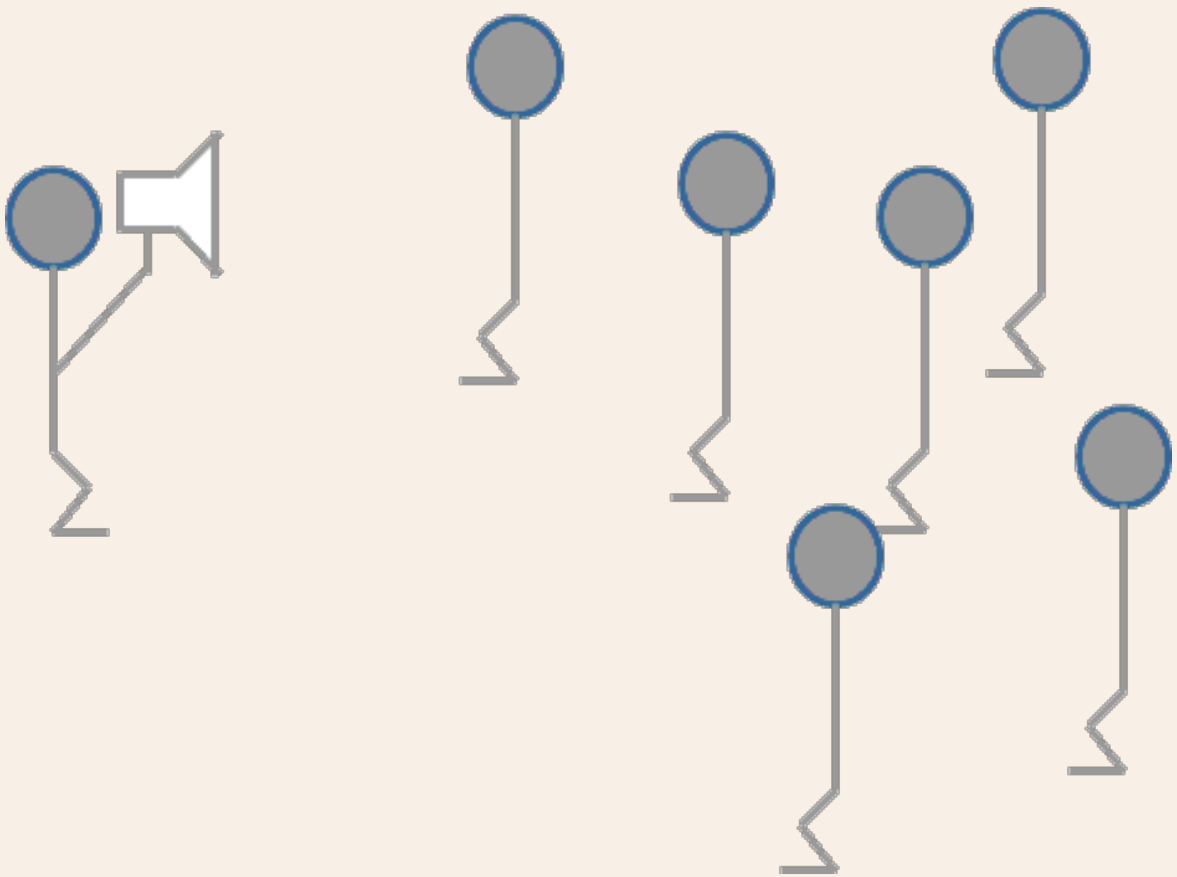
Reductions — Combine results into single result



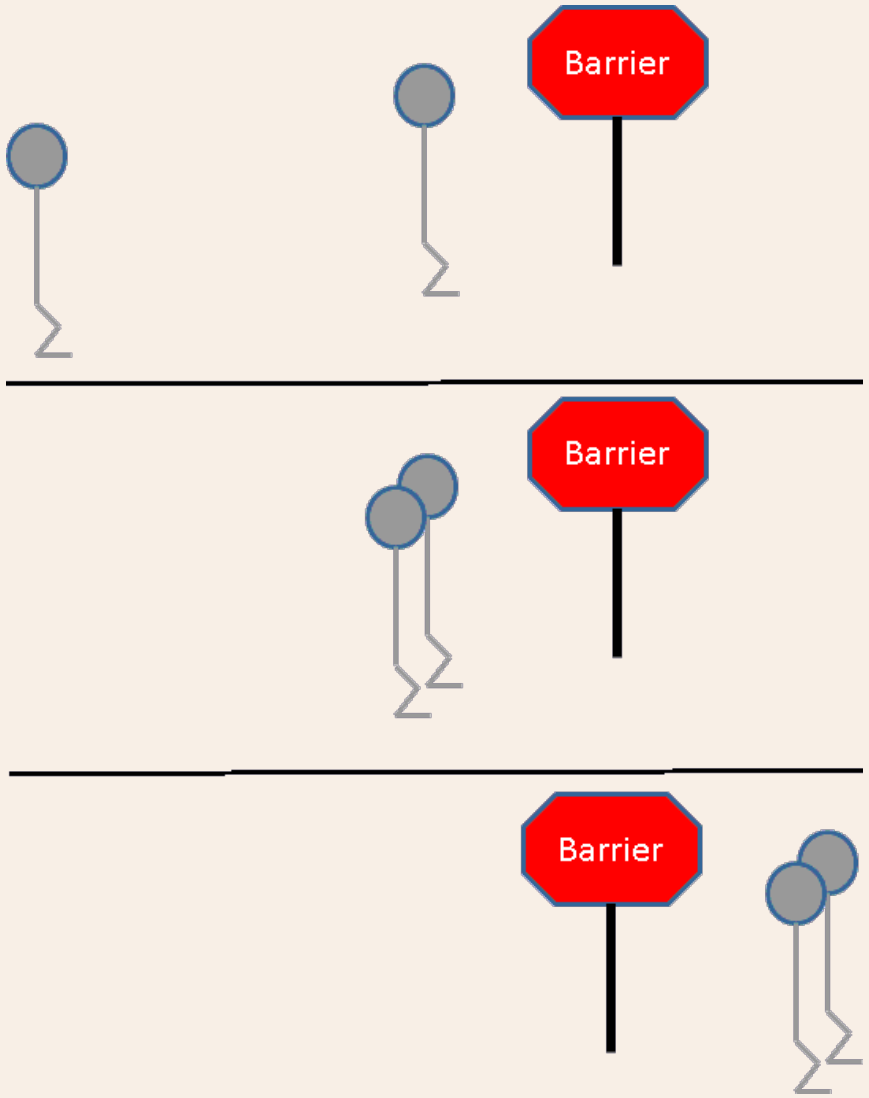
Gather — Many-to-one



Broadcast — One-to-many



Barrier — Synchronization



MPI Operations (2 of 2)

- **Reduction**: each processor has a number `x`; add all of them up, find the largest/smallest, ...
  `reduce(x, op='sum')` — reduce to one
  `allreduce(x, op='sum')` — reduce to all

- **Gather**: each processor has a number; create a new object on some processor containing all of those numbers.
  `gather(x)` — gather to one
  `allgather(x)` — gather to all

- **Broadcast**: one processor has a number `x` that every other processor should also have.
  `bcast(x)`

- **Barrier**: "computation wall"; no processor can proceed until *all* processors can proceed.
  `barrier()`

Quick Example 3

Reduce and Gather: 3_gt.r

```
library(pbdMPI, quietly=TRUE)
init()

comm.set.seed(1234, diff=TRUE)

n <- sample(1:10, size=1)

gt <- gather(n)
comm.print(unlist(gt))

sm <- allreduce(n, op='sum')
comm.print(sm, all.rank=T)

finalize()
```

Execute this script via:

```
mpirun -np 2 Rscript 3_gt.r
```

Sample Output:

```
COMM.RANK = 0
[1] 2 8
COMM.RANK = 0
[1] 10
COMM.RANK = 1
[1] 10
```

Quick Example 3: pbdinline

```
library(pbdinline)
body <- "
  comm.set.seed(1234, diff=TRUE)

  n <- sample(1:10, size=1)

  gt <- gather(n)
  comm.print(unlist(gt))

  sm <- allreduce(n, op='sum')
  comm.print(sm, all.rank=T)
"

pbdRscript(body, cores=2)
```

Quick Example 4

Broadcast: 4_bcast.r

```
library(pbdMPI, quietly=T)
init()

if (comm.rank()==0){
  x <- matrix(1:4, nrow=2)
} else {
  x <- NULL
}

y <- bcast(x, rank.source=0)

comm.print(y, rank=1)

finalize()
```

Execute this script via:

```
mpirun -np 2 Rscript 4_bcast.r
```

Sample Output:

```
COMM.RANK = 1
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Quick Example 4: pbdinline

```
library(pbdinline)
body <- "
  if (comm.rank()==0){
    x <- matrix(1:4, nrow=2)
  } else {
    x <- NULL
  }

  y <- bcast(x, rank.source=0)

  comm.print(y, rank=1)
"

pbdRscript(body, cores=2)
```

Other Helper Tools

pbdMPI Also contains useful tools for Manager/Worker and task parallelism codes:

- **Task Subsetting**: Distributing a list of jobs/tasks
  `get.jid(n)`
- ***ply**: Functions in the *ply family.
  `pbdApply(X, MARGIN, FUN, ...)` — analogue of `apply()`
  `pbdLapply(X, FUN, ...)` — analogue of `lapply()`
  `pbdSapply(X, FUN, ...)` — analogue of `sapply()`

9 Distributed Matrices

Distributed Matrices

Most problems in data science are matrix algebra problems, so:

$$\text{Distributed matrices} \implies \text{Handle Bigger data}$$

`ddmatrix`: 2-dimensional Block-Cyclic with 6 Processors

$$x = \left[\begin{array}{cc|cc|cc|cc|c} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & x_{17} & x_{18} & x_{19} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} & x_{27} & x_{28} & x_{29} \\ \hline x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & x_{37} & x_{38} & x_{39} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} & x_{47} & x_{48} & x_{49} \\ \hline x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} & x_{57} & x_{58} & x_{59} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} & x_{67} & x_{68} & x_{69} \\ \hline x_{71} & x_{72} & x_{73} & x_{74} & x_{75} & x_{76} & x_{77} & x_{78} & x_{79} \\ x_{81} & x_{82} & x_{83} & x_{84} & x_{85} & x_{86} & x_{87} & x_{88} & x_{89} \\ \hline x_{91} & x_{92} & x_{93} & x_{94} & x_{95} & x_{96} & x_{97} & x_{98} & x_{99} \end{array}\right]_{9\times 9}$$

$$\text{Processor grid} = \begin{vmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{vmatrix} = \begin{vmatrix} (0,0) & (0,1) & (0,2) \\ (1,0) & (1,1) & (1,2) \end{vmatrix}$$

Understanding `ddmatrix`: Local View

$$\left[\begin{array}{cc|cc} x_{11} & x_{12} & x_{17} & x_{18} \\ x_{21} & x_{22} & x_{27} & x_{28} \\ \hline x_{51} & x_{52} & x_{57} & x_{58} \\ x_{61} & x_{62} & x_{67} & x_{68} \\ \hline x_{91} & x_{92} & x_{97} & x_{98} \end{array}\right]_{5\times 4} \left[\begin{array}{cc|c} x_{13} & x_{14} & x_{19} \\ x_{23} & x_{24} & x_{29} \\ \hline x_{53} & x_{54} & x_{59} \\ x_{63} & x_{64} & x_{69} \\ \hline x_{93} & x_{94} & x_{99} \end{array}\right]_{5\times 3} \left[\begin{array}{cc} x_{15} & x_{16} \\ x_{25} & x_{26} \\ \hline x_{55} & x_{56} \\ x_{65} & x_{66} \\ \hline x_{95} & x_{96} \end{array}\right]_{5\times 2}$$

$$\left[\begin{array}{cc|cc} x_{31} & x_{32} & x_{37} & x_{38} \\ x_{41} & x_{42} & x_{47} & x_{48} \\ \hline x_{71} & x_{72} & x_{77} & x_{78} \\ x_{81} & x_{82} & x_{87} & x_{88} \end{array}\right]_{4\times 4} \left[\begin{array}{cc|c} x_{33} & x_{34} & x_{39} \\ x_{43} & x_{44} & x_{49} \\ \hline x_{73} & x_{74} & x_{79} \\ x_{83} & x_{84} & x_{89} \end{array}\right]_{4\times 3} \left[\begin{array}{cc} x_{35} & x_{36} \\ x_{45} & x_{46} \\ \hline x_{75} & x_{76} \\ x_{85} & x_{86} \end{array}\right]_{4\times 2}$$

$$\text{Processor grid} = \begin{vmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{vmatrix} = \begin{vmatrix} (0,0) & (0,1) & (0,2) \\ (1,0) & (1,1) & (1,2) \end{vmatrix}$$

Methods for class `ddmatrix`

pbdDMAT has over 100 methods with *identical* syntax to R:

- `` `[` ``, `rbind()`, `cbind()`, ...
- `lm.fit()`, `prcomp()`, `cov()`, ...
- `` `%*%` ``, `solve()`, `svd()`, `norm()`, ...
- `median()`, `mean()`, `rowSums()`, ...

Serial Code

```
cov(x)
```

Parallel Code

```
cov(x)
```

ddmatrix Syntax

```
cov.x <- cov(x)
pca <- prcomp(x)
x <- x[, -1]
col.sd <- apply(x, MARGIN=2, FUN=sd)
```

10 Matrix Exponentiation

Exponential Function

Recall from calculus that if $x \in \mathbb{R}$:

$$\exp(x) = \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Matrix Exponentiation

For a square matrix $X_{n\times n}$, we define the matrix exponential:

$$\text{expm}(X) = \frac{1}{1!}X + \frac{1}{2!}X^2 + \frac{1}{3!}X^3 + \ldots$$

when $X \neq \mathbf{0}_{n\times n}$; in this case, we take:

$$\text{expm}(\mathbf{0}_{n\times n}) = \mathbf{id}_{n\times n}$$

Computing the Matrix Exponential

- The naive implementation leads to a loss of accuracy for many matrices.
- This problem has been vigorously argued for 30+ years.
- Moler and Van Loan, *Nineteen Dubious Ways to Compute the Exponential of a Matrix*.

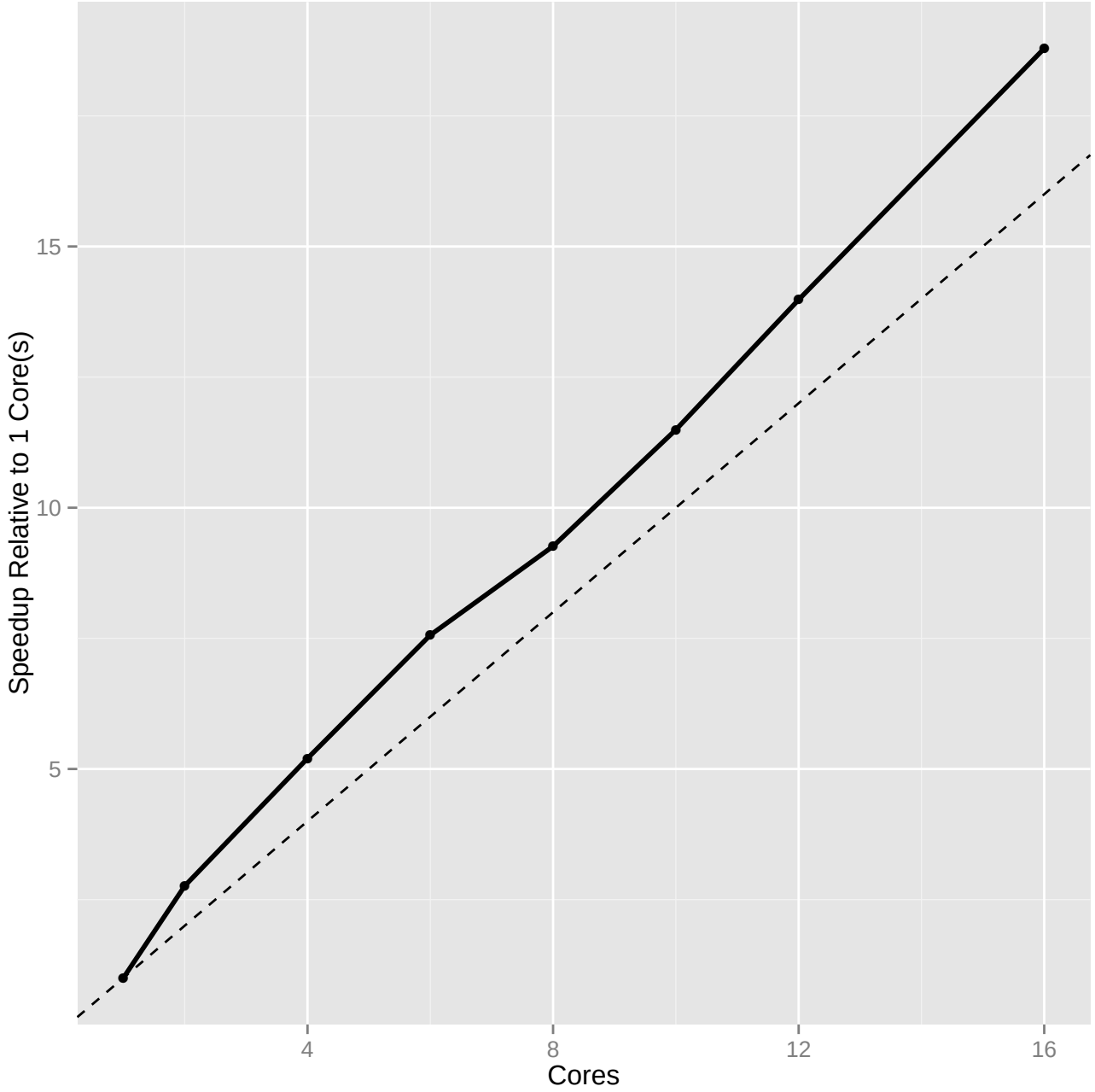
Scaling and Squaring

We use an improvement from Al-Mohy and Higham, *A New Scaling and Squaring Algorithm for the Matrix Exponential*.

```
expm <- function(x)
{
  n <- 2^j
  x <- x/n

  S <- matexp_pade(x)
  S <- matpow_by_squaring(S, n)

  return( S )
}
```

expm()

```
library(pbdDMAT)

x <- matrix(rnorm(25), 5, 5)
expm(x)

dx <- as.ddmatrix(x)
expm(dx)
```


Speedup Relative to 1 Core(s)
15
10
5
4
8
12
16
Cores

Thanks for coming!

Questions?