Introducing R: From Your Laptop to HPC and Big Data

Drew Schmidt

July 22, 2013

Affiliations and Support

The pbdR Core Team
http://r-pbd.org

Wei-Chen Chen[1], George Ostrouchov[1,2], Pragneshkumar Patel[2], Drew Schmidt[1]

Ostrouchov, Patel, and Schmidt were supported in part by the project "NICS Remote Data Analysis and Visualization Center" funded by the Office of Cyberinfrastructure of the U.S. National Science Foundation under Award No. ARRA-NSF-OCI-0906324 for NICS-RDAV center.

Chen and Ostrouchov were supported in part by the project "Visual Data Exploration and Analysis of Ultra-large Climate Data" funded by U.S. DOE Office of Science under Contract No. DE-AC05-00OR22725.

[1] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN
[2] Remote Data Analysis and Visualization Center, University of Tennessee, Knoxville, TN

About This Presentation

Downloads

This presentation and supplemental materials are available at:

http://r-pbd.org/xsede13

About This Presentation

Speaking Serial R with a Parallel Accent

The content of this presentation is based in part on the **pbdDEMO** vignette *Speaking Serial R with a Parallel Accent*

http://goo.gl/HZkRt

It contains more examples, and sometimes added detail.

About This Presentation

Installation Instructions

Installation instructions for setting up a pbdR environment are available:

http://r-pbd.org/install.html

This includes instructions for installing R, MPI, and pbdR.

About This Presentation

Conventions For Code Presentation

We will use two different forms of syntax highlighting. One for displaying results from an interactive R session:

```
R> "interactive"
[1] "interactive"
```

and one for presenting R scripts

```
"not interactive"
```

Contents

Contents

What is R?

- *lingua franca* for data analytics and statistical computing.
- Part programming language, part data analysis package.
- Dialect of S (Bell Labs).
- Syntax designed for data.

Who uses R?

Google, Pfizer, Merck, Bank of America, Shell[a], Oracle[b], Facebook, bing, Mozilla, okcupid[c], ebay[d], kickstarter[e], the New York Times[f]

---

[a] https://www.nytimes.com/2009/01/07/technology/business-computing/07program.html?_r=0

[b] http://www.oracle.com/us/corporate/features/features-oracle-r-enterprise-498732.html

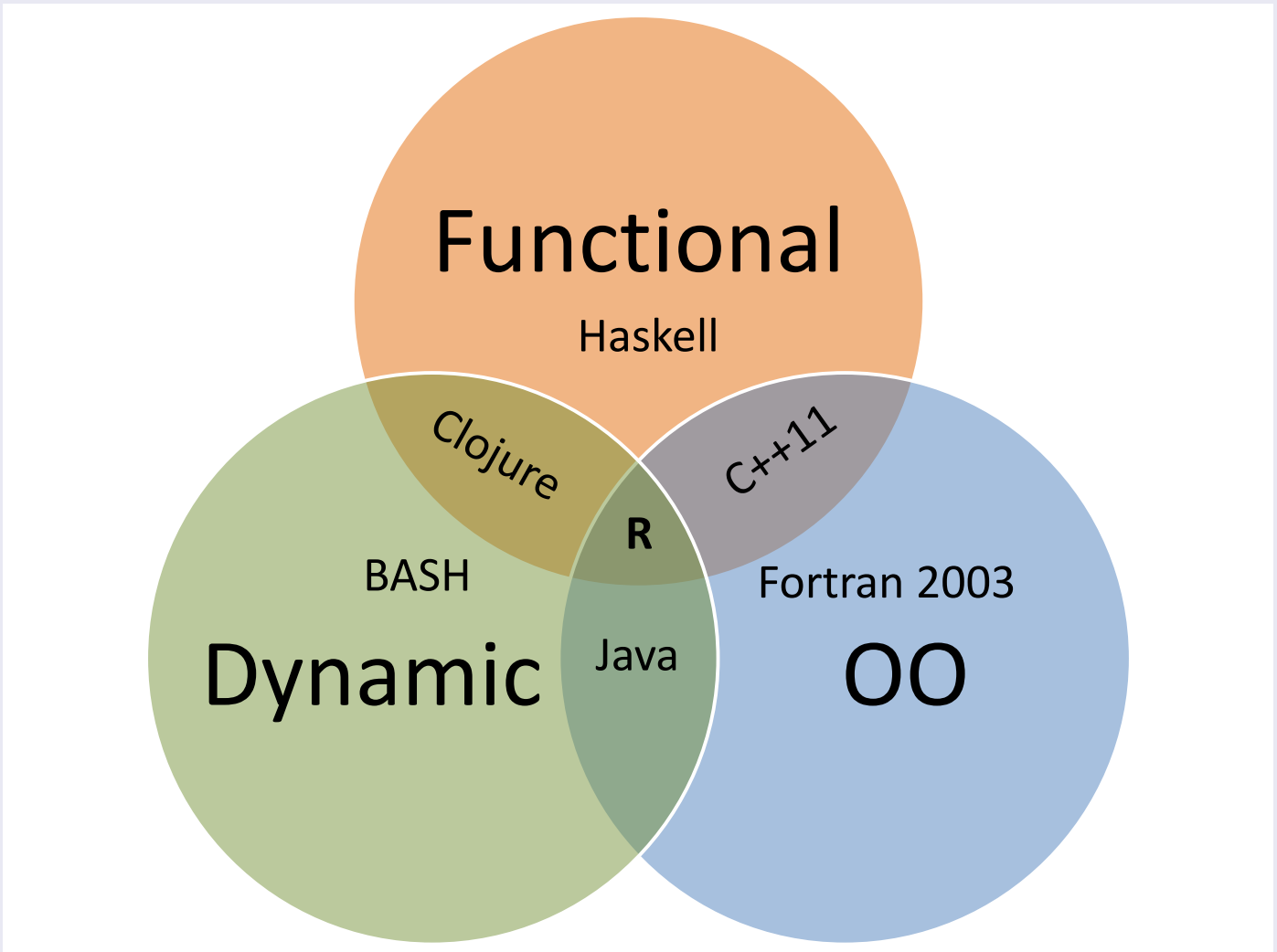
[c] http://www.revolutionanalytics.com/what-is-open-source-r/companies-using-r.php

[d] http://blog.revolutionanalytics.com/2012/09/using-r-in-production-industry-experts-share-their-experiences.html

[e] http://blog.revolutionanalytics.com/2012/09/kickstarter-facilitates-50m-in-indie-game-funding.html

[f] http://blog.revolutionanalytics.com/2012/05/nyt-charts-the-facebook-ipo-with-r.html

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.

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

```
1 R> ?set.seed
2 R> ?comm.set.seed
3 No documentation for 'comm.set.seed' in
    specified packages and libraries:
4 you could try '??comm.set.seed'
5 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

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

From Shell

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

Lists (1 of 1)

```
1  R> l <- list(a=1, b="a")
2  R> l
3  $a
4  [1] 1
5
6  $b
7  [1] "a"
8
9  R> l$a
10 [1] 1
11
12 R> list(x=list(a=1, b="a"), y=TRUE)
13 $x
14 $x$a
15 [1] 1
16
17 $x$b
18 [1] "a"
19
20
21 $y
22 [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

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

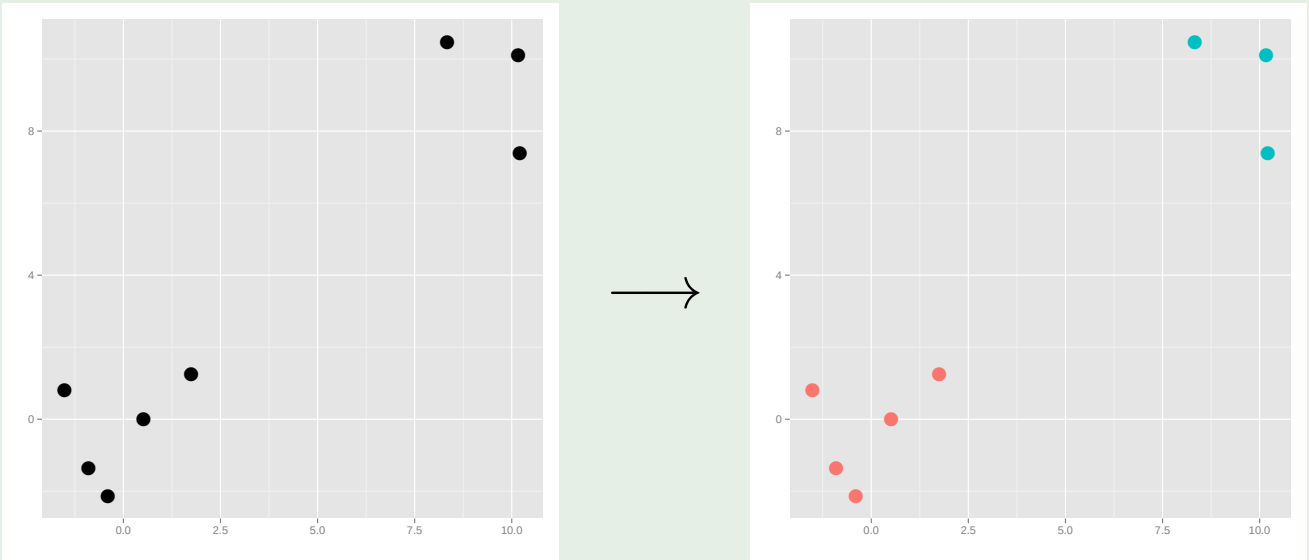
```
1  x <- matrix(rnorm(30), nrow=10)
2
3  # least recommended
4  cm <- colMeans(x)
5  crossprod(sweep(x, MARGIN=2, STATS=cm))
6
7  # less recommended
8  crossprod(scale(x, center=TRUE, scale=FALSE))
9
10 # recommended
11 cov(x)
```

Advanced Statistics (1 of 2): Principal Components

PCA = centering + scaling + rotation (via SVD)

```
1  R> x <- matrix(rnorm(30), nrow=10)
2
3  R> prcomp(x, retx=TRUE, scale=TRUE)
4  Standard deviations:
5  [1] 1.1203373 1.0617440 0.7858397
6
7  Rotation:
8                 PC1          PC2          PC3
9  [1,]  0.71697825 -0.3275365  0.6153552
10 [2,] -0.03382385  0.8653562  0.5000147
11 [3,]  0.69627447  0.3793133 -0.6093630
```

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




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

Contents

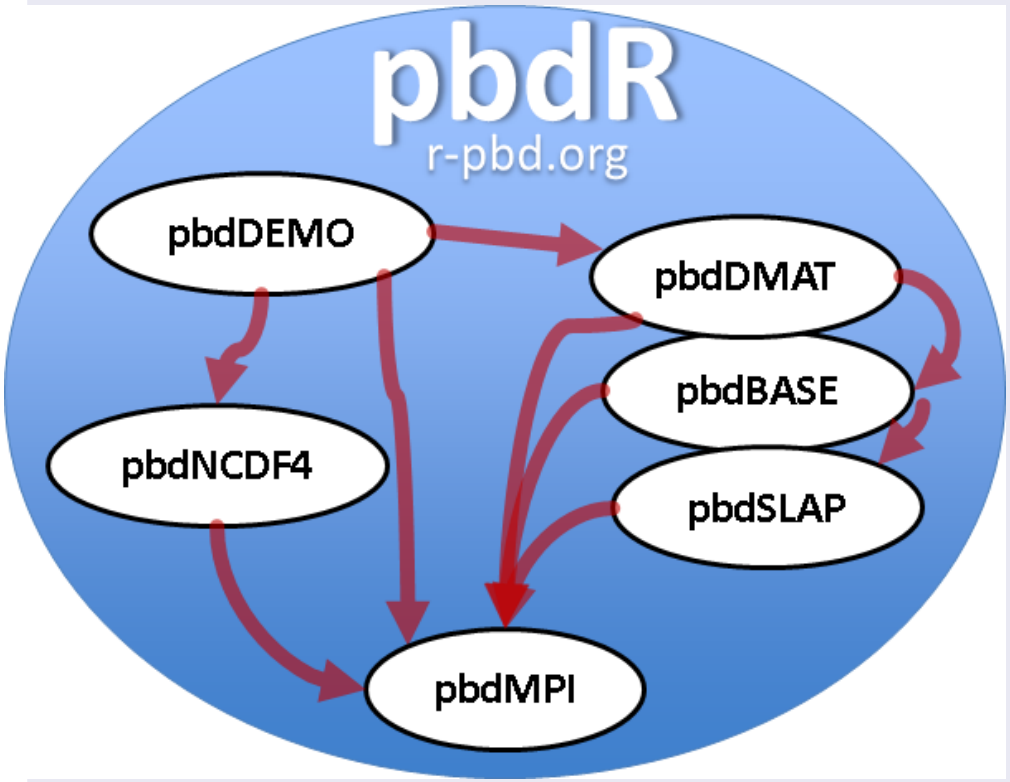
Programming with Big Data in R (pbdR)

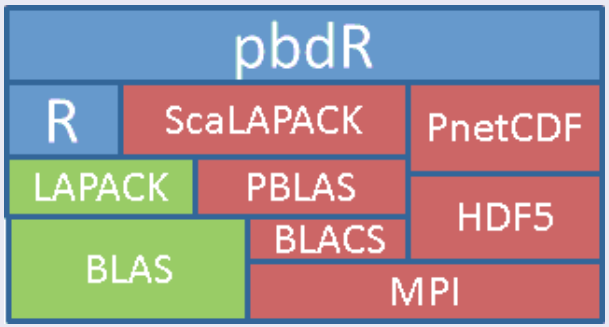
Striving for *Productivity, Portability, Performance*

- *Free*[a] R packages.
- Bridging high-performance C with high-productivity of R
- Scalable, big data analytics.
- Distributed data details implicitly managed.
- Methods have syntax *identical* to R.
- Powered by state of the art numerical libraries (MPI, ScaLAPACK, . . .)

[a] MPL, BSD, and GPL licensed

pbdR Packages





pbdR on XSEDE Resources

pbdR is currently actively maintained on

- **Nautilus**
- **Kraken**

If you are interested in maintaining pbdR, contact us at RBigData@gmail.com

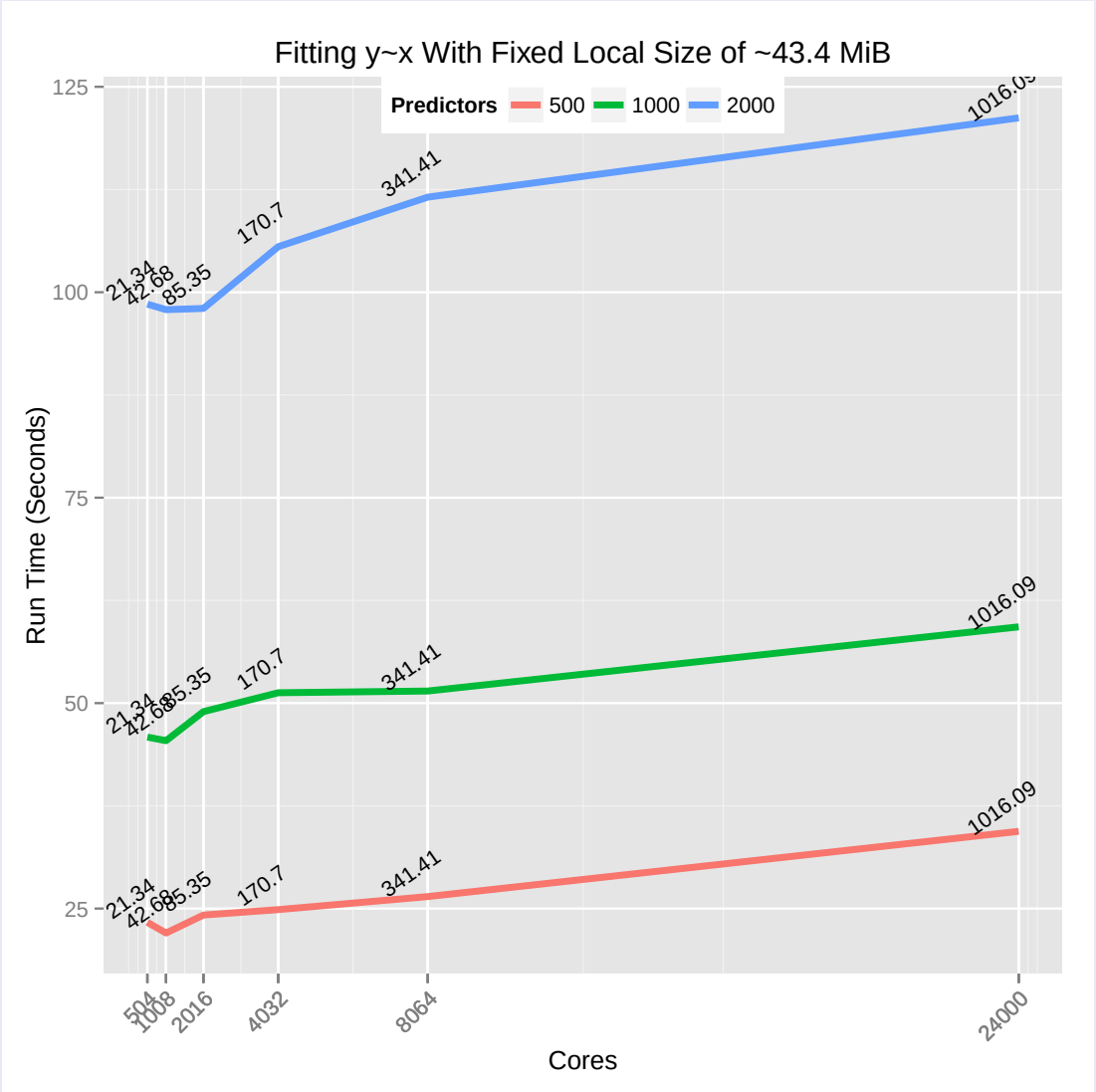
Example Syntax

```
1 x <- x[-1, 2:5]
2 x <- log(abs(x) + 1)
3 xtx <- t(x) %*% x
4 ans <- svd(solve(xtx))
```

Look familiar?

The above runs on 1 core with R or 10,000 cores with pbdR

Least Squares Benchmark



pbdR Paradigms

Programs that use pbdR utilize:

- Batch execution
- Single Program/Multiple Data (SPMD) style

And generally utilize:

- Data Parallelism

Batch Execution

- Non-interactive
- Use

```
1 Rscript my_script.r
```

or

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

- In parallel:

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

Single Program/Multiple Data (SPMD)

- Difficult to describe, easy to do...
- Only one program is written, executed in batch on all processors.
- Different processors are autonomous; there is no manager.
- The dominant programming model for large machines.

Contents

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 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

```
1 library(pbdMPI, quiet = TRUE)
2 init()
3
4 my.rank <- comm.rank()
5 comm.print(my.rank, all.rank=TRUE)
6
7 finalize()
```

Execute this script via:

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

Sample Output:

```
1 COMM.RANK = 0
2 [1] 0
3 COMM.RANK = 1
4 [1] 1
```

Quick Example 2

Hello World: 2_hello.r

```
1 library(pbdMPI, quiet=TRUE)
2 init()
3
4 comm.print("Hello, world")
5
6 comm.print("Hello again", all.rank=TRUE, quiet=TRUE)
7
8 finalize()
```

Execute this script via:

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

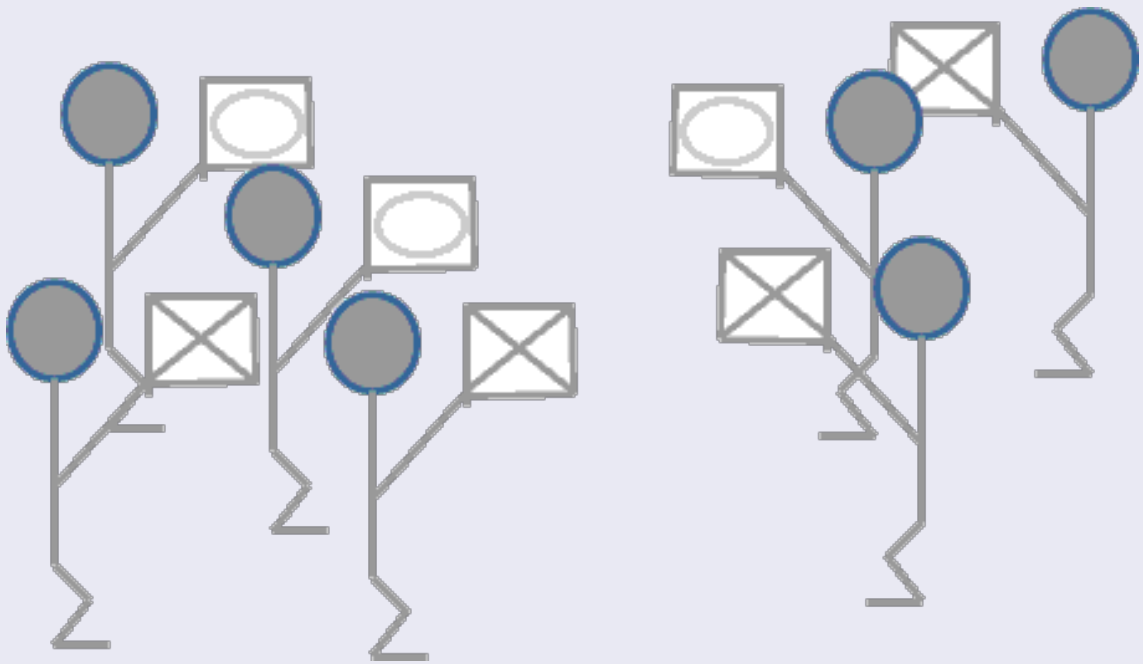
Sample Output:

```
1 COMM.RANK = 0
2 [1] "Hello, world"
3 [1] "Hello again"
4 [1] "Hello again"
```

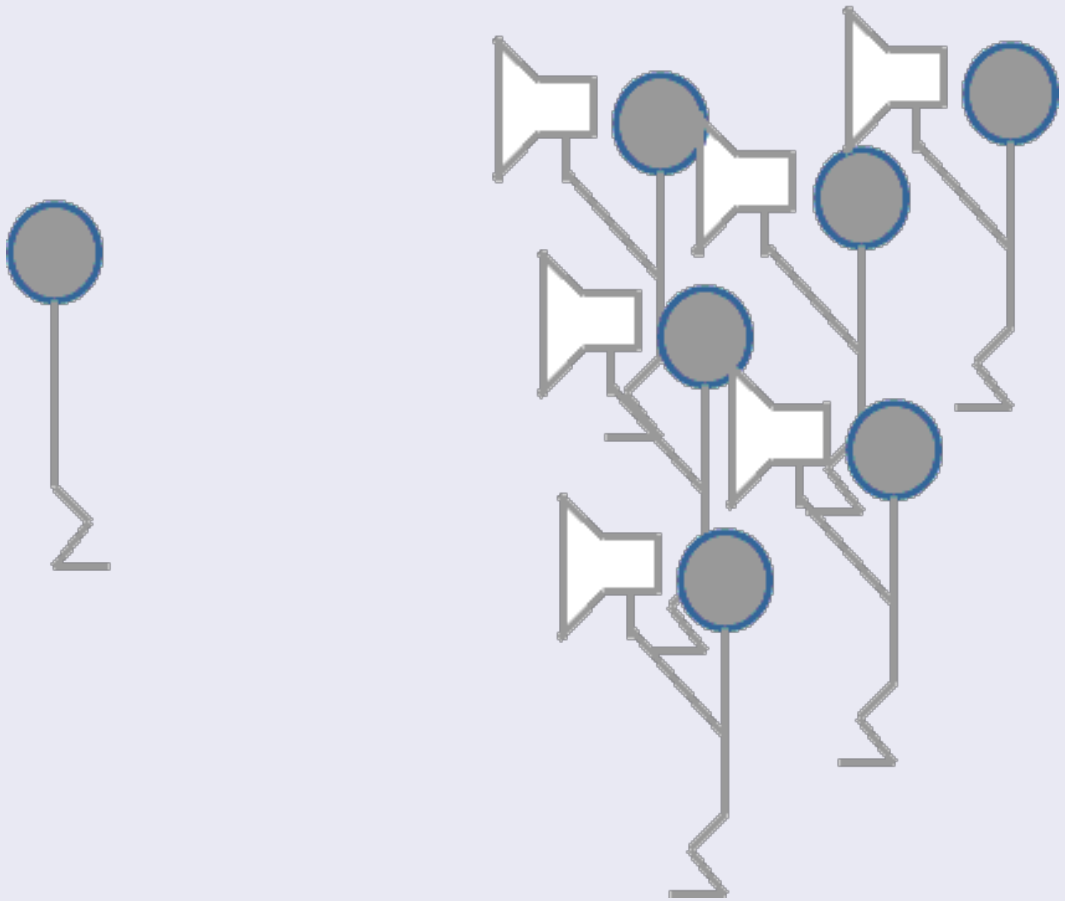
MPI Operations

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

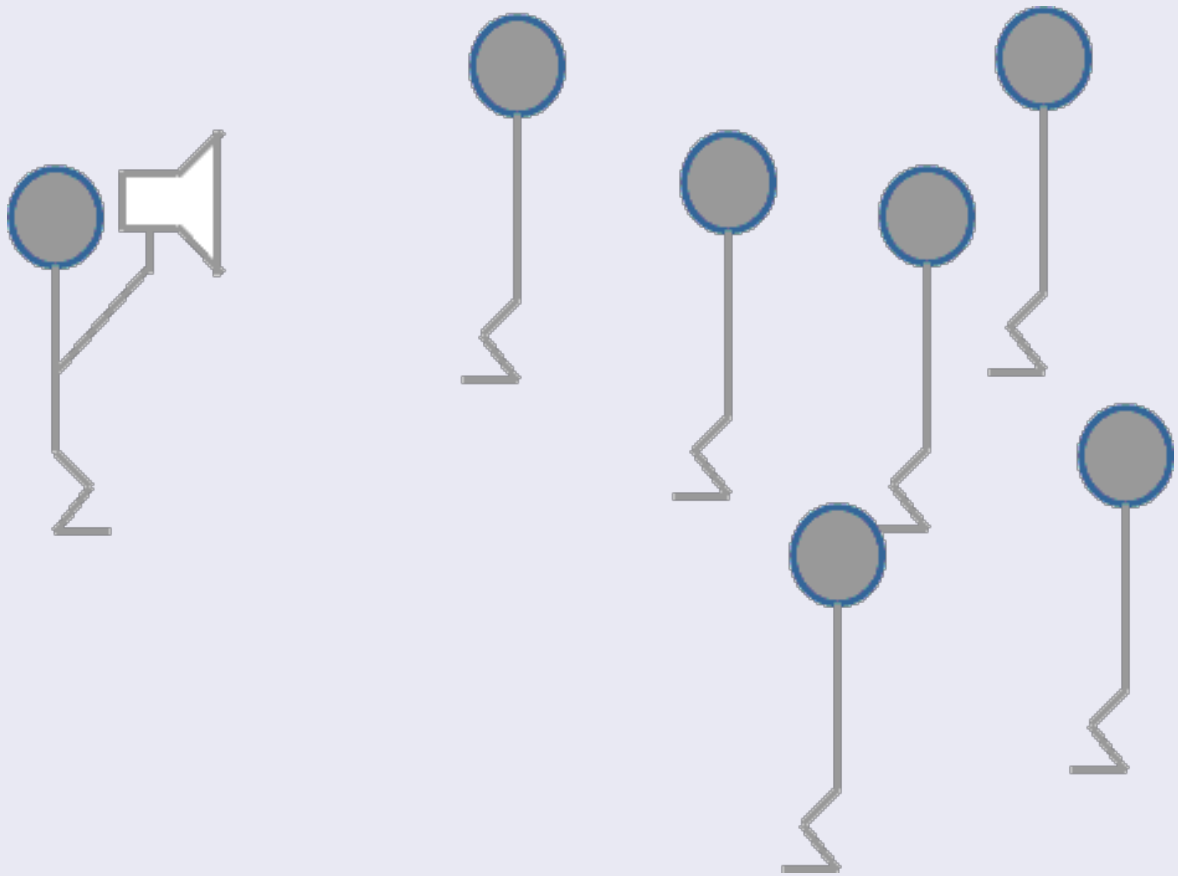
Reductions — Combine results into single result



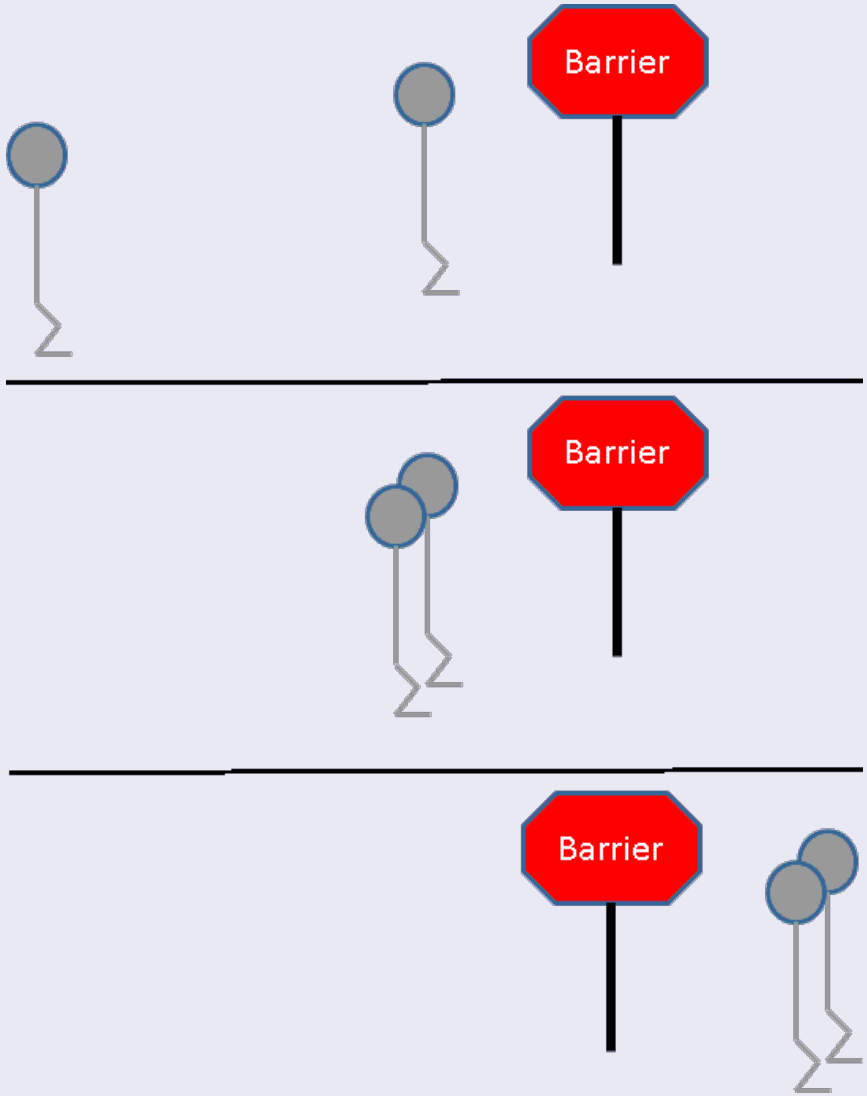
Gather — Many-to-one



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()`

MPI Package Controls

The `.SPMD.CT` object allows for setting different package options with **pbdMPI**. See the entry *SPMD Control* of the **pbdMPI** manual for information about the `.SPMD.CT` object:

http://cran.r-project.org/web/packages/pbdMPI/pbdMPI.pdf

Random Seeds

pbdMPI offers a simple interface for managing random seeds:

- `comm.set.seed(diff=TRUE)` — Independent streams via the **rlecuyer** package.
- `comm.set.seed(seed=1234, diff=FALSE)` — All processors use the same seed `seed=1234`
- `comm.set.seed(diff=FALSE)` — All processors use the same seed, determined by processor 0 (using the system clock and PID of processor 0).

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()`

Quick Comments for Using pbdMPI

1. Start by loading the package:

```
1 library(pbdMPI, quiet = TRUE)
```

2. Always initialize before starting and finalize when finished:

```
1 init()
2
3 # ...
4
5 finalize()
```

Basic MPI Exercises

1. Experiment with Quick Examples 1 through 6, running them on 2, 4, and 8 processors.

Contents

Distributing Data

Problem: How to distribute the data

$$x = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \\ x_{4,1} & x_{4,2} & x_{4,3} \\ x_{5,1} & x_{5,2} & x_{5,3} \\ x_{6,1} & x_{6,2} & x_{6,3} \\ x_{7,1} & x_{7,2} & x_{7,3} \\ x_{8,1} & x_{8,2} & x_{8,3} \\ x_{9,1} & x_{9,2} & x_{9,3} \\ x_{10,1} & x_{10,2} & x_{10,3} \end{bmatrix}_{10\times 3}$$

?

Distributing a Matrix Across 4 Processors: Block Distribution

Data

$$x = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \\ \hline x_{4,1} & x_{4,2} & x_{4,3} \\ x_{5,1} & x_{5,2} & x_{5,3} \\ x_{6,1} & x_{6,2} & x_{6,3} \\ \hline x_{7,1} & x_{7,2} & x_{7,3} \\ x_{8,1} & x_{8,2} & x_{8,3} \\ x_{9,1} & x_{9,2} & x_{9,3} \\ \hline x_{10,1} & x_{10,2} & x_{10,3} \end{bmatrix}_{10\times 3}$$

Processors

0
1
2
3

Distributing a Matrix Across 4 Processors: Local Load Balance

Data

$$x = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \\ \hline x_{4,1} & x_{4,2} & x_{4,3} \\ x_{5,1} & x_{5,2} & x_{5,3} \\ x_{6,1} & x_{6,2} & x_{6,3} \\ \hline x_{7,1} & x_{7,2} & x_{7,3} \\ x_{8,1} & x_{8,2} & x_{8,3} \\ \hline x_{9,1} & x_{9,2} & x_{9,3} \\ x_{10,1} & x_{10,2} & x_{10,3} \end{bmatrix}_{10\times 3}$$

Processors

0
1
2
3

The GBD Data Structure

Throughout the examples, we will make use of the Generalized Block Distribution, or GBD distributed matrix structure.

1. GBD is *distributed*. No processor owns all the data.
2. GBD is *non-overlapping*. Rows uniquely assigned to processors.
3. GBD is *row-contiguous*. If a processor owns one element of a row, it owns the entire row.
4. GBD is globally *row-major*, locally *column-major*.
5. GBD is often *locally balanced*, where each processor owns (almost) the same amount of data. But this is not required.

$$\left[\begin{array}{ccc} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \\ \hline x_{4,1} & x_{4,2} & x_{4,3} \\ x_{5,1} & x_{5,2} & x_{5,3} \\ x_{6,1} & x_{6,2} & x_{6,3} \\ \hline x_{7,1} & x_{7,2} & x_{7,3} \\ x_{8,1} & x_{8,2} & x_{8,3} \\ \hline x_{9,1} & x_{9,2} & x_{9,3} \\ x_{10,1} & x_{10,2} & x_{10,3} \end{array}\right]$$

6. The last row of the local storage of a processor is adjacent (by global row) to the first row of the local storage of next processor (by communicator number) that owns data.
7. GBD is (relatively) easy to understand, but can lead to bottlenecks if you have many more columns than rows.

Understanding GBD: Global Matrix

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

Processors = 0 1 2 3 4 5

Understanding GBD: Load Balanced GBD

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

Processors = 0 1 2 3 4 5

Understanding GBD: Local View

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & x_{17} & x_{18} & x_{19} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} & x_{27} & x_{28} & x_{29} \end{bmatrix}_{2\times 9}$$

$$\begin{bmatrix} x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & x_{37} & x_{38} & x_{39} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} & x_{47} & x_{48} & x_{49} \end{bmatrix}_{2\times 9}$$

$$\begin{bmatrix} x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} & x_{57} & x_{58} & x_{59} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} & x_{67} & x_{68} & x_{69} \end{bmatrix}_{2\times 9}$$

$$\begin{bmatrix} x_{71} & x_{72} & x_{73} & x_{74} & x_{75} & x_{76} & x_{77} & x_{78} & x_{79} \end{bmatrix}_{1\times 9}$$

$$\begin{bmatrix} x_{81} & x_{82} & x_{83} & x_{84} & x_{85} & x_{86} & x_{87} & x_{88} & x_{89} \end{bmatrix}_{1\times 9}$$

$$\begin{bmatrix} x_{91} & x_{92} & x_{93} & x_{94} & x_{95} & x_{96} & x_{97} & x_{98} & x_{99} \end{bmatrix}_{1\times 9}$$

Processors = 0 1 2 3 4 5

Understanding GBD: Non-Balanced GBD

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

Processors = 0 1 2 3 4 5

Understanding GBD: Local View

$$
\begin{bmatrix} \end{bmatrix}_{0\times 9}
\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & x_{17} & x_{18} & x_{19} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} & x_{27} & x_{28} & x_{29} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & x_{37} & x_{38} & x_{39} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} & x_{47} & x_{48} & x_{49} \end{bmatrix}_{4\times 9}
\begin{bmatrix} x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} & x_{57} & x_{58} & x_{59} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} & x_{67} & x_{68} & x_{69} \end{bmatrix}_{2\times 9}
\begin{bmatrix} x_{71} & x_{72} & x_{73} & x_{74} & x_{75} & x_{76} & x_{77} & x_{78} & x_{79} \end{bmatrix}_{1\times 9}
\begin{bmatrix} \end{bmatrix}_{0\times 9}
\begin{bmatrix} x_{81} & x_{82} & x_{83} & x_{84} & x_{85} & x_{86} & x_{87} & x_{88} & x_{89} \\ x_{91} & x_{92} & x_{93} & x_{94} & x_{95} & x_{96} & x_{97} & x_{98} & x_{99} \end{bmatrix}_{2\times 9}
$$

Processors = 0 1 2 3 4 5

Quick Comment for GBD

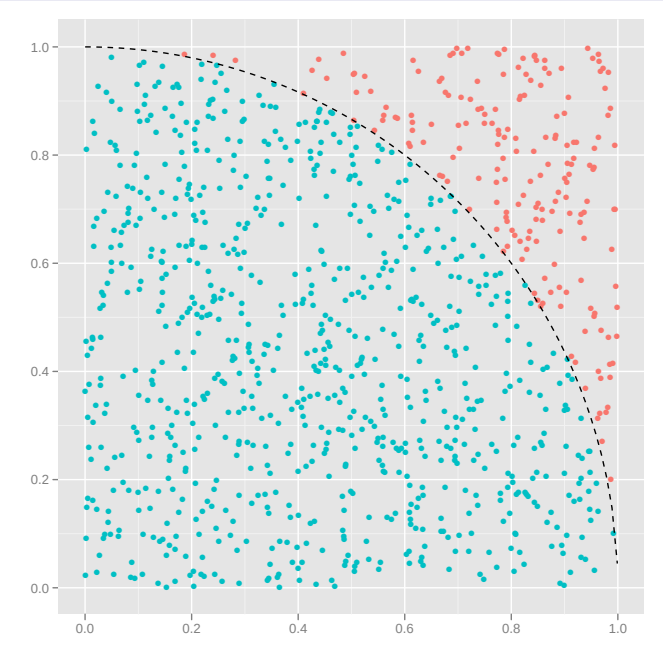
Local pieces of GBD distributed objects will be given the suffix .gbd to visually help distinguish them from global objects. This suffix carries no semantic meaning.

Contents

Example 1: Monte Carlo Simulation

Sample N uniform observations (x_i, y_i) in the unit square $[0,1] \times [0,1]$. Then

$$\pi \approx 4\left(\frac{\#\ Inside\ Circle}{\#\ Total}\right) = 4\left(\frac{\#\ \text{Blue}}{\#\ \text{Blue} + \#\ \text{Red}}\right)$$



Example 1: Monte Carlo Simulation GBD Algorithm

1. Let n be big-ish; we'll take $n = 50,000$.
2. Generate an $n \times 2$ matrix x of standard uniform observations.
3. Count the number of rows satisfying $x^2 + y^2 \leq 1$
4. Ask everyone else what their answer is; sum it all up.
5. Take this new answer, multiply by 4 and divide by n
6. If my rank is 0, print the result.

Example 1: Monte Carlo Simulation Code

Serial Code

```
1  N <- 50000
2  X <- matrix(runif(N * 2), ncol=2)
3  r <- sum(rowSums(X^2) <= 1)
4  PI <- 4*r/N
5  print(PI)
```

Parallel Code

```
1  library(pbdMPI, quiet = TRUE)
2  init()
3  comm.set.seed(diff=TRUE)
4
5  N.gbd <- 50000 / comm.size()
6  X.gbd <- matrix(runif(N.gbd * 2), ncol = 2)
7  r.gbd <- sum(rowSums(X.gbd^2) <= 1)
8  r <- allreduce(r.gbd)
9  PI <- 4*r/(N.gbd * comm.size())
10 comm.print(PI)
11
12 finalize()
```

Note

For the remainder, we will exclude loading, init, and finalize calls.

Example 2: Sample Covariance

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

Example 2: Sample Covariance GBD Algorithm

1. Determine the total number of rows N.
2. Compute the vector of column means of the full matrix.
3. Subtract each column's mean from that column's entries in each local matrix.
4. Compute the crossproduct locally and reduce.
5. Divide by $N - 1$.

Example 2: Sample Covariance Code

Serial Code

```
1 N <- nrow(X)
2 mu <- colSums(X) / N
3
4 X <- sweep(X, STATS=mu, MARGIN=2)
5 Cov.X <- crossprod(X) / (N-1)
6
7 print(Cov.X)
```

Parallel Code

```
1 N <- allreduce(nrow(X.gbd), op="sum")
2 mu <- allreduce(colSums(X.gbd) / N, op="sum")
3
4 X.gbd <- sweep(X.gbd, STATS=mu, MARGIN=2)
5 Cov.X <- allreduce(crossprod(X.gbd), op="sum") / (N-1)
6
7 comm.print(Cov.X)
```

Example 3: Linear Regression

Find β such that

$$\mathbf{y} = \mathbf{X}\beta + \epsilon$$

When $\mathbf{X}$ is full rank,

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Example 3: Linear Regression GBD Algorithm

1. Locally, compute $tx = x^T$
2. Locally, compute $A = tx * x$. Query every other processor for this result and sum up all the results.
3. Locally, compute $B = tx * y$. Query every other processor for this result and sum up all the results.
4. Locally, compute $A^{-1} * B$

Example 3: Linear Regression Code

Serial Code

```
1 tX <- t(X)
2 A <- tX %*% X
3 B <- tX %*% y
4
5 ols <- solve(A) %*% B
```

Parallel Code

```
1 tX.gbd <- t(X.gbd)
2 A <- allreduce(tX.gbd %*% X.gbd, op = "sum")
3 B <- allreduce(tX.gbd %*% y.gbd, op = "sum")
4
5 ols <- solve(A) %*% B
```

MPI Exercises

1. Experiment with Statistics Examples 1 through 3, running them on 2, 4, and 8 processors.

Advanced MPI Exercises I

1. Write a script that will have each processor randomly take a sample of size 1 of `TRUE` and `FALSE`. Have each processor print its result.
2. Modify the script in Exercise 1 above to determine if any processors sampled `TRUE`. Do the same to determine if all processors sampled `TRUE`. In each case, print the result. Compare to the functions `comm.all()` and `comm.any()`.
3. Generate $50,000,000$ (total) random normal values in parallel on 2, 4, and 8 processors. Time each run.

Advanced MPI Exercises II

4. Distribute the matrix `x <- matrix(1:24, nrow=12)` in GBD format across 4 processors and call it `x.spmd`.
   1. Add `x.spmd` to itself.
   2. Compute the mean of `x.spmd`.
   3. Compute the column means of `x.spmd`.

Contents

Distributed Matrices

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

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

Distributed Matrices

High level OOP allows *native* serial R syntax:

```
1 x <- x[-1, 2:5]
2 x <- log(abs(x) + 1)
3 xtx <- t(x) %*% x
4 ans <- svd(solve(xtx))
```

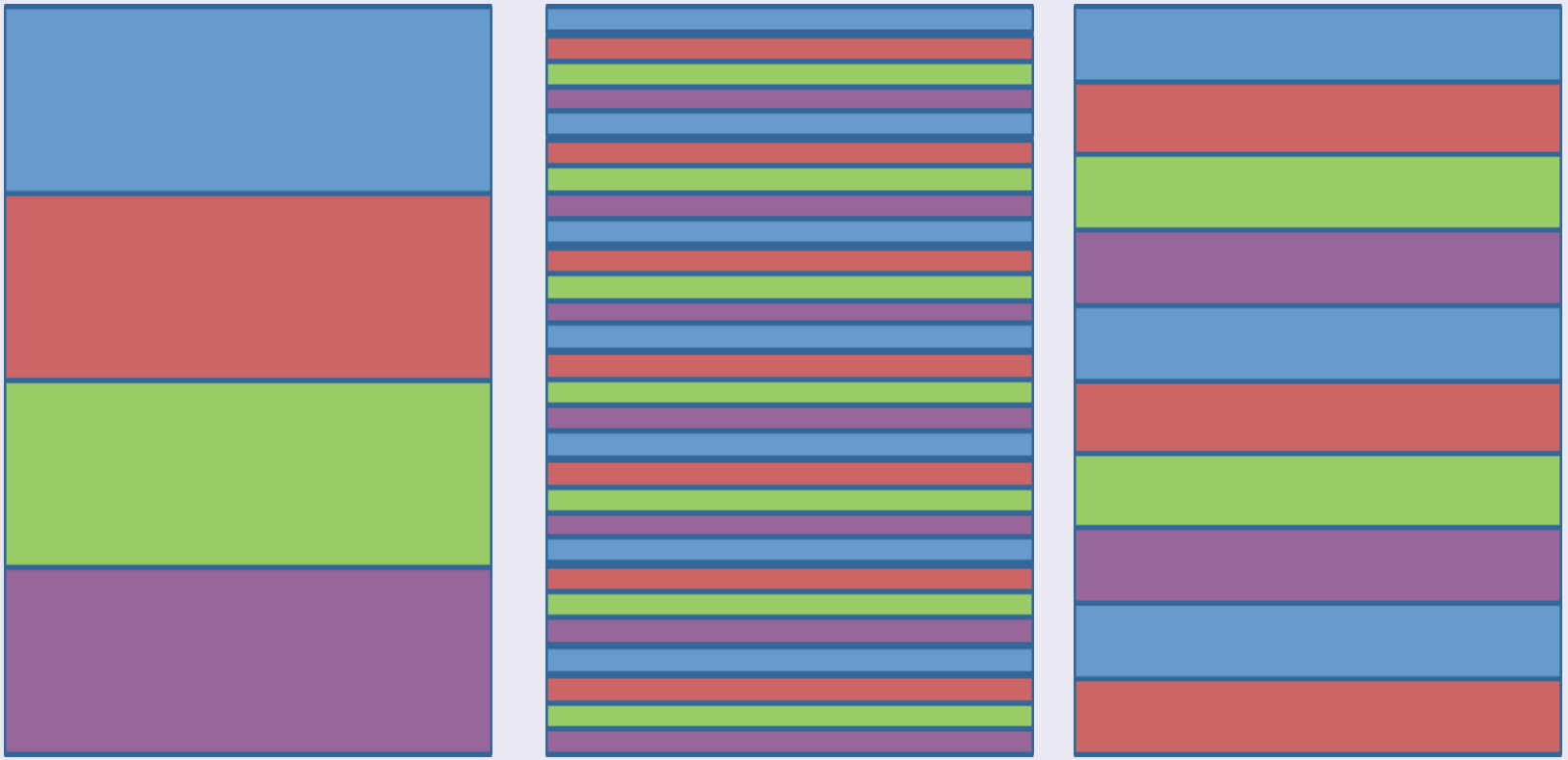
However...

Distributed Matrices

DMAT:

- **D**istributed **MAT**rix data structure.
- No single processor should hold all of the data.
- Block-cyclic matrix distributed across a 2-dimensional grid of processors.
- Very robust, but confusing data structure.

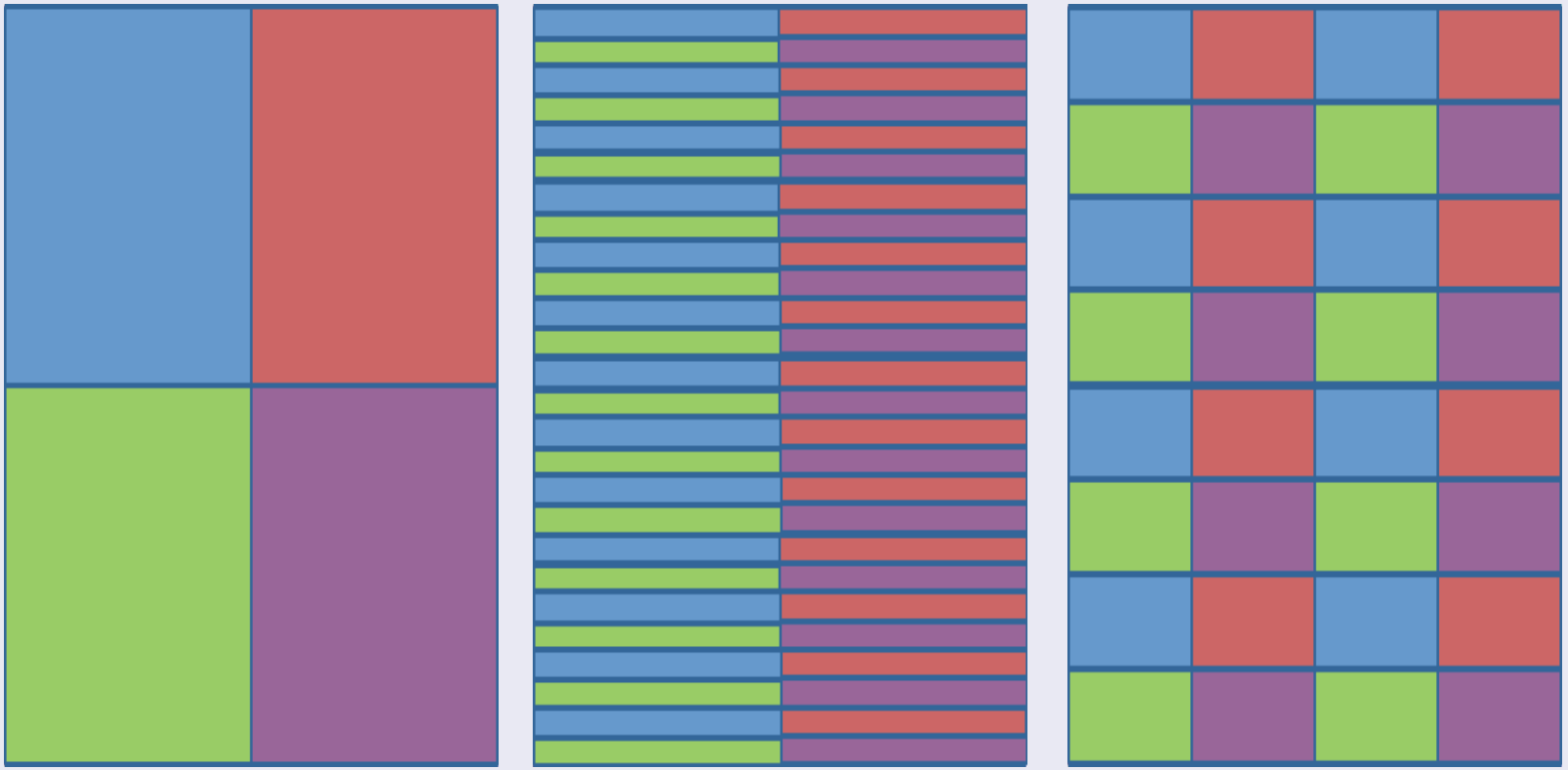
Distributed Matrices



(a) Block (b) Cyclic (c) Block-Cyclic

Figure: Matrix Distribution Schemes

Distributed Matrices



(a) 2d Block (b) 2d Cyclic (c) 2d Block-Cyclic

Figure: Matrix Distribution Schemes Onto a 2-Dimensional Grid

Processor Grid Shapes

$$\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}^T \qquad \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} \qquad \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

(a) 1×6 (b) 2×3 (c) 3×2 (d) 6×1

Table: Processor Grid Shapes with 6 Processors

Distributed Matrices

The data structure is a special R class (in the OOP sense) called `ddmatrix`. It is the "under the rug" storage for a block-cyclic matrix distributed onto a 2-dimensional processor grid.

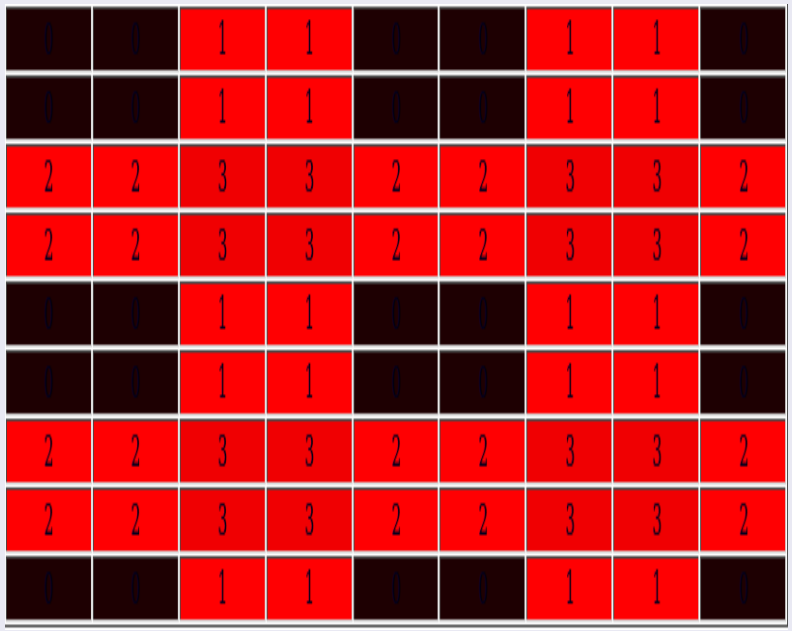
$$\texttt{ddmatrix} = \begin{cases} \textbf{Data} & \text{S4 local submatrix, an R matrix} \\ \textbf{dim} & \text{S4 dimension of the global matrix, a numeric pair} \\ \textbf{ldim} & \text{S4 dimension of the local submatrix, a numeric pair} \\ \textbf{bldim} & \text{S4 ScaLAPACK blocking factor, a numeric pair} \\ \textbf{CTXT} & \text{S4 BLACS context, an numeric singleton} \end{cases}$$

with prototype

$$\texttt{new("ddmatrix")} = \begin{cases} \textbf{Data} & = \texttt{matrix(0.0)} \\ \textbf{dim} & = \texttt{c(1,1)} \\ \textbf{ldim} & = \texttt{c(1,1)} \\ \textbf{bldim} & = \texttt{c(1,1)} \\ \textbf{CTXT} & = \texttt{0.0} \end{cases}$$

Distributed Matrices: The Data Structure

Example: an 9×9 matrix is distributed with a "block-cycling" factor of 2×2 on a 2×2 processor grid:



$$= \begin{cases} \textbf{Data} & = \texttt{matrix(...)} \\ \textbf{dim} & = \texttt{c(9, 9)} \\ \textbf{ldim} & = \texttt{c(...)} \\ \textbf{bldim} & = \texttt{c(2, 2)} \\ \textbf{CTXT} & = 0 \end{cases}$$

See http://acts.nersc.gov/scalapack/hands-on/datadist.html

Understanding Dmat: Global Matrix

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

DMAT: 1-dimensional Row Block

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

$$\text{Processor grid} = \left|\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}\right| = \left|\begin{array}{c} (0,0) \\ (0,1) \\ (1,0) \\ (1,1) \end{array}\right|$$

DMAT: 2-dimensional Row Block

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

$$\text{Processor grid} = \left|\begin{array}{cc} 0 & 1 \\ 2 & 3 \end{array}\right| = \left|\begin{array}{cc} (0,0) & (0,1) \\ (1,0) & (1,1) \end{array}\right|$$

DMAT: 1-dimensional Row Cyclic

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

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

DMAT: 2-dimensional Row Cyclic

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

$$\text{Processor grid} = \left|\begin{array}{cc} 0 & 1 \\ 2 & 3 \end{array}\right| = \left|\begin{array}{cc} (0,0) & (0,1) \\ (1,0) & (1,1) \end{array}\right|$$

DMAT: 2-dimensional Block-Cyclic

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

$$\text{Processor grid} = \left|\begin{array}{cc} 0 & 1 \\ 2 & 3 \end{array}\right| = \left|\begin{array}{cc} (0,0) & (0,1) \\ (1,0) & (1,1) \end{array}\right|$$

The DMAT Data Structure

The more complicated the processor grid, the more complicated the distribution.

DMAT: 2-dimensional Block-Cyclic with 6 Processors

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

$$\text{Processor grid} = \left|\begin{array}{ccc} 0 & 1 & 2 \\ 3 & 4 & 5 \end{array}\right| = \left|\begin{array}{ccc} (0,0) & (0,1) & (0,2) \\ (1,0) & (1,1) & (1,2) \end{array}\right|$$

Understanding DMAT: Local View

$$\left[\begin{array}{cc|cc} x_{11} & x_{12} & x_{17} & x_{18} \\ x_{21} & x_{22} & x_{27} & x_{28} \\ \hline x_{51} & x_{52} & x_{57} & x_{58} \\ x_{61} & x_{62} & x_{67} & x_{68} \\ \hline x_{91} & x_{92} & x_{97} & x_{98} \end{array}\right]_{5\times4} \left[\begin{array}{cc|c} x_{13} & x_{14} & x_{19} \\ x_{23} & x_{24} & x_{29} \\ \hline x_{53} & x_{54} & x_{59} \\ x_{63} & x_{64} & x_{69} \\ \hline x_{93} & x_{94} & x_{99} \end{array}\right]_{5\times3} \left[\begin{array}{cc} x_{15} & x_{16} \\ x_{25} & x_{26} \\ \hline x_{55} & x_{56} \\ x_{65} & x_{66} \\ \hline x_{95} & x_{96} \end{array}\right]_{5\times2}$$

$$\left[\begin{array}{cc|cc} x_{31} & x_{32} & x_{37} & x_{38} \\ x_{41} & x_{42} & x_{47} & x_{48} \\ \hline x_{71} & x_{72} & x_{77} & x_{78} \\ x_{81} & x_{82} & x_{87} & x_{88} \end{array}\right]_{4\times4} \left[\begin{array}{cc|c} x_{33} & x_{34} & x_{39} \\ x_{43} & x_{44} & x_{49} \\ \hline x_{73} & x_{74} & x_{79} \\ x_{83} & x_{84} & x_{89} \end{array}\right]_{4\times3} \left[\begin{array}{cc} x_{35} & x_{36} \\ x_{45} & x_{46} \\ \hline x_{75} & x_{76} \\ x_{85} & x_{86} \end{array}\right]_{4\times2}$$

$$\text{Processor grid} = \left|\begin{array}{ccc} 0 & 1 & 2 \\ 3 & 4 & 5 \end{array}\right| = \left|\begin{array}{ccc} (0,0) & (0,1) & (0,2) \\ (1,0) & (1,1) & (1,2) \end{array}\right|$$

The DMAT Data Structure

1. DMAT is *distributed*. No one processor owns all of the matrix.
2. DMAT is *non-overlapping*. Any piece owned by one processor is owned by no other processors.
3. DMAT can be row-contiguous or not, depending on the processor grid and blocking factor used.
4. DMAT is locally column-major and globally, it depends...

$$\left[\begin{array}{ccccc} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} \\ \hline x_{31} & x_{32} & x_{33} & x_{34} & x_{35} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} \\ \hline x_{51} & x_{52} & x_{53} & x_{54} & x_{55} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} \\ \hline x_{71} & x_{72} & x_{73} & x_{74} & x_{75} \\ x_{81} & x_{82} & x_{83} & x_{84} & x_{85} \\ \hline x_{91} & x_{92} & x_{93} & x_{94} & x_{95} \end{array}\right]$$

6. GBD is a generalization of the one-dimensional block DMAT distribution. Otherwise there is no relation.
7. DMAT is confusing, but very robust.

Pros and Cons of This Data Structure

Pros

- Fast for distributed matrix computations

Cons

- Literally everything else

This is why we hide most of the distributed details.

The details are there if you want them (you don't want them).

Distributed Matrix Methods

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

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

Serial Code

```
1 cov(x)
```

Parallel Code

```
1 cov(x)
```

Comparing pbdMPI and pbdDMAT

pbdMPI:

- MPI + sugar.
- `GBD` not the only structure **pbdMPI** can handle (just a useful convention).

pbdDMAT:

- More of a software package.
- `DMAT` structure *must* be used for **pbdDMAT**.
- If the data is not 2d block-cyclic compatible, DMAT will *definitely* give the wrong answer.

Quick Comments for Using pbdDMAT

1. Start by loading the package:

```
1 library(pbdDMAT, quiet = TRUE)
```

2. Always initialize before starting and finalize when finished:

```
1 init.grid()
2
3 # ...
4
5 finalize()
```

3. Distributed `DMAT` objects will be given the suffix `.dmat` to visually help distinguish them from global objects. This suffix carries no semantic meaning.

Contents

Sample Covariance

Serial Code

```
1 Cov.X <- cov(X)
2 print(Cov.X)
```

Parallel Code

```
1 Cov.X <- cov(X)
2 print(Cov.X)
```

Linear Regression

Serial Code

```
1 tX <- t(X)
2 A <- tX %*% X
3 B <- tX %*% y
4
5 ols <- solve(A) %*% B
6
7 # or
8 ols <- lm.fit(X, y)
```

Parallel Code

```
1 tX <- t(X)
2 A <- tX %*% X
3 B <- tX %*% y
4
5 ols <- solve(A) %*% B
6
7 # or
8 ols <- lm.fit(X, y)
```

Example 5: PCA

PCA: pca.r

```
1  library(pbdDMAT, quiet=T)
2  init.grid()
3
4  n <- 1e4
5  p <- 250
6
7  comm.set.seed(diff=T)
8  x.dmat <- ddmatrix("rnorm", nrow=n, ncol=p, mean=100, sd=25)
9
10 pca <- prcomp(x=x.dmat, retx=TRUE, scale=TRUE)
11 prop_var <- cumsum(pca$sdev)/sum(pca$sdev)
12 i <- max(min(which(prop_var > 0.9)) - 1, 1)
13
14 y.dmat <- pca$x[, 1:i]
15
16 comm.cat("\nCols: ", i, "\n", quiet=T)
17 comm.cat("%Cols:", i/dim(x.dmat)[2], "\n\n", quiet=T)
18
19 finalize()
```

Execute this script via:

```
1  mpirun -np 2 Rscript 5_pca.r
```

Sample Output:

```
1  Cols:  221
2  %Cols:  0.884
```

Distributed Matrices

pbdDEMO contains many other examples of reading and managing GBD and DMAT data

Randomized SVD[3]

Prototype for Randomized SVD

Given an $m \times n$ matrix $\boldsymbol{A}$, a target number k of singular vectors, and an exponent q (say, $q = 1$ or $q = 2$), this procedure computes an approximate rank-$2k$ factorization $\boldsymbol{U\Sigma V}^*$, where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthonormal, and $\boldsymbol{\Sigma}$ is nonnegative and diagonal.

Stage A:

1 Generate an $n \times 2k$ Gaussian test matrix $\boldsymbol{\Omega}$.
2 Form $\boldsymbol{Y} = (\boldsymbol{AA}^*)^q \boldsymbol{A\Omega}$ by multiplying alternately with $\boldsymbol{A}$ and $\boldsymbol{A}^*$.
3 Construct a matrix $\boldsymbol{Q}$ whose columns form an orthonormal basis for the range of $\boldsymbol{Y}$.

Stage B:

4 Form $\boldsymbol{B} = \boldsymbol{Q}^* \boldsymbol{A}$.
5 Compute an SVD of the small matrix: $\boldsymbol{B} = \widetilde{\boldsymbol{U}} \boldsymbol{\Sigma} \boldsymbol{V}^*$.
6 Set $\boldsymbol{U} = \boldsymbol{Q}\widetilde{\boldsymbol{U}}$.

Note: The computation of $\boldsymbol{Y}$ in step 2 is vulnerable to round-off errors. When high accuracy is required, we must incorporate an orthonormalization step between each application of $\boldsymbol{A}$ and $\boldsymbol{A}^*$; see Algorithm 4.4.

Algorithm 4.4: Randomized Subspace Iteration

Given an $m \times n$ matrix $\boldsymbol{A}$ and integers ℓ and q, this algorithm computes an $m \times \ell$ orthonormal matrix $\boldsymbol{Q}$ whose range approximates the range of $\boldsymbol{A}$.

1 Draw an $n \times \ell$ standard Gaussian matrix $\boldsymbol{\Omega}$.
2 Form $\boldsymbol{Y}_0 = \boldsymbol{A\Omega}$ and compute its QR factorization $\boldsymbol{Y}_0 = \boldsymbol{Q}_0 \boldsymbol{R}_0$.
3 **for** $j = 1, 2, \ldots, q$
4 Form $\widetilde{\boldsymbol{Y}}_j = \boldsymbol{A}^* \boldsymbol{Q}_{j-1}$ and compute its QR factorization $\widetilde{\boldsymbol{Y}}_j = \widetilde{\boldsymbol{Q}}_j \widetilde{\boldsymbol{R}}_j$.
5 Form $\boldsymbol{Y}_j = \boldsymbol{A}\widetilde{\boldsymbol{Q}}_j$ and compute its QR factorization $\boldsymbol{Y}_j = \boldsymbol{Q}_j \boldsymbol{R}_j$.
6 **end**
7 $\boldsymbol{Q} = \boldsymbol{Q}_q$.

Serial R

```
1  randSVD <- function(A, k, q=3)
2    {
3      ## Stage A
4      Omega <-  matrix(rnorm(n*2*k),
5                       nrow=n, ncol=2*k)
6      Y <- A %*% Omega
7      Q <- qr.Q(qr(Y))
8      At <- t(A)
9      for(i in 1:q)
10       {
11         Y <- At %*% Q
12         Q <- qr.Q(qr(Y))
13         Y <- A %*% Q
14         Q <- qr.Q(qr(Y))
15       }
16
17     ## Stage B
18     B <- t(Q) %*% A
19     U <- La.svd(B)$u
20     U <- Q %*% U
21     U[, 1:k]
22   }
```

[1] Halko N, Martinsson P-G and Tropp J A 2011 Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions *SIAM Rev.* **53** 217–88

Randomized SVD

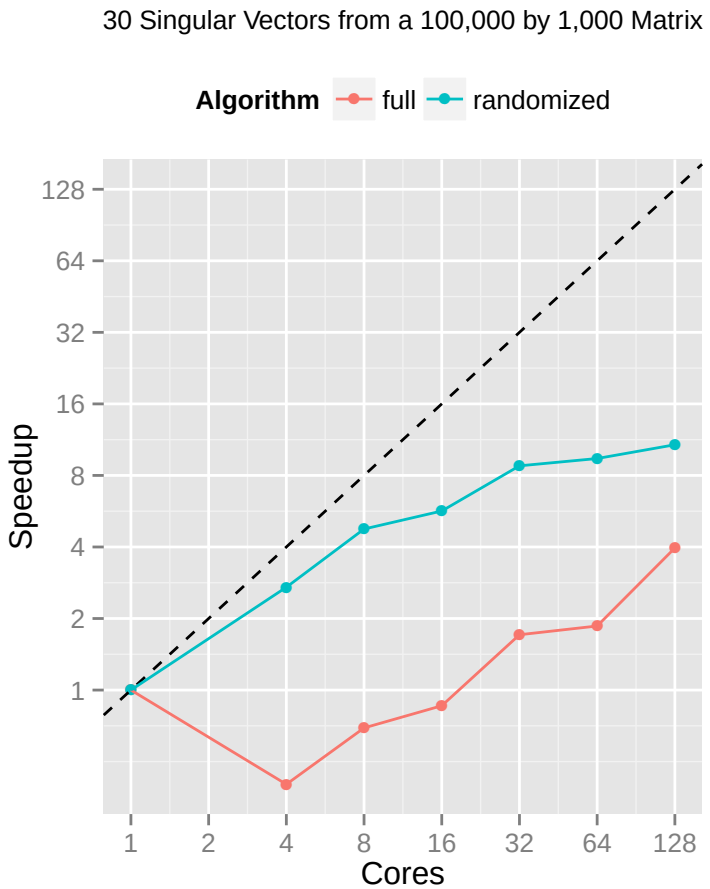
Serial R

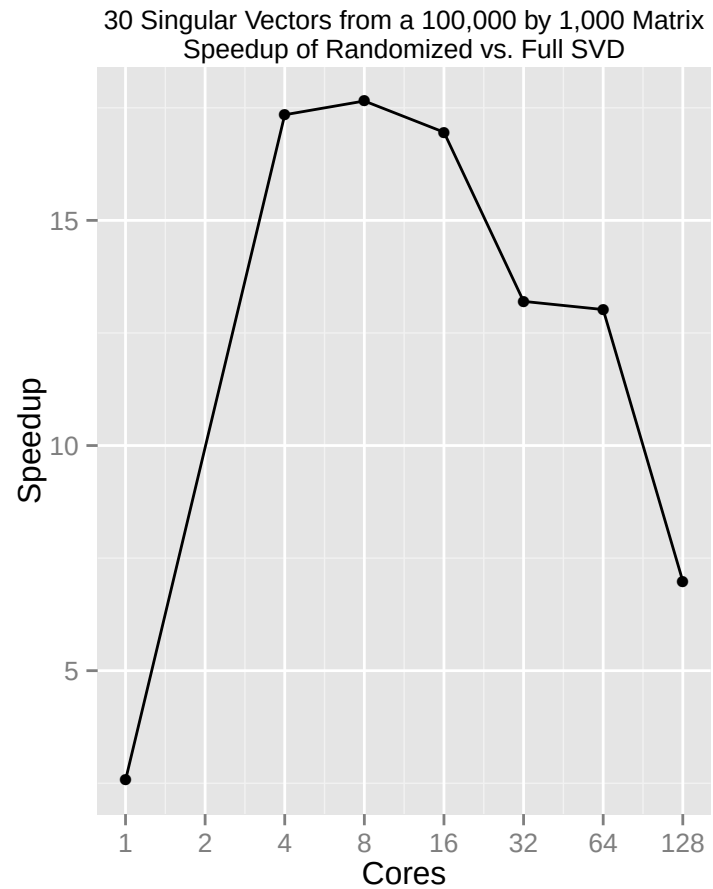
```
1  randSVD <- function(A, k, q=3)
2    {
3      ## Stage A
4      Omega <-  matrix(rnorm(n*2*k),
5              nrow=n, ncol=2*k)
6      Y <- A %*% Omega
7      Q <- qr.Q(qr(Y))
8      At <- t(A)
9      for(i in 1:q)
10       {
11         Y <- At %*% Q
12         Q <- qr.Q(qr(Y))
13         Y <- A %*% Q
14         Q <- qr.Q(qr(Y))
15       }
16
17     ## Stage B
18     B <- t(Q) %*% A
19     U <- La.svd(B)$u
20     U <- Q %*% U
21     U[, 1:k]
22   }
```

Parallel pbdR

```
1  randSVD <- function(A, k, q=3)
2    {
3      ## Stage A
4      Omega <-  ddmatrix("rnorm",
5              nrow=n, ncol=2*k)
6      Y <- A %*% Omega
7      Q <- qr.Q(qr(Y))
8      At <- t(A)
9      for(i in 1:q)
10       {
11         Y <- At %*% Q
12         Q <- qr.Q(qr(Y))
13         Y <- A %*% Q
14         Q <- qr.Q(qr(Y))
15       }
16
17     ## Stage B
18     B <- t(Q) %*% A
19     U <- La.svd(B)$u
20     U <- Q %*% U
21     U[, 1:k]
22   }
```

Randomized SVD





DMAT Exercises

1. Experiment with DMAT Examples 1 through 5, running them on 2 and 4 processors.

Advanced DMAT Exercises I

1. Subsetting, selection, and filtering are basic matrix operations featured in R. The following may look silly, but it is useful for data processing. Let `x.dmat <- ddmatrix(1:30, 10, 3)`. Do the following:
   - ```
     y.dmat <- x.dmat[c(1, 5, 4, 3), ]
     y.dmat <- x.dmat[c(10:3, 5, 5), ]
     y.dmat <- x.dmat[1:5, 3:1]
     ```
   - ```
     y.dmat <- x.dmat[x.dmat[, 2] > 13, ]
     y.dmat <- x.dmat[x.dmat[, 2] > x.dmat[, 3], ]
     y.dmat <- x.dmat[, x.dmat[2,] > x.dmat[3, ]]
     y.dmat <- x.dmat[c(1, 3, 5), x.dmat[, 2] >
     x.dmat[, 3]]
     ```

Advanced DMAT Exercises II

2. The method `crossprod()` is an optimized form of the crossproduct computation `t(x.dmat) %*% x.dmat`. For this exercise, let `x.dmat <- ddmatrix(1:30, nrow=10, ncol=3)`.
   1. Verify that these computations really do produce the same results.
   2. Time each operation. Which is faster?
3. The `prcomp()` method returns rotations for all components. Computationally verify by example that these rotations are orthogonal, i.e., that their crossproduct is the identity matrix.

Contents

Where to Learn More

- Our website http://r-pbd.org/
- The **pbdDEMO** package http://cran.r-project.org/web/packages/pbdDEMO/
- The **pbdDEMO** Vignette: http://goo.gl/HZkRt
- Our Google Group: http://group.r-pbd.org

Tutorials

- OLCF Data Workshop, August 8, Oak Ridge National Laboratory
- SC13, November 17-22, Denver, Colorado

Invited Talks

- JSM 2013, August 3-8, Montréal, Québec
- IASC, Aug 22-23, Seoul
- World Statistics Congress, August 25-30, Hong Kong

Thanks for coming!

Questions? Comments?