Introducing R: From Your Laptop to HPC and Big Data

Drew Schmidt and George Ostrouchov

SC Tutorial, New Orleans, November 16-17, 2014

The pbdR Core Team

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

Support

This work used resources of National Institute for Computational Sciences at the University of Tennessee, Knoxville, which is supported by the Office of Cyberinfrastructure of the U.S. National Science Foundation under Award No. ARRA-NSF-OCI-0906324 for NICS-RDAV center. This work used resources of the Newton HPC Program at the University of Tennessee, Knoxville. This work also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

[1]University of Tennessee. 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.

[2]Oak Ridge National Laboratory. 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.

About This Presentation

Downloads

This presentation and supplemental materials are available at:

http://r-pbd.org/tutorial

About This Presentation

Tutorial Evaluations

http://bit.ly/sc13-tut-mf08

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

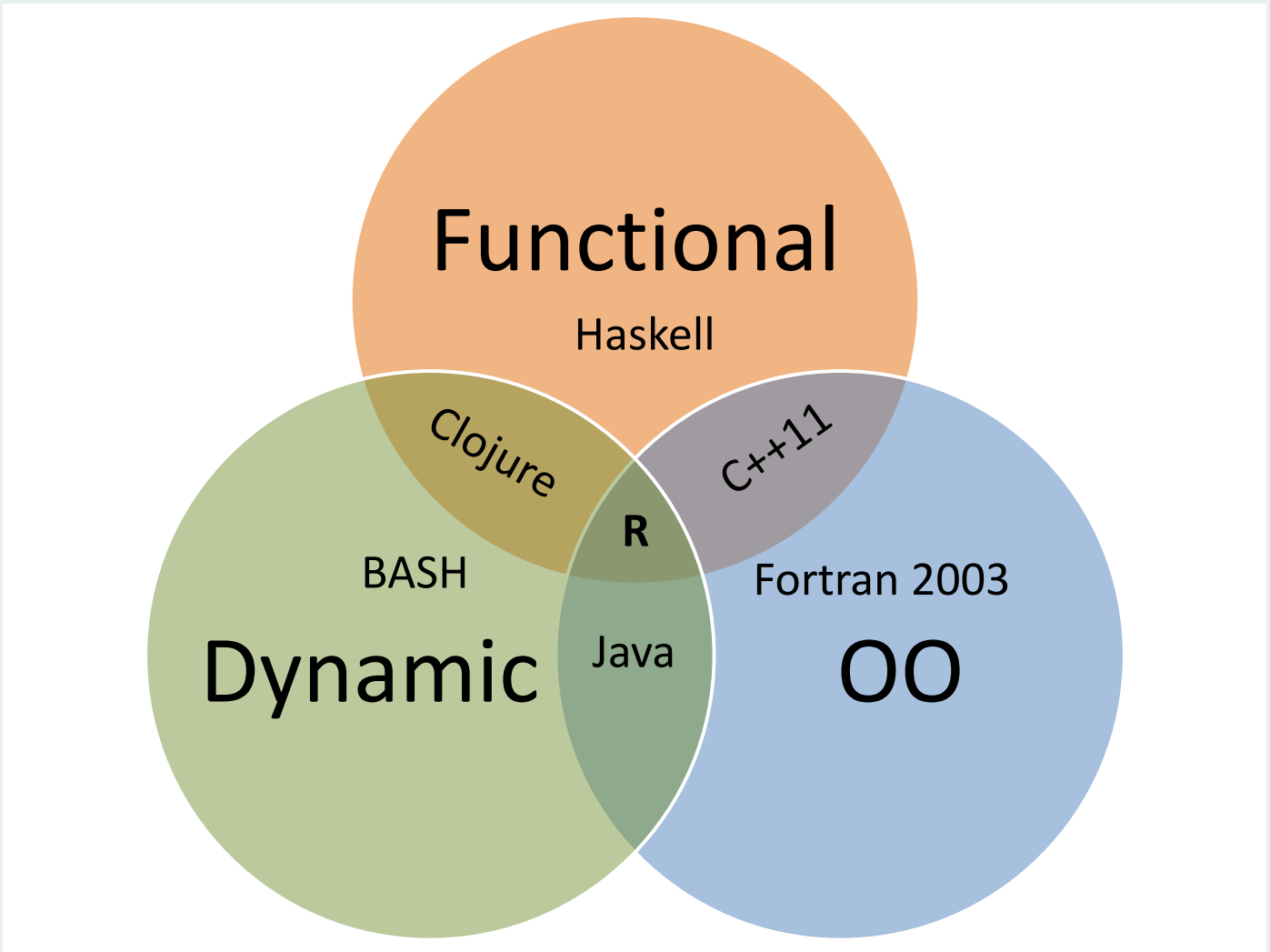
What is R?

- *lingua franca* for data analytics and statistical computing.
- Part programming language, part data analysis package.
- Dialect of S (Bell Labs).
- Started May 5, 1976.
- Syntax designed for data.

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.

Basics (1 of 2)

- The default method is to print:

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

- Use <- for assignment:

```
1 R> x <- 1
2 R> x+1
3 [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)

```
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)

```
1  R> dim(x)
2  [1] 2 3
3
4  R> dim(x) <- NULL
5  R> x
6  [1] 1 2 3 4 5 6
7
8  R> dim(x) <- c(3,2)
9  R> x
10      [,1] [,2]
11 [1,]    1    4
12 [2,]    2    5
13 [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

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

```
1  R> x <- matrix(rnorm(5*5), nrow=5)
2  R> y <- solve(x)
3
4  R> round(x %*% y)
5       [,1] [,2] [,3] [,4] [,5]
6  [1,]    1    0    0    0    0
7  [2,]    0    1    0    0    0
8  [3,]    0    0    1    0    0
9  [4,]    0    0    0    1    0
10 [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

```
1  R> x <- matrix(rnorm(30, mean=10, sd=3), nrow=10)
2
3  R> mean(x)
4  [1] 9.825177
5
6  R> median(x)
7  [1] 9.919243
8
9  R> sd(as.vector(x))
10 [1] 3.239388
11
12 R> colMeans(x)
13 [1]  9.661822 10.654686  9.159025
14
15 R> apply(x, MARGIN=2, FUN=sd)
16 [1] 2.101059 3.377347 4.087131
```

Simple Statistics (2 of 2): Sample Covariance

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

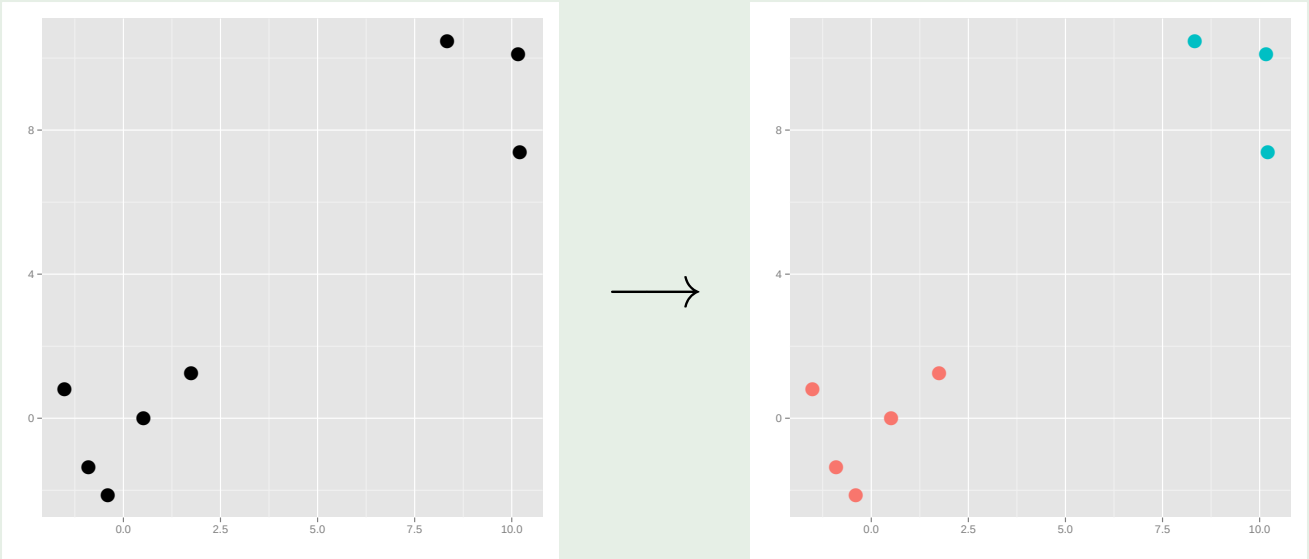
```
1  x <- matrix(rnorm(30), nrow=10)
2
3  # least recommended
4  cm <- colMeans(x)
5  crossprod(sweep(x, MARGIN=2, STATS=cm))
6
7  # less recommended
8  crossprod(scale(x, center=TRUE, scale=FALSE))
9
10 # recommended
11 cov(x)
```

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



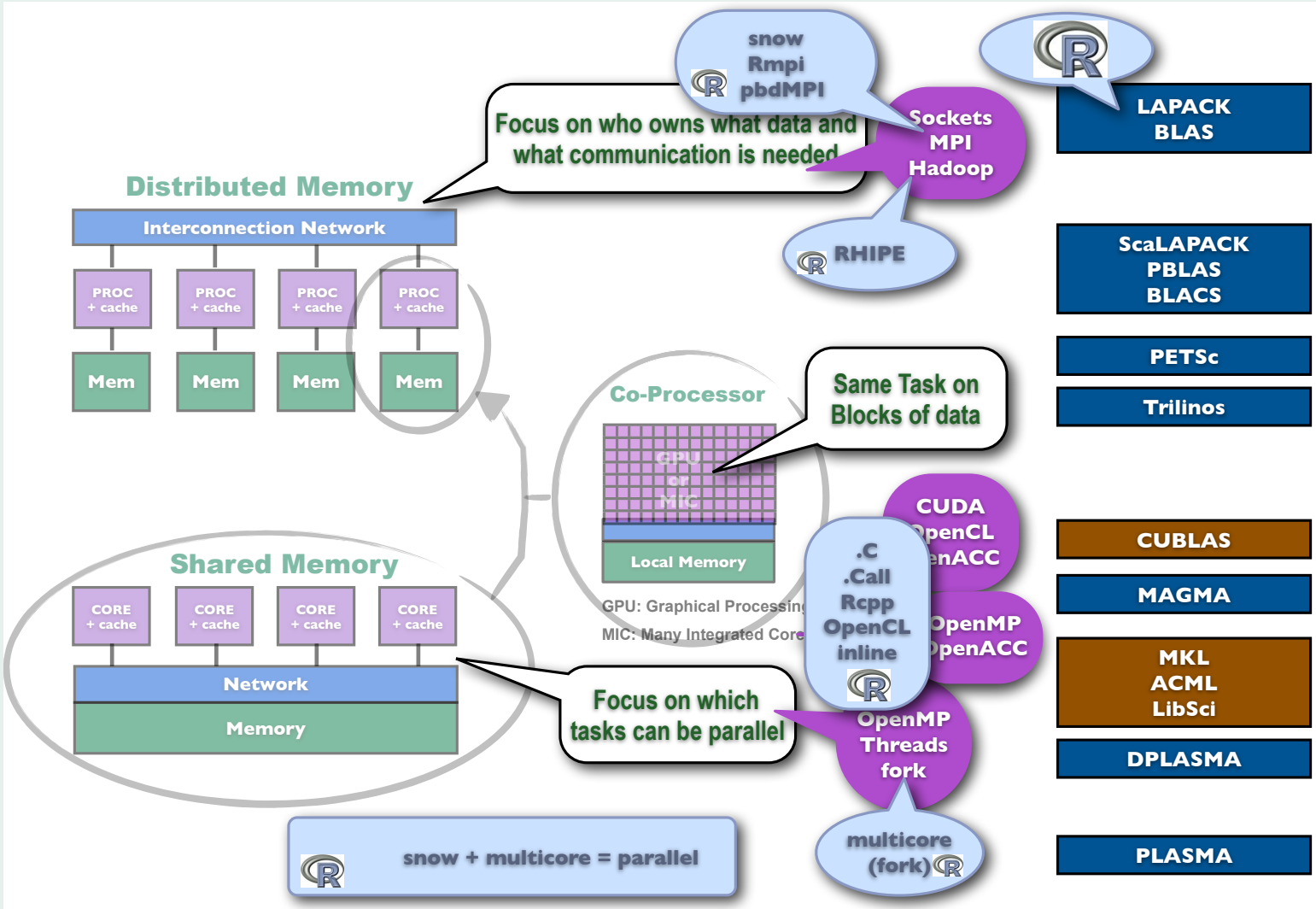

```
R> x <- rbind(matrix(rnorm(5*2, mean=0), ncol=2),
              matrix(rnorm(3*2, mean=10), ncol=2))
```

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

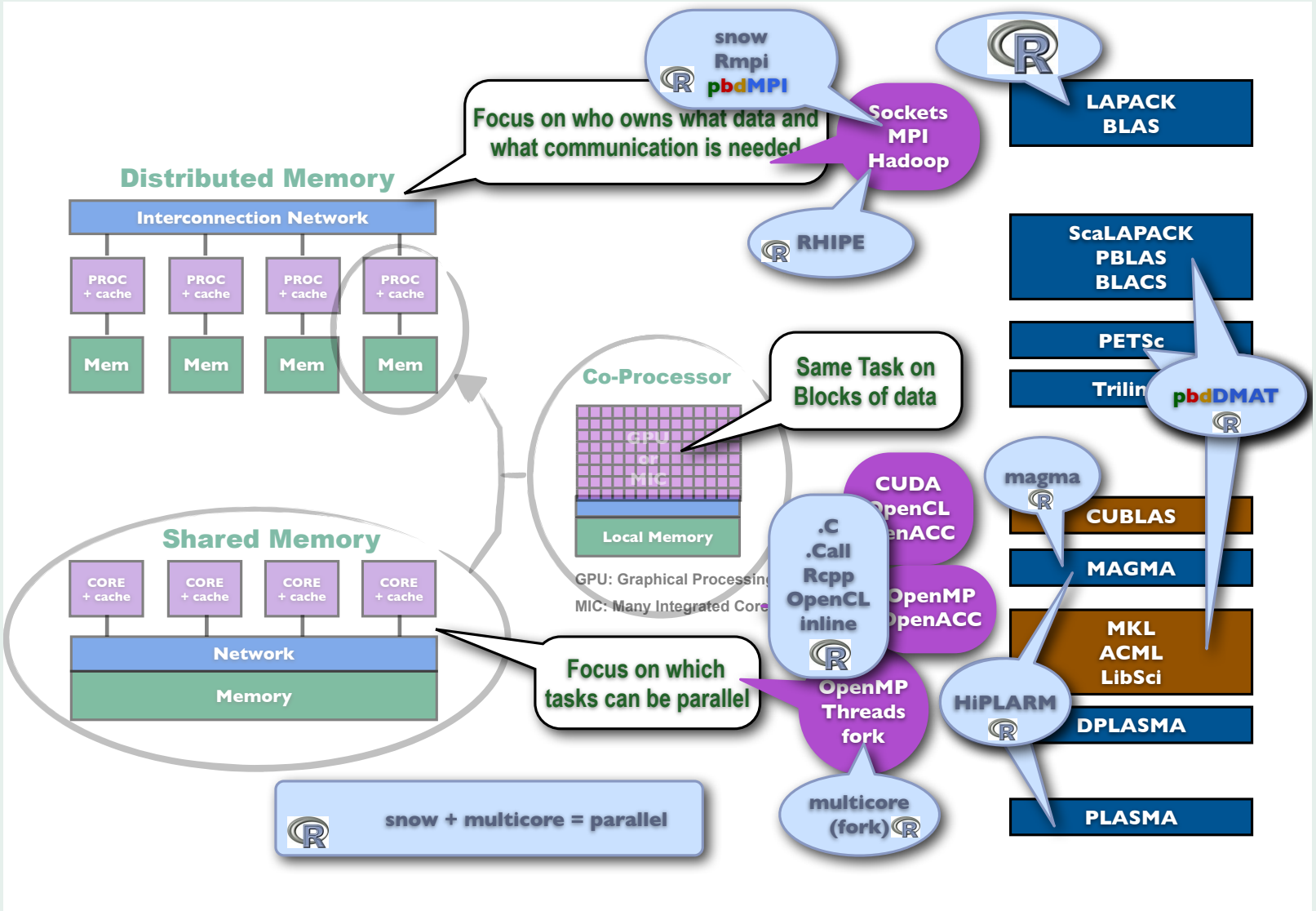
```
R> kmeans(x, centers=2)
K-means clustering with 2 clusters of sizes 5, 3

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"
```

Contents

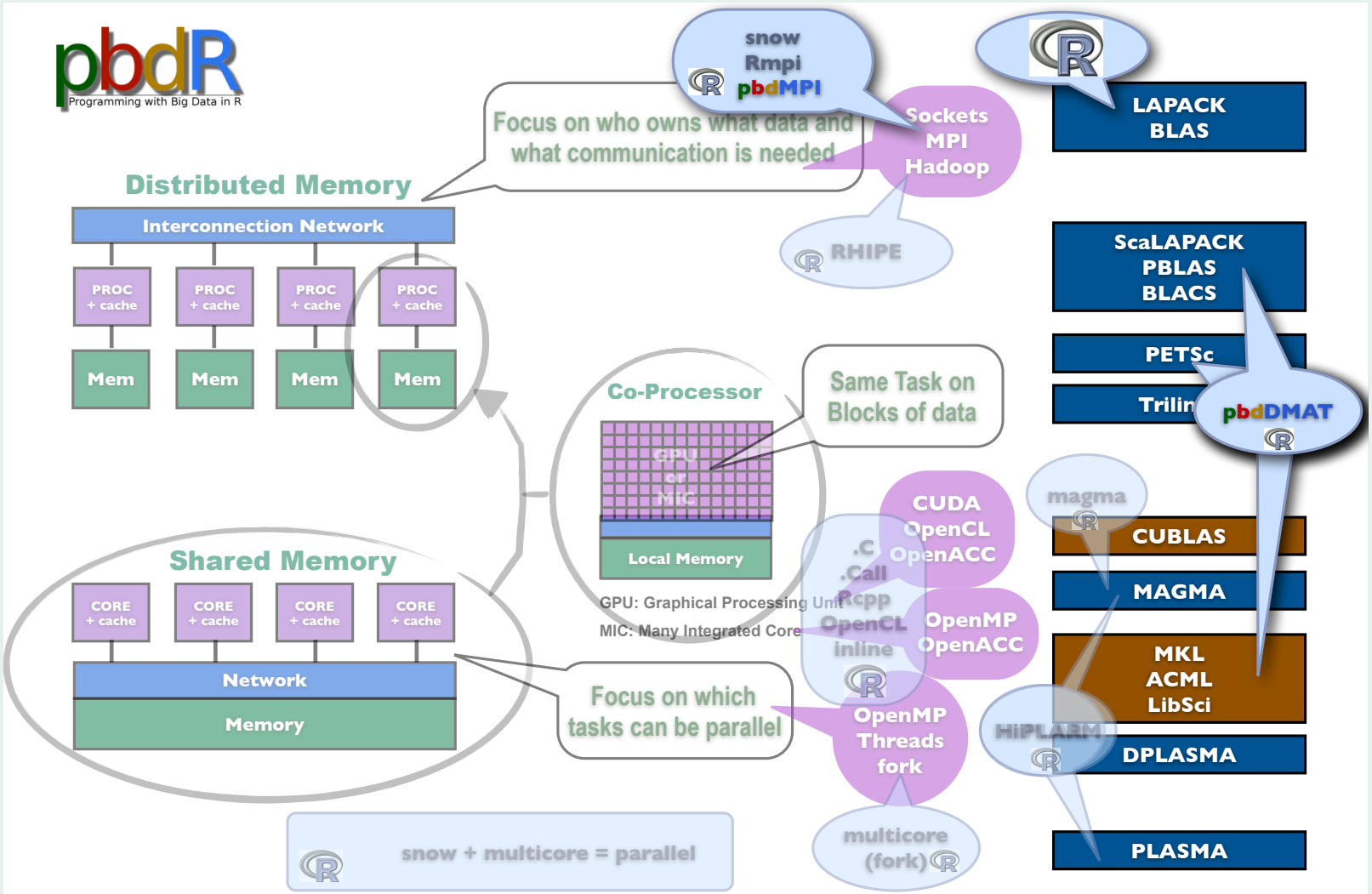
HPC Libraries



R Interfaces to Scalable HPC Libraries



pbdR Interfaces to Scalable HPC Libraries



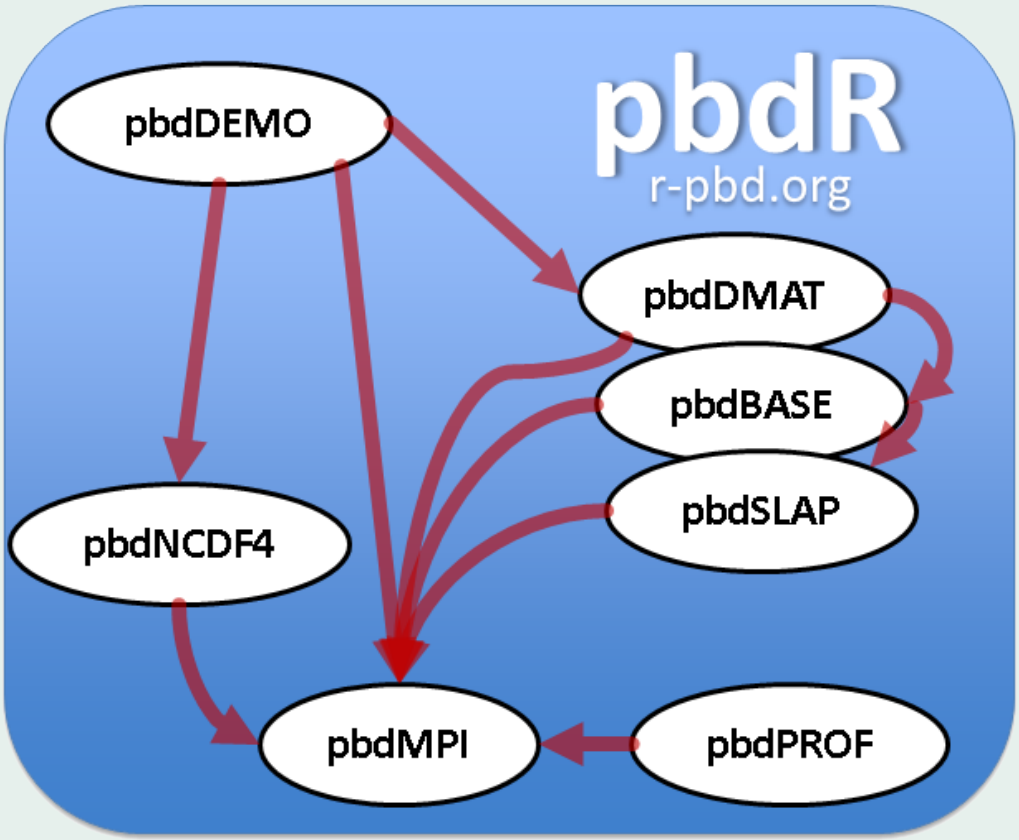
Programming with Big Data in R (pbdR)

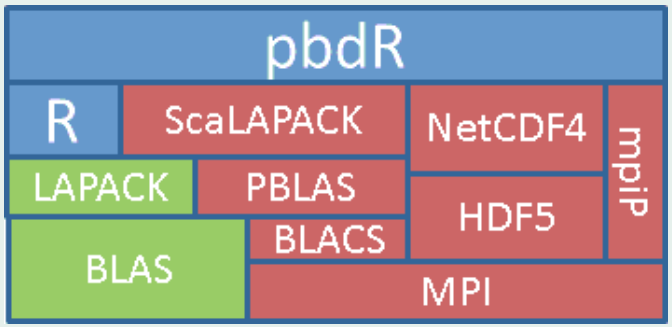
Striving for *Productivity, Portability, Performance*

- *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



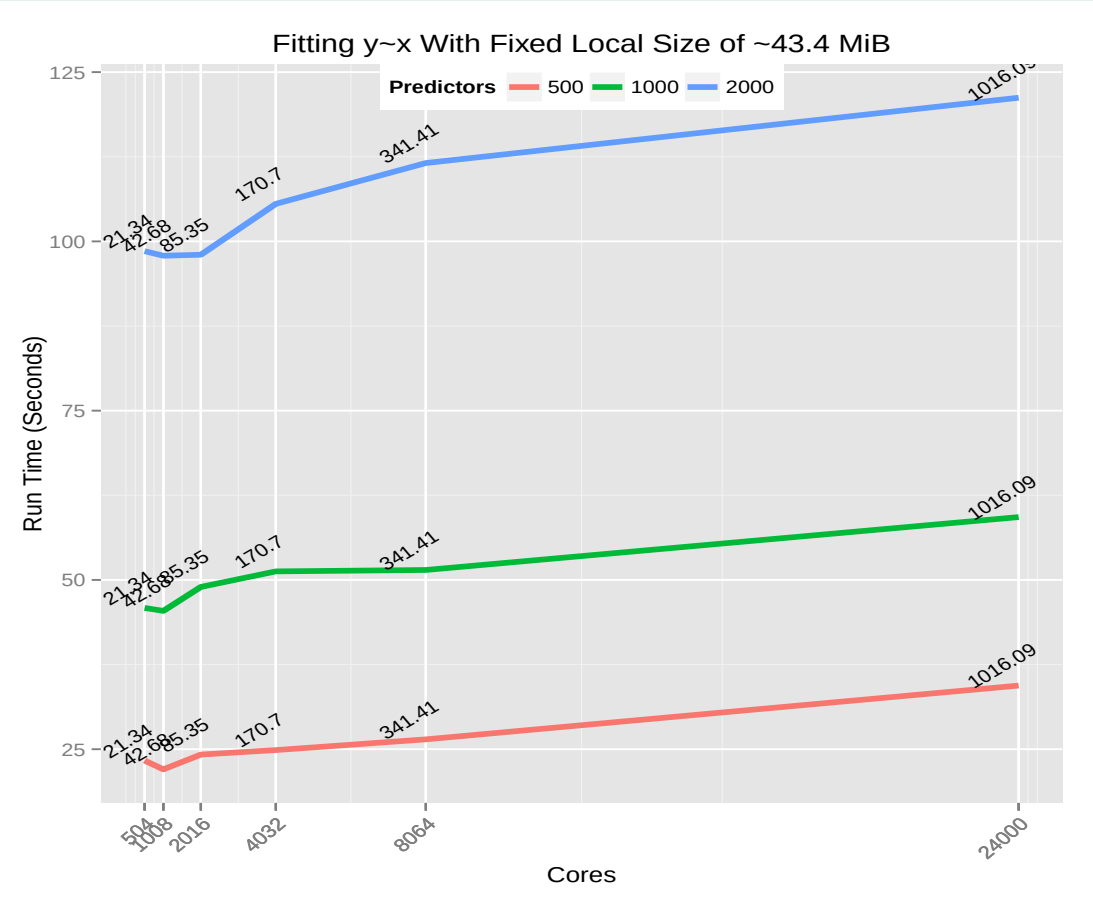


Simple, Intuitive MPI Operations with **pbdMPI**

Placeholder

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"
```

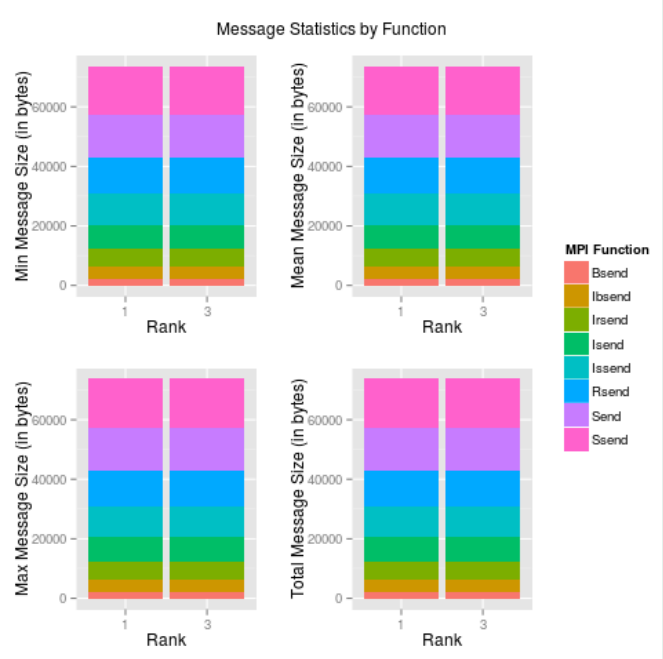
2. Run code

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

3. Analyze results

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

Publication-quality graphs



pbdR Paradigms

pbdR programs are R programs!

Differences:

- 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:

```
1 Rscript my_script.r
```

or

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

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

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

Single Program/Multiple Data (SPMD)

- SPMD is a programming *paradigm*.
- Not to be confused with SIMD.

Paradigms

Programming models
e.g. Procedural, OOP, Functional, SPMD, ...

SIMD

Hardware instructions
e.g. MMX, SSE, ...

SPMD is arguably the simplest extension of serial programming.

Single Program/Multiple Data (SPMD)

- 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, quietly = 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, quietly=TRUE)
2 init()
3
4 comm.print("Hello, world")
5
6 comm.print("Hello again", all.rank=TRUE, quietly=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 (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

```
1  library(pbdMPI, quietly=TRUE)
2  init()
3
4  comm.set.seed(diff=TRUE)
5
6  n <- sample(1:10, size=1)
7
8  gt <- gather(n)
9  comm.print(unlist(gt))
10
11 sm <- allreduce(n, op='sum')
12 comm.print(sm, all.rank=T)
13
14 finalize()
```

Execute this script via:

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

Sample Output:

```
1  COMM.RANK = 0
2  [1] 2 8
3  COMM.RANK = 0
4  [1] 10
5  COMM.RANK = 1
6  [1] 10
```

Quick Example 4

Broadcast: 4_bcast.r

```
1  library(pbdMPI, quietly=T)
2  init()
3
4  if (comm.rank()==0){
5    x <- matrix(1:4, nrow=2)
6  } else {
7    x <- NULL
8  }
9
10 y <- bcast(x, rank.source=0)
11
12 comm.print(y, rank=1)
13
14 finalize()
```

Execute this script via:

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

Sample Output:

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

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

Quick Example 5

Barrier: 5_barrier.r

```
1  library(pbdMPI, quiet = TRUE)
2  init()
3
4  .SPMD.CT$msg.barrier <- TRUE
5  .SPMD.CT$print.quiet <- TRUE
6
7  for (rank in 1:comm.size()-1){
8    if (comm.rank() == rank){
9      cat(paste("Hello", rank+1, "of", comm.size(), "\n"))
10   }
11   barrier()
12 }
13
14 comm.cat("\n")
15
16 comm.cat(paste("Hello", comm.rank()+1, "of",
       comm.size(), "\n"), all.rank=TRUE)
17
18 finalize()
```

Execute this script via:

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

Sample Output:

```
1  Hello 1 of 2
2  Hello 2 of 2
```

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).

Quick Example 6

Timing: 6_timer.r

```
1  library(pbdMPI, quiet=TRUE)
2  init()
3
4  comm.set.seed(diff=T)
5
6  test <- function(timed)
7  {
8    ltime <- system.time(timed)[3]
9
10   mintime <- allreduce(ltime, op='min')
11   maxtime <- allreduce(ltime, op='max')
12   meantime <- allreduce(ltime, op='sum')/comm.size()
13
14   return(data.frame(min=mintime, mean=meantime,
         max=maxtime))
15 }
16
17 times <- test(rnorm(1e6)) # ~7.6MiB of data
18 comm.print(times)
19
20 finalize()
```

Execute this script via:

```
1  mpirun -np 2 Rscript 6_timer.r
```

Sample Output:

```
1      min  mean   max
2  1 0.17 0.173 0.176
```

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()
```

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

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

Processors

0
1
2
3

Distributing a Matrix Across 4 Processors: Local Load Balance

Data

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

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.

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.

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

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

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

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

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

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

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

Processors = 0 1 2 3 4 5

Basic MPI Exercises

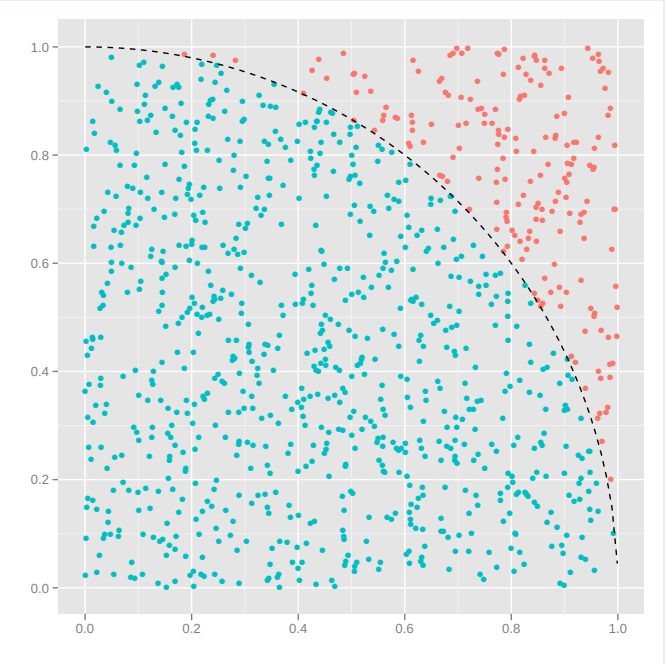
1. Experiment with Quick Examples 1 through 6, running them on 2, 4, and 8 processors.

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

$$y = X\beta + \epsilon$$

When X is full rank,

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

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

Separate manual: http://r-project.org/

- scan()
- read.table()
- read.csv()
- ...
- readBin()
- ncvar_get()
- readSocket()

No parallel file system: Read Serial then Distribute

read.csv()

```
1 library(pbdDMAT)
2 if(comm.rank() == 0) { # only read on process 0
3   x <- read.csv("myfile.csv")
4 } else {
5   x <- NULL
6 }
7
8 dx <- as.ddmatrix(x)
```

New Issues

- How to read in parallel?
- CSV, SQL, NetCDF4, HDF, ADIOS, custom binary
- How to partition data across nodes?
- How to structure for scalable libraries?
- Read directly into form needed or restructure?
- ...
- A lot of work needed here!

CSV Data

Serial Code

```
1 d <- read.csv(``x.csv'')
```

Parallel Code 0_readcsv.r

```
1 library(pbdDEMO, quiet = TRUE)
2 init.grid()
3 dx <- read.csv.ddmatrix("x.csv", header=TRUE,
4           sep='','', nrows=10, ncols=10,
5           num.rdrs=2, ICTXT=0)
6 comm.print(dx)
7 finalize()
```

NetCDF4 Data

Parallel Read

```
1  ### Must determine who will read what portion(s) and how
       to assemble them before reading
2
3  ### parallel read after determining st and co
4  nc <- nc_open_par(file.name)
5
6  nc_var_par_access(nc, "TREFHT")
7  new.X.gbdc <- ncvar_get(nc, "TREFHT", start = st, count
       = co)
8  nc_close(nc)
9
10 finalize()
```

Binary Data

Read subcube

```
1  library(pbdDMAT, quiet = TRUE)
2  init.grid()
3
4  data.dim <- c(2048, 2048, 2048) # full data dimension
5  g.start <- c(1, 1, 513)         # global subcube corner
6  g.dim <- c(64, 64, 1024)        # global subcube extent
7
8  my.start <- g.start + c(0, 0, comm.rank()*my.dim[3])
9  my.dim <- g.dim / c(1, 1, comm.size())
10
11 size <- 4 # file is single precision floats
12
13 vx <- block3d.read(``filename'', data.dim, my.start,
       my.dim, size)
14
15 ## local reshape dimensions
16 my.nrow <- prod(my.dim[1:2])
17 my.ncol <- my.dim[3]
18 ldim <- c(my.nrow, my.ncol)
19
20 ## global reshape dimensions
21 g.nrow <- prod(g.dim[1:2])
22 g.ncol <- g.dim[3]
23 gdim <- c(g.nrow, g.ncol)
24
25 ## reshape local
26 X <- matrix(vx, nrow=my.nrow, ncol=my.ncol, byrow=FALSE)
27
28 ## glue local pieces into a ddmatrix
29 X <- new("ddmatrix", Data=X, dim=gdim, ldim=ldim,
       bldim=ldim, ICTXT=1)
30
31 ## transform to 2d block cyclic
32 X <- redistribute(X, bldim=c(8,8), ICTXT=0)
```

Binary Data

3d Block Binary Reader

```
1  block3d.read <- function(file, data.dim, my.start,
       my.dim, size=4) {
2    con.x <- file(file, "rb", blocking=TRUE)
3
4    start <- sum((my.start - 1) * c(1,
         cumprod(data.dim)[-length(data.dim)]))
5
6    x <- rep(NA, prod(my.dim))
7
8    block <- 1:my.dim[1]
9
10   for(j in 1:my.dim[3]) {
11     sofar <- 0
12     for(i in 1:my.dim[2]) {
13       seek(con.x, where=start, rw="read", origin="start")
14       x[block] <- readBin(con=con.x, what="numeric",
             n=my.dim[1], size=size)
15       block <- block + my.dim[1]
16
17       start <- start + data.dim[1]*size
18       sofar <- sofar + data.dim[1]*size
19     }
20     start <- start - sofar + data.dim[1]*data.dim[2]*size
21   }
22
23   close(con.x)
24   x
25 }
```

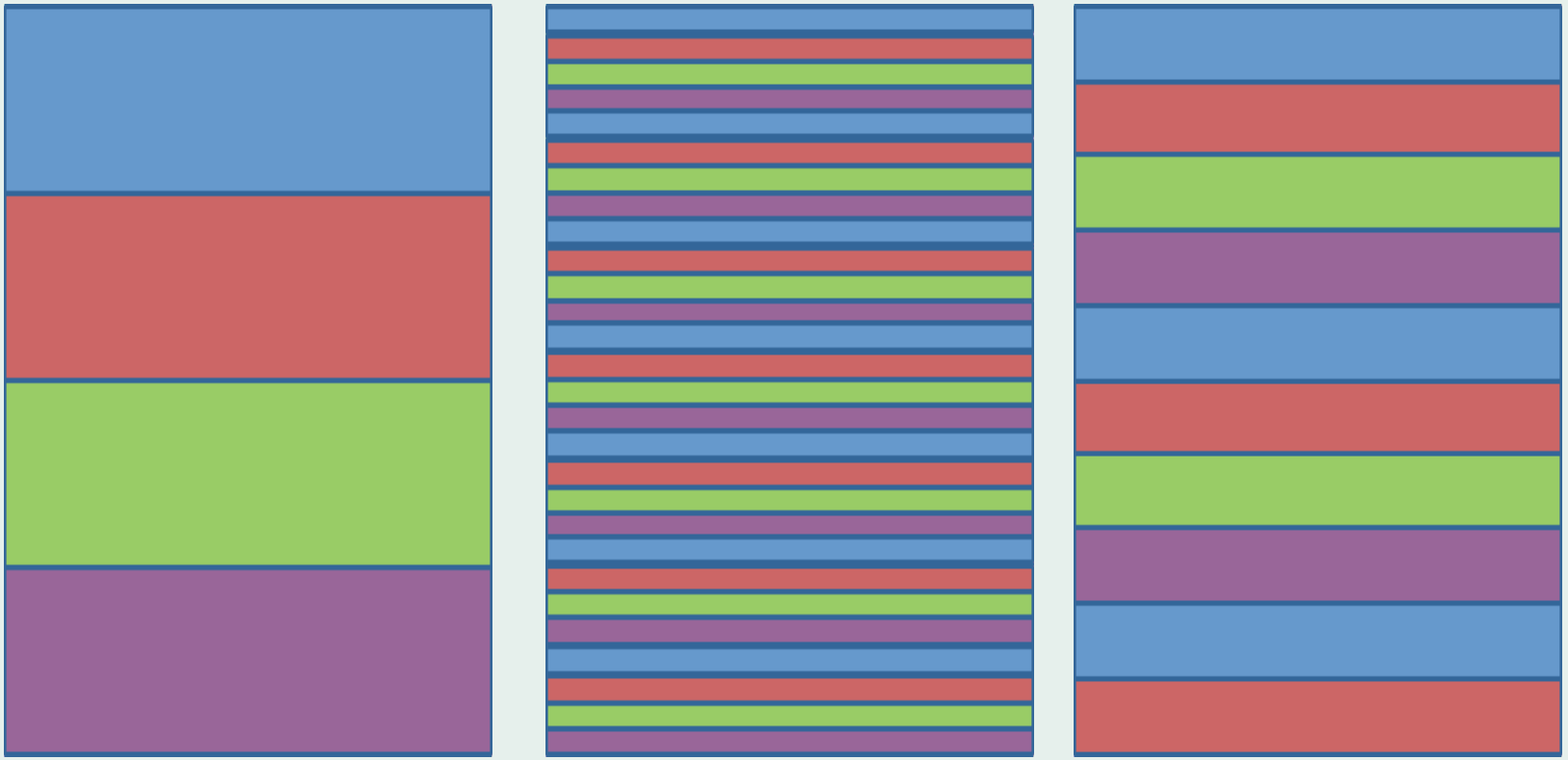
Contents

Distributed Matrices

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

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

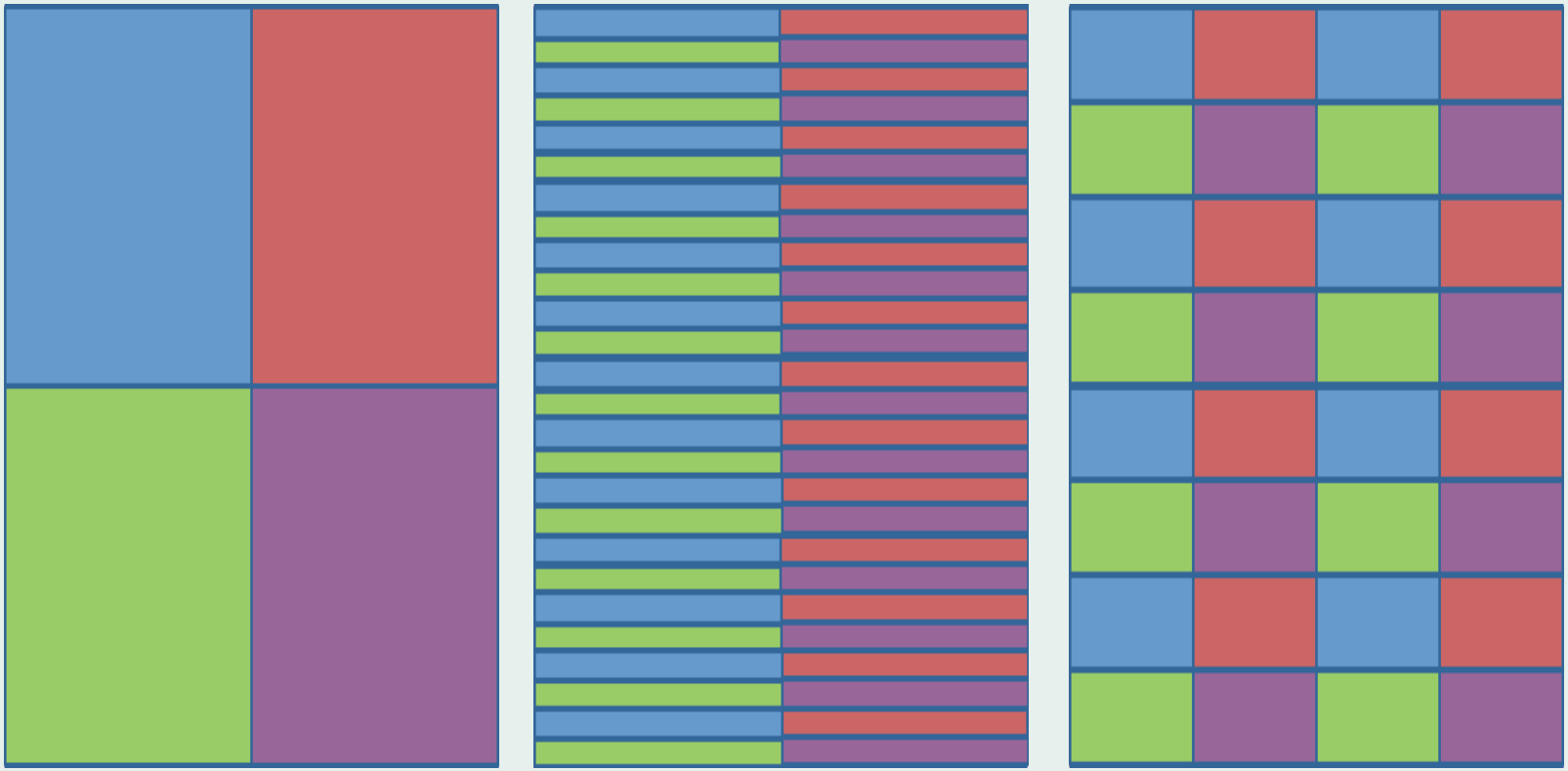
Distributed Matrices



(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

Understanding ddmatrix: Global Matrix

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

ddmatrix: 1-dimensional Row Block

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

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

ddmatrix: 2-dimensional Row Block

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

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

ddmatrix: 1-dimensional Row Cyclic

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

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

ddmatrix: 2-dimensional Row Cyclic

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

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

ddmatrix: 2-dimensional Block-Cyclic

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

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

The `ddmatrix` Data Structure

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

ddmatrix: 2-dimensional Block-Cyclic with 6 Processors

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

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

Understanding `ddmatrix`: Local View

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

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

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

The ddmatrix Data Structure

1. ddmatrix is *distributed*. No one processor owns all of the matrix.
2. ddmatrix is *non-overlapping*. Any piece owned by one processor is owned by no other processors.

---

3. ddmatrix can be row-contiguous or not, depending on the processor grid and blocking factor used.
4. ddmatrix 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 ddmatrix distribution. Otherwise there is no relation.
7. ddmatrix is confusing, but very robust.

Pros and Cons of This Data Structure

Pros

- Robust for matrix computations.

Cons

- Confusing layout.

This is why we hide most of the distributed details.

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

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

```
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.
- The `ddmatrix` structure *must* be used for **pbdDMAT**.
- If the data is not 2d block-cyclic compatible, `ddmatrix` 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 `ddmatrix` objects will be given the suffix `.dmat` to visually help distinguish them from global objects. This suffix carries no semantic meaning.

Contents

Example 1: ddmatrix Construction

Generate a global matrix and distribute it

```
1  library(pbdDMAT, quiet=TRUE)
2  init.grid()
3
4  # Common global on all processors --> distributed
5  x <- matrix(1:100, nrow=10, ncol=10)
6  x.dmat <- as.ddmatrix(x)
7
8  x.dmat
9
10 # Global on processor 0 --> distributed
11 if (comm.rank()==0){
12   y <- matrix(1:100, nrow=10, ncol=10)
13 } else {
14   y <- NULL
15 }
16 y.dmat <- as.ddmatrix(y)
17
18 y.dmat
19
20 finalize()
```

Execute this script via:

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

Example 2: ddmatrix Construction

Generate locally only what is needed

```
1  library(pbdDMAT, quiet=TRUE)
2  init.grid()
3
4  zero.dmat <- ddmatrix(0, nrow=100, ncol=100)
5  zero.dmat
6
7  id.dmat <- diag(1, nrow=100, ncol=100, type="ddmatrix")
8  id.dmat
9
10 comm.set.seed(diff=T)
11 rand.dmat <- ddmatrix("rnorm", nrow=100, ncol=100,
       mean=10, sd=100)
12 rand.dmat
13
14 finalize()
```

Execute this script via:

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

Example 3: ddmatrix Operations

Generate locally only what is needed

```
1  library(pbdDMAT, quiet=TRUE)
2  init.grid()
3
4  x.dmat <- ddmatrix(1:30, nrow=10)
5  y.dmat <- x.dmat[c(1, 3, 5, 7, 9), -3]
6
7  comm.print(y.dmat)
8  y <- as.matrix(y.dmat)
9  comm.print(y)
10
11 finalize()
```

Execute this script via:

```
1  mpirun -np 2 Rscript 3_extract.r
```

Example 4: More ddmatrix Operations

```
1  library(pbdDMAT, quiet=TRUE)
2  init.grid()
3
4  x.dmat <- ddmatrix(1:30, nrow=10)
5  y.dmat <- x.dmat + 1:7
6  z.dmat <- scale(x.dmat, center=TRUE, scale=TRUE)
7
8  y <- as.matrix(y.dmat)
9  z <- as.matrix(z.dmat)
10
11 comm.print(y)
12 comm.print(z)
13
14 finalize()
```

Execute this script via:

```
1  mpirun -np 2 Rscript 4_misc.r
```

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 `ddmatrix` data

Randomized SVD[1]

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

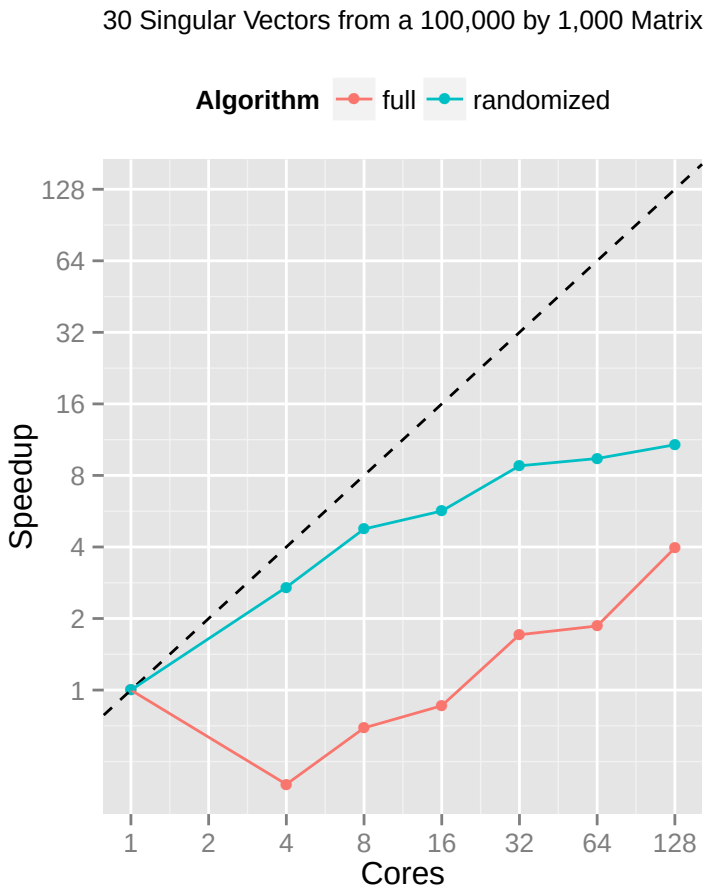
Serial R

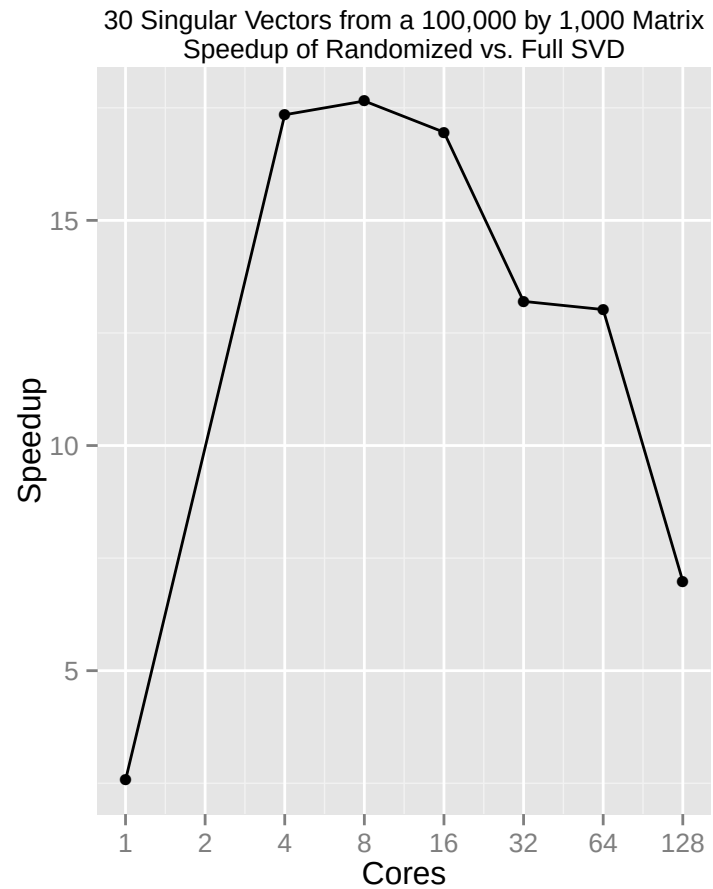
```
1  randSVD <- function(A, k, q=3)
2    {
3      ## Stage A
4      Omega <-  matrix(rnorm(n*2*k),
5              nrow=n, ncol=2*k)
6      Y <- A %*% Omega
7      Q <- qr.Q(qr(Y))
8      At <- t(A)
9      for(i in 1:q)
10       {
11         Y <- At %*% Q
12         Q <- qr.Q(qr(Y))
13         Y <- 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

Empirical Orthogonal Functions in Climate Analysis

- Computation and volume rendering of large-scale EOF coherent modes in rotating turbulent flow data, AGU Fall Meeting, December 2013

Principal Components Analysis

Coherent Modes in Turbulent Flow

Get and Redistribute the Data

```
1  library(pbdDMAT, quiet = TRUE)
2  init.grid()
3
4  ## load local data (file assumes 4 processors!)
5  g.dim <- c(64, 64, 1024)
6  my.dim <- g.dim / c(1, 1, comm.size())
7  save.file <- paste("xyz.RData", comm.rank(), sep="") #
       assumes 4 processors!
8  load(save.file)
9
10 ## reshape 3d array into a matrix for PCA (EOF)
       computation
11 ## first two dimensions become rows and third becomes
       columns
12 ## local reshape dimensions
13 my.nrow <- prod(my.dim[1:2])
14 my.ncol <- my.dim[3]
15 ldim <- c(my.nrow, my.ncol)
16
17 ## global reshape dimensions
18 g.nrow <- prod(g.dim[1:2])
19 g.ncol <- g.dim[3]
20 gdim <- c(g.nrow, g.ncol)
21
22 ## now reshape local
23 X <- matrix(vx, nrow=my.nrow, ncol=my.ncol, byrow=FALSE)
24 Y <- matrix(vy, nrow=my.nrow, ncol=my.ncol, byrow=FALSE)
25 Z <- matrix(vz, nrow=my.nrow, ncol=my.ncol, byrow=FALSE)
26
27 ## glue local pieces into a ddmatrix
28 X <- new("ddmatrix", Data=X, dim=gdim, ldim=ldim,
       bldim=ldim, ICTXT=1)
29 Y <- new("ddmatrix", Data=Y, dim=gdim, ldim=ldim,
       bldim=ldim, ICTXT=1)
30 Z <- new("ddmatrix", Data=Z, dim=gdim, ldim=ldim,
       bldim=ldim, ICTXT=1)
31
32 ## transform to 2d block cyclic
33 X <- redistribute(X, bldim=c(8,8), ICTXT=0)
34 Y <- redistribute(Y, bldim=c(8,8), ICTXT=0)
35 Z <- redistribute(Z, bldim=c(8,8), ICTXT=0)
```

Coherent Modes in Turbulent Flow

Compute PCA and do Scree Plot (0_pca.r)

```
1  E <- sqrt(X^2 + Y^2 + Z^2) # energy from velocity
2  E.pca <- prcomp(x=E, retx=TRUE, scale=FALSE)
3
4  # plot using one process
5  if(comm.rank() == 0)
6    {
7      ## scree plot for first 50 components
8      variance <- E.pca$sdev^2  # note: all own sdev
9      proportion <- variance[1:50]/sum(variance)
10     cumulative <- cumsum(proportion)
11     component <- 1:length(proportion)
12     png("scree.png")
13     plot(component, cumulative, ylim=c(0,1))
14     points(component, proportion, type="h", col="blue")
15     dev.off()
16   }
17
18 finalize()
```

How can we plot in parallel?

- Several plots, one or more on each processor (can do now)
- One plot by several processors (need to rewrite graphics)

Plots in parallel

png.slice

```
1  png.slice <- function(x, g.dim, lab="slice", title=lab,
       work.dir="", zero.center=TRUE, most.positive=TRUE)
2  {
3    X <- array(as.vector(x), dim=g.dim)
4
5    ## prepare zero centered topo.colors
6    if(zero.center)
7      {
8        . . .
9      }
10   else
11     zlim <- range(X)
12
13   ## set most positive (for unique up to sign)
14   if(most.positive)
15     {
16       . . .
17     }
18
19   ## make png file
20   file <- paste(work.dir, lab, "-r", comm.rank(),
         ".png", sep="")
21   png(file)
22   image(x=1:g.dim[1], y=1:g.dim[2], z=X,
         col=topo.colors(40), useRaster=TRUE, asp=1,
         xlim=c(1, g.dim[1] + 1), ylim=c(1, g.dim[2] + 1),
         zlim=zlim)
23   title(title)
24   ret <- dev.off()
25   invisible(ret)
26 }
```

Parallel Plot Ensembles

Plots in parallel

Get and Redistribute the Data

```
1  library(pbdDMAT, quiet = TRUE)
2  init.grid()
3
4  ## set global and local dimensions
5  g.dim <- c(64, 64, 1024)
6  my.dim <- g.dim / c(1, 1, comm.size())
7
8  save.file <- paste("xyz.RData", comm.rank(), sep="")
9  load(save.file)    # gets vx vector
10
11 ## reshape 3d array into a matrix
12 ## first two dimensions become rows and third becomes
       columns
13
14 ## local reshape dimensions
15 my.nrow <- prod(my.dim[1:2])
16 my.ncol <- my.dim[3]
17 ldim <- c(my.nrow, my.ncol)
18
19 ## global reshape dimensions
20 g.nrow <- prod(g.dim[1:2])
21 g.ncol <- g.dim[3]
22 gdim <- c(g.nrow, g.ncol)
23
24 ## now reshape local
25 X <- matrix(vx, nrow=my.nrow, ncol=my.ncol, byrow=FALSE)
26
27 ## glue local pieces into a ddmatrix
28 X <- new("ddmatrix", Data=X, dim=gdim, ldim=ldim,
       bldim=ldim, ICTXT=1)
29
30 ## transform to 2d block cyclic
31 X <- redistribute(X, bldim=c(8,8), ICTXT=0)
32
33 ## Plot few columns in parallel
34    . . .
35 finalize()
```

Plots in parallel

Make comm.size() plots in parallel

```
1  step <- 5
2  max.plots <- min(20, ncol(X) %/% step)
3  last.plot <- 1 - step
4  time <- comm.timer(
5  for(i in 1:max.plots)
6      {
7          now.plots <- last.plot + step*(1:comm.size())
8          my.col <- gather.col(X[, now.plots])
9          lab <- paste("col", lead0(now.plots[comm.rank()
               + 1]), sep="")
10         png.slice(my.col, g.dim[1:2], lab)
11         last.plot <- now.plots[length(now.plots)]
12     }
13 )
```

Plots in parallel

gather.col First Attempt (1_plot.r)

```
1 gather.col <- function(x, num=min(ncol(x), comm.size()))
2 {
3     ## gather complete columns of a global array to
          different ranks
4     my.local <- as.vector(x[, comm.rank() + 1],
          proc.dest=comm.rank())
5     my.local
6 }
```

Plots in parallel

gather.col Second Attempt (2_plot.r)

```
1  gather.col <- function(x, num=min(ncol(x), comm.size()))
2  {
3    ## gather complete columns of a global array to
         different ranks
4    my.local <- NULL
5    for(i in 1:num)
6      {
7        ## serial collection of unique data to each rank
8        local <- as.vector(x[, i])
9        if(comm.rank() + 1 == i) my.local <- local
10     }
11   my.local
12 }
```

Plots in parallel

gather.col The Right Way (3_plot.r)

```
1  gather.col <- function(x, num=min(ncol(x), comm.size()))
2  {
3    ## gather complete columns of a global array to
         different ranks
4    x.num <- x[, 1:num]
5    x.num <- as.colblock(x.num)
6
7    ## ScaLAPACK fix (a future release will automate)
8    if(ownany(x.num))
9      ret <- as.vector(submatrix(x.num))
10   else
11     ret <- NULL
12   ret
13 }
```

Plots in parallel

Now Plot the PCA Components (4_plot.r)

```
1  E <- sqrt(X^2 + Y^2 + Z^2)
2
3  E.pca <- prcomp(x=E, retx=TRUE, scale=FALSE)
4
5  ## Use ranks 1 to n.pca to plot individual components in
       parallel
6  n.pca <- min(comm.size(), g.nrow)
7  my.col <- gather.col(E.pca$x, num=n.pca)
8
9  if(!is.null(my.col))
10   {
11     ## component plots on rank 1 to n.pca
12     lab <- paste("pc", comm.rank(), sep="")
13     title <- paste(lab, "sigma^2 =",
           variance[comm.rank() + 1])
14     png.slice(my.col, g.dim[1:2], lab, title=title,
           work.dir=work.dir)
15   }
```

Exercise: scripts/pbdDMAT/dmat_app

- Experiment with scripts 0_pca.r, 1_plot.r, 2_plot.r, 3_plot.r, 4_plot.r

Simple Redistributions

- as.block(dx, square.bldim = TRUE)
- as.rowblock(dx)
- as.colblock(dx)
- as.rowcyclic(dx, bldim = .BLDIM)
- as.colcyclic(dx, bldim = .BLDIM)
- as.blockcyclic(dx, bldim = .BLDIM)

BLACS context (Processor Grid)

- init.grid(P,Q)
- .ICTXT = 0 gives P x Q
- .ICTXT = 1 gives PQ x 1
- .ICTXT = 2 gives 1 x PQ

Exercise: scripts/pbdDMAT/dmat_app

- Experiment with scripts 5ictxt.r, 6_ictxt.r, and 7_ictxt.r
- Experiment with other redistributions

Contents

The pbdR Project

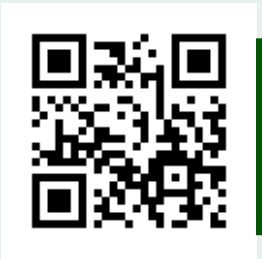
- Our website: http://r-pbd.org/
- Email us at: RBigData@gmail.com
- Our google group: http://group.r-pbd.org/

Where to begin?

- The **pbdDEMO** package
  http://cran.r-project.org/web/packages/pbdDEMO/
- The **pbdDEMO** Vignette: http://goo.gl/HZkRt

Thanks for coming!

Questions?



pbdR

Programming with Big Data in R

http://r-pbd.org/