pbdR: R for HPC and Big Data Analytics

George Ostrouchov
Oak Ridge National Laboratory and University of Tennessee
USA

PRACE Winter School 2015, January 12-15
VŠB-Technical University of Ostrava, Czech Republic

The pbdR Core Team

Wei-Chen Chen[1]
George Ostrouchov[2,3]
Pragneshkumar Patel[3]
Drew Schmidt[3]

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 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]FDA
Washington, DC, USA

[2]Computer Science and Mathematics Division
Oak Ridge National Laboratory, Oak Ridge TN, USA

[3]Joint Institute for Computational Sciences
University of Tennessee, Knoxville TN, USA

About This Presentation

Downloads

This presentation is available at: http://r-pbd.org/tutorial

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.

Contents

OAK RIDGE National Laboratory

Contents

What is R?

- *lingua franca* for data analytics and statistical computing.
- Part programming language, part data analysis package.
- Dialect of S (May 5, 1976, Bell Labs).
- Free (GPL ≥ 2)
- A C program (mostly): 52% C, 26% Fortran, 22% R
- Highly extensible; has over 6000 user-contributed packages.

Who uses R?


ORACLE
MERCK
KICKSTARTER
JOHN DEERE
ebay
Bank of America
The New York Times
NIST
mozilla FOUNDATION
ORBITZ
cfpb
Shell
Pfizer
FDA
NOVARTIS
LLOYD'S
bing
Google
Ford
Zillow
okcupid

The R Language

- R is slow; if you don't know what you're doing, it's *really* slow.
- High-level scripting language.
- Syntax designed for data: models are first-class constructs, missingness is built into the core of the language, ...

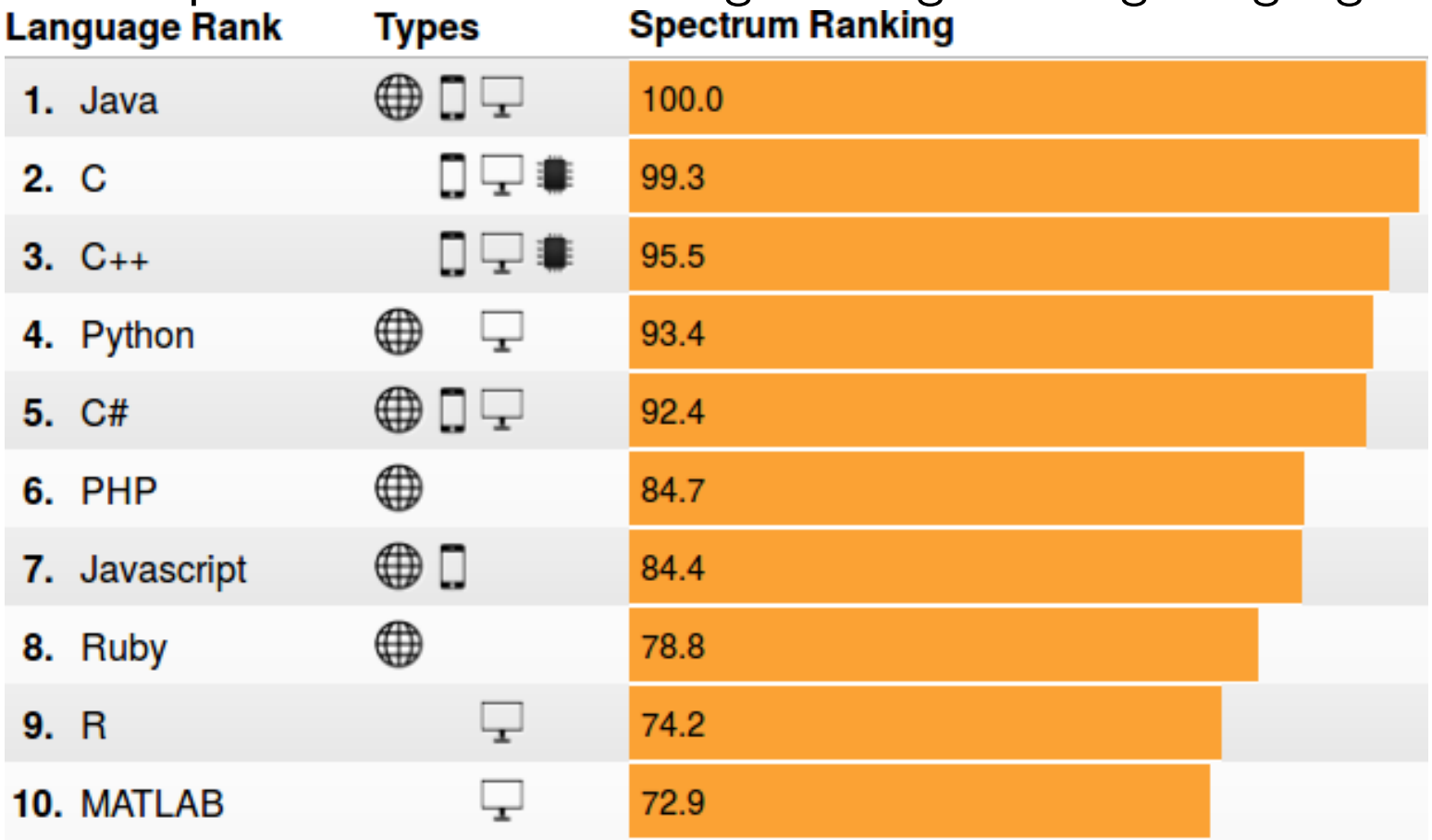
The R Language

If you come from CS, you may find R a bit strange:

- Storage: logical, int, double, double complex, character (strings)
- Structures: vector, matrix, array, list, dataframe
- Caveats: (Logical) `TRUE`, `FALSE`, `NA`
- In fact, there's an `NA` for each type:
  - Integer: $-(2^{-31}-1)$
  - Double: value at address `0x7FF00000000007A2LL`
- 3 official OOP systems (none of them work like anything you're used to), several unofficial ones.
- No official support for C++; several unofficial packages supporting C++.

But you can't deny its popularity!

IEEE Spectrum's 2014 Ranking of Programming Languages



See:

http://spectrum.ieee.org/static/interactive-the-top-programming-languages#index

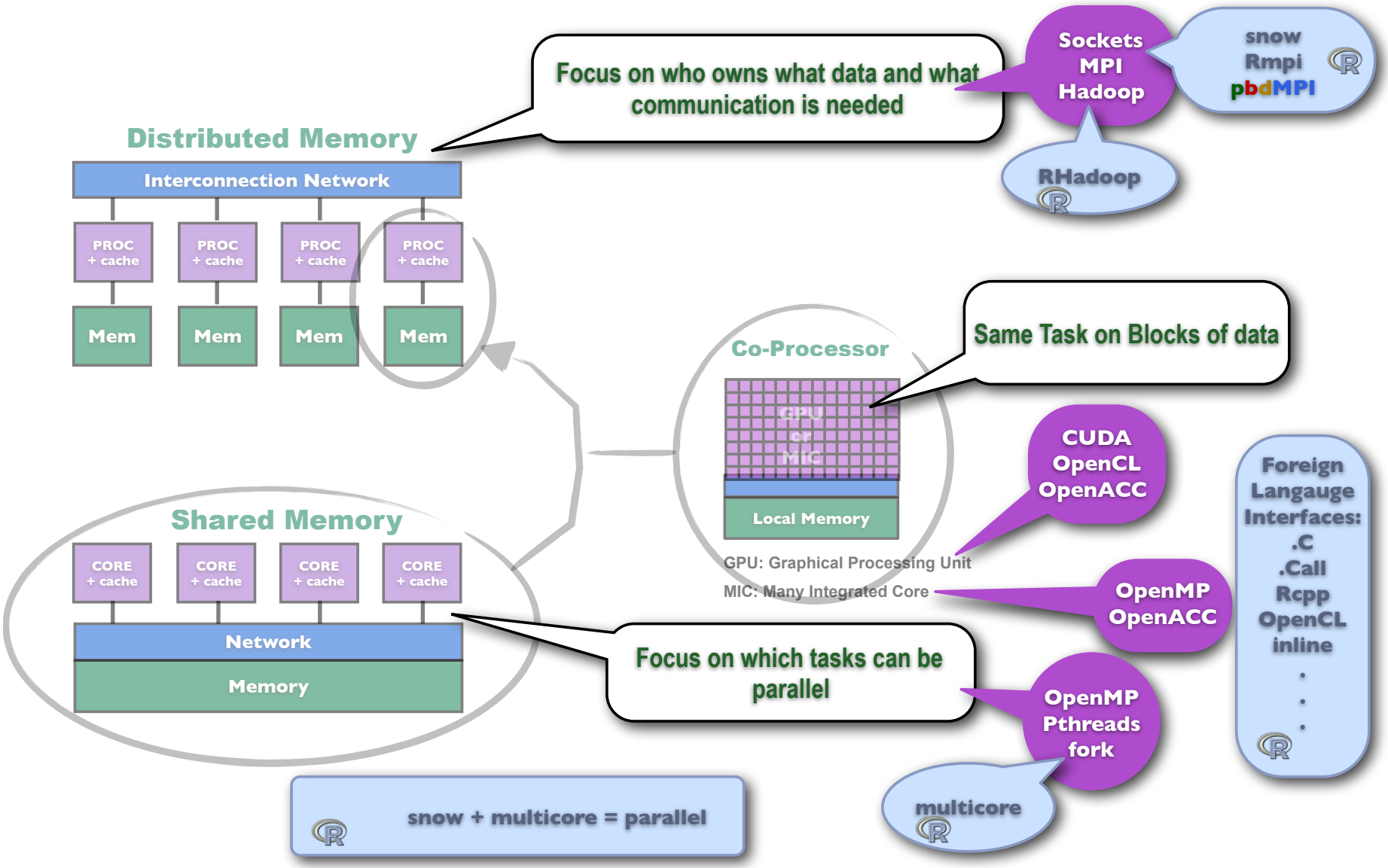
Resources for Learning R

- *Advanced R*: http://adv-r.had.co.nz/ and *ggplot2* http://docs.ggplot2.org/current/ by Hadley Wickham
- *The Art of R Programming* by Norm Matloff: http://nostarch.com/artofr.htm
- *An Introduction to R* by Venables, Smith, and the R Core Team: http://cran.r-project.org/doc/manuals/R-intro.pdf
- *The R Inferno* by Patrick Burns: http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
- Mathesaurus: http://mathesaurus.sourceforge.net/
- R programming for those coming from other languages: http://www.johndcook.com/R_language_for_programmers.html
- *aRrgh: a newcomer's (angry) guide to R*, by Tim Smith and Kevin Ushey: http://tim-smith.us/arrgh/

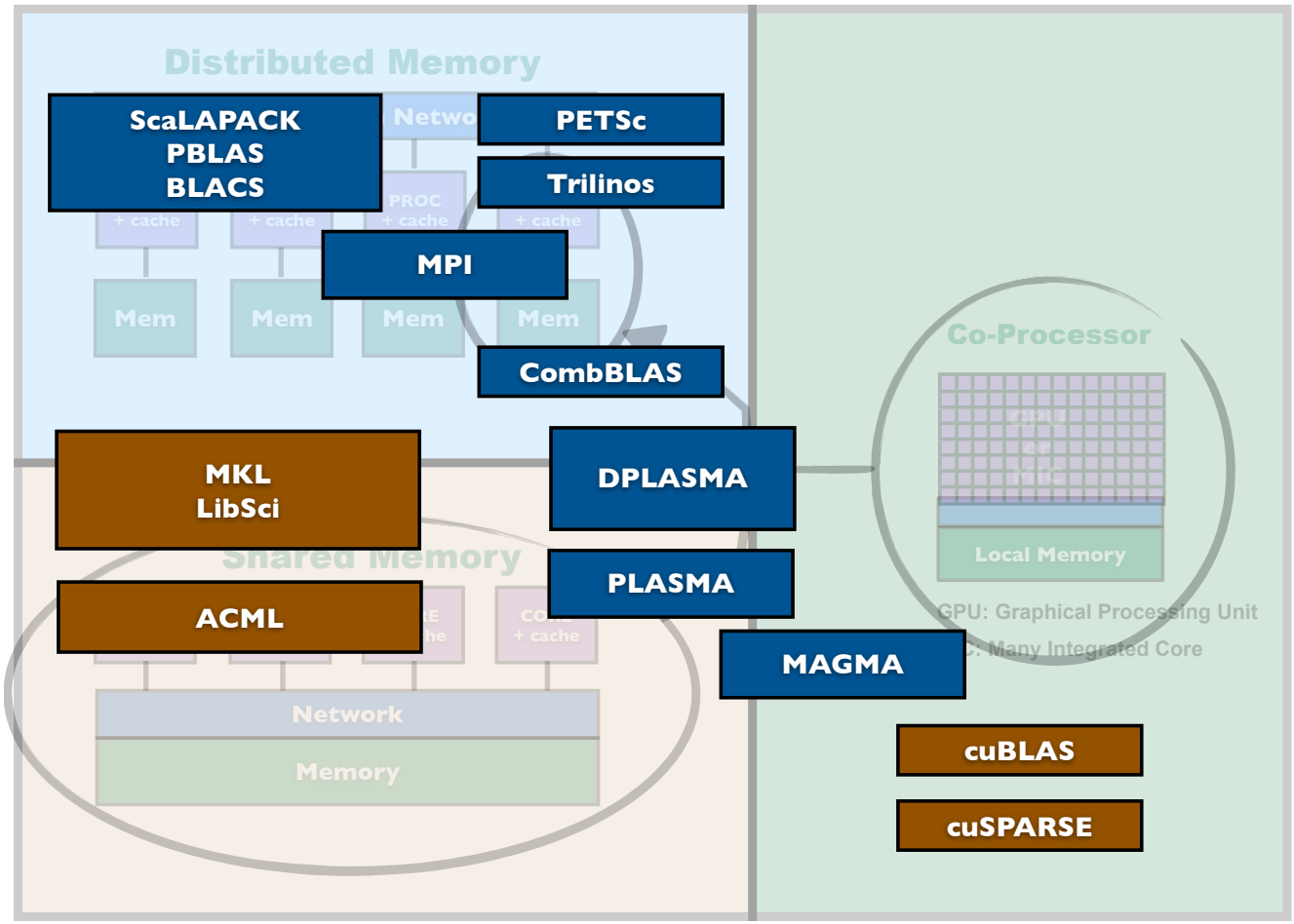
Other Invaluable Resources

- *R Installation and Administration*: http://cran.r-project.org/doc/manuals/R-admin.html
- *Task Views*: http://cran.at.r-project.org/web/views
- *Writing R Extensions*: http://cran.r-project.org/doc/manuals/R-exts.html
- Mailing list archives: http://tolstoy.newcastle.edu.au/R/
- The [R] stackoverflow tag.

R Interfaces to Low-Level Native Tools



HPC Libraries: 30+ Years of Research



Tau

mpiP

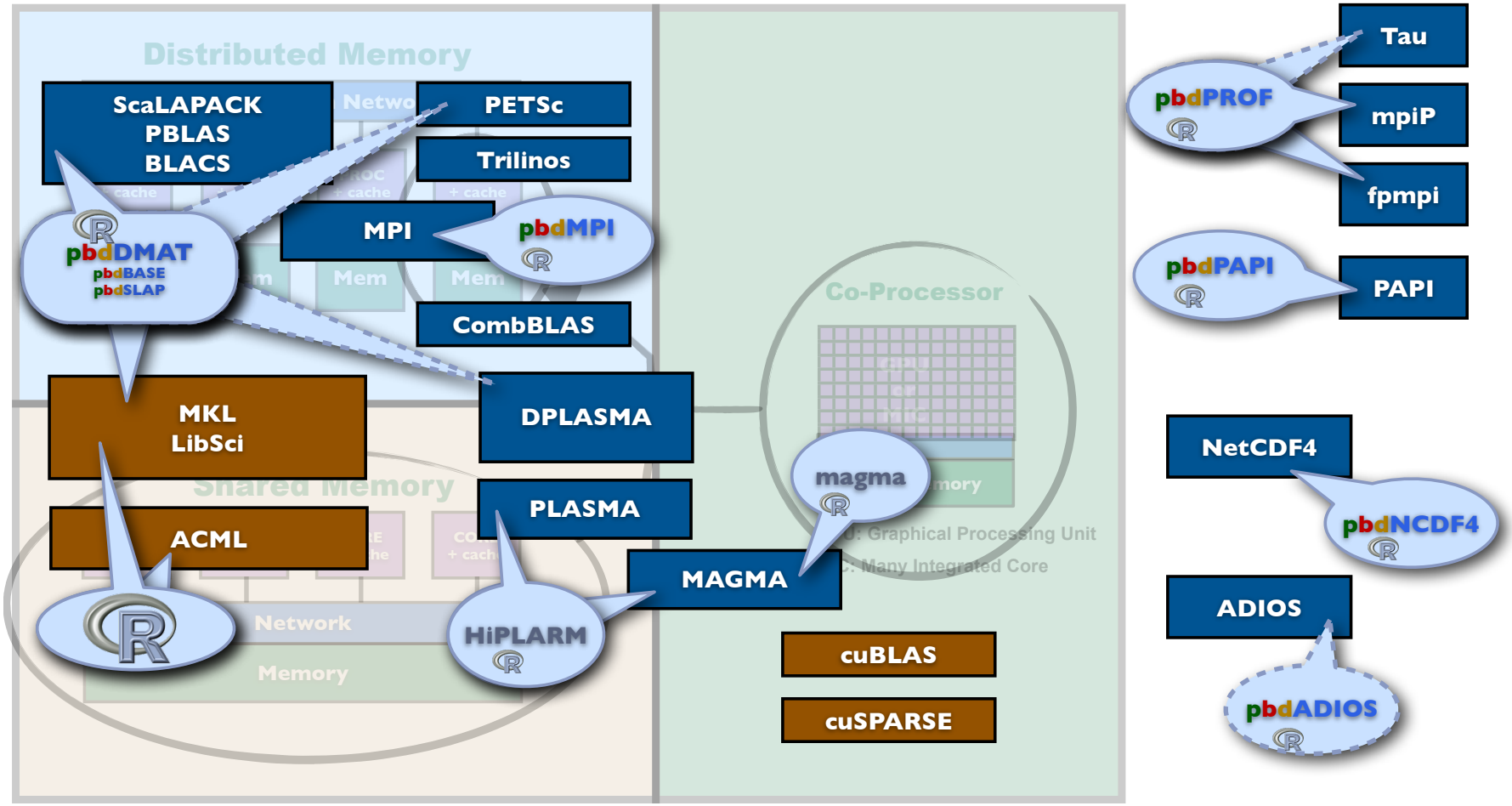
fpmpi

PAPI

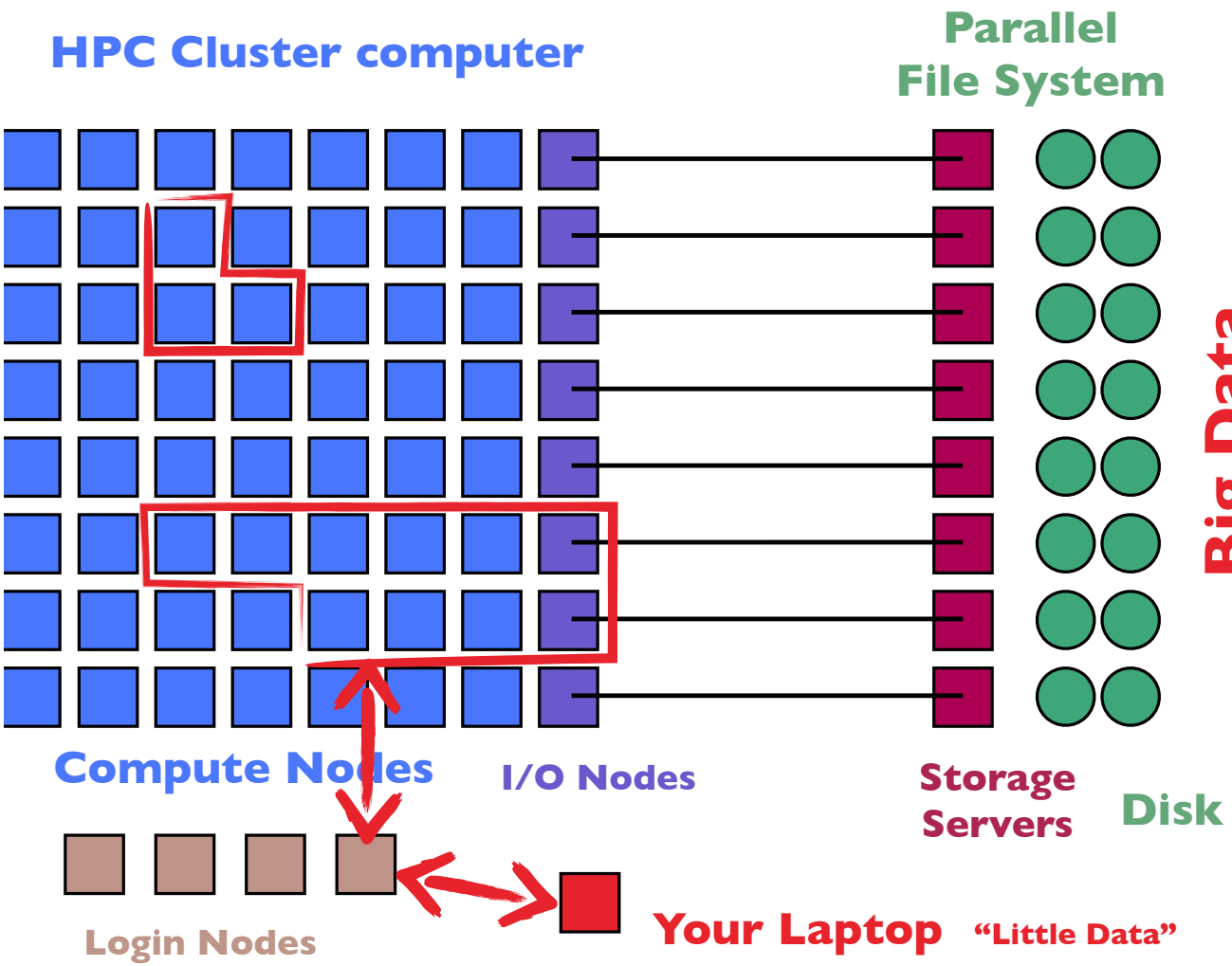
NetCDF4

ADIOS

R and pbdR R Interfaces to HPC Libraries



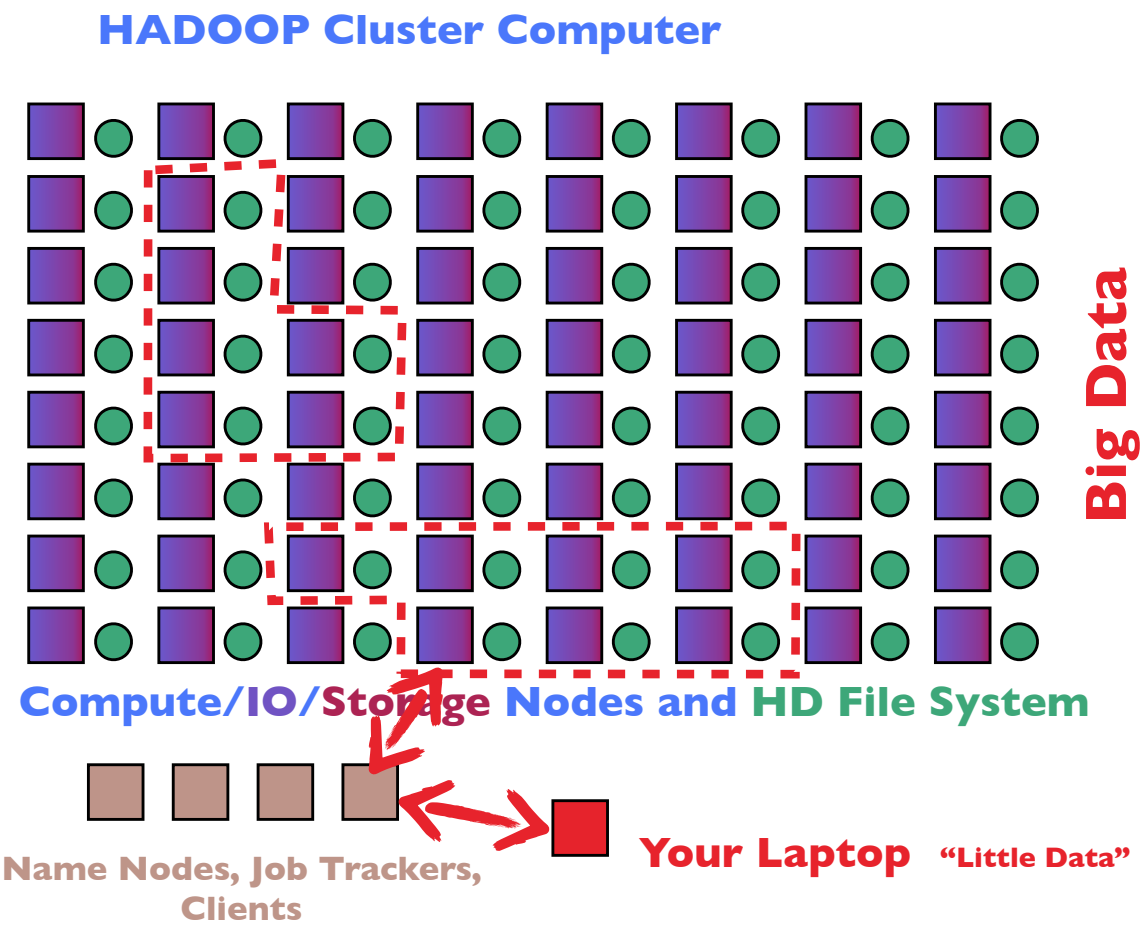
Big Data and Little Data



Analysis Components

Reader or Generator
Analysis script
Display/write results

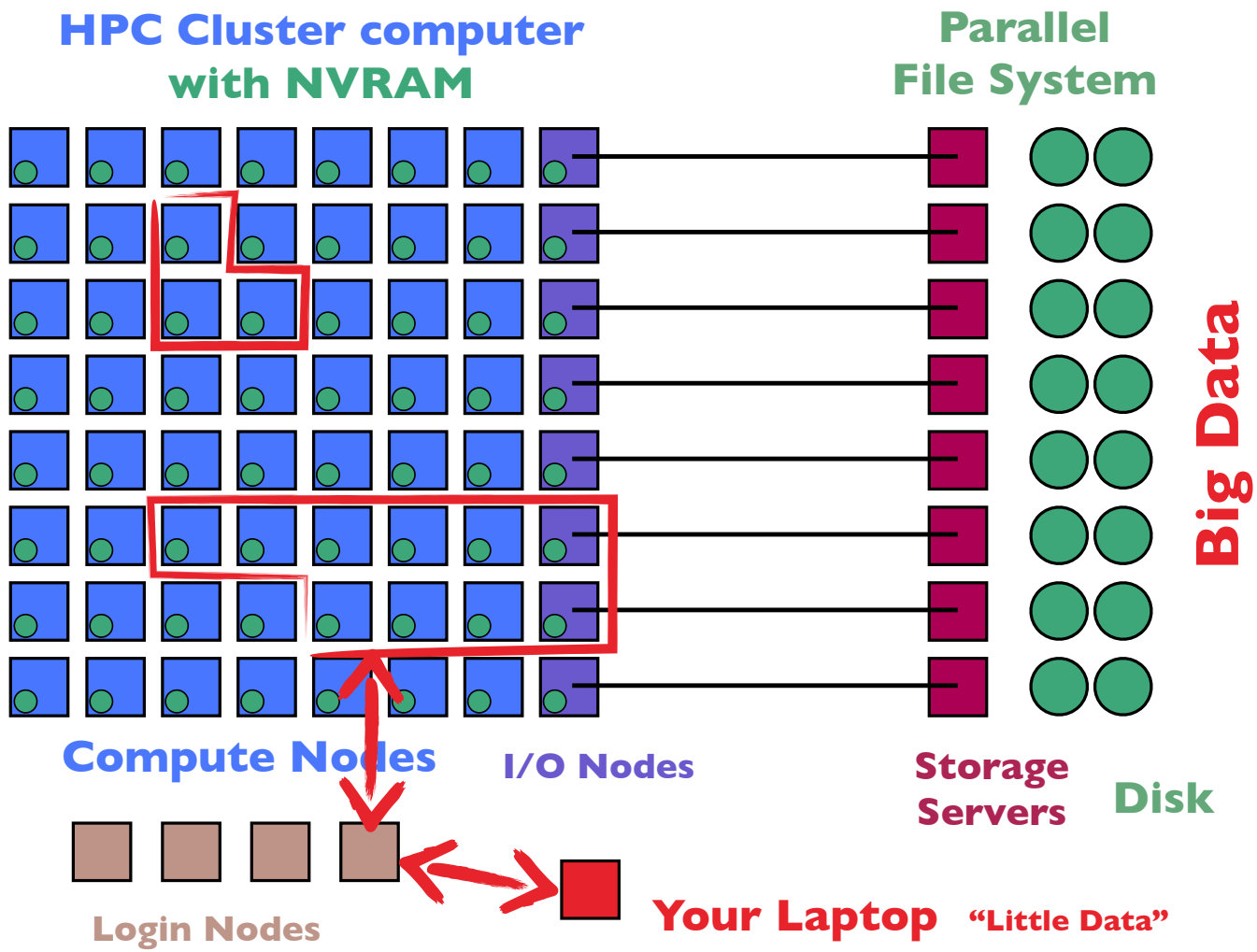
Hadoop: File System or HPC Cluster?



Components

HDFS file system
Yarn resource manager
Map-reduce limitation

Adding NVRAM to New HPC Systems



Summary

- R is the high-level "language" for data science
- Three flavors of hardware
  - Distributed is stable
  - Multicore and co-processor are evolving
  - Two memory models
  - Distributed works in multicore, but not the other way around
- Parallelism hierarchy
- Medium to big machines have all three
- HPC Libraries can span hierarchy

Contents

Timings

Getting simple timings as a basic measure of performance is easy, and valuable.

- `system.time()` — timing blocks of code.
- `Rprof()` — timing execution of R functions.
- `Rprofmem()` — reporting memory allocation in R .
- `tracemem()` — detect when a copy of an R object is created.
- The **rbenchmark** package — Benchmark comparisons.

Performance Profiling Tools: system.time()

system.time() is a basic R utility for timing expressions

```
1  x <- matrix(rnorm(20000*750), nrow=20000, ncol=750)
2
3  system.time(t(x) %*% x)
4  #     user   system  elapsed
5  #    2.187    0.032    2.324
6
7  system.time(crossprod(x))
8  #     user   system  elapsed
9  #    1.009    0.003    1.019
10
11 system.time(cov(x))
12 #     user   system  elapsed
13 #    6.264    0.026    6.338
```

Performance Profiling Tools: Rprof()

Rprof() times the execution of all R functions:

```
Rprof(filename="Rprof.out", append=FALSE, interval=0.02,
  memory.profiling=FALSE, gc.profiling=FALSE,
  line.profiling=FALSE, numfiles=100L, bufsize=10000L)
```

```
1  x <- matrix(rnorm(10000*250), nrow=10000, ncol=250)
2
3  Rprof()
4  invisible(prcomp(x))
5  Rprof(NULL)
6
7  summaryRprof()
8
9  Rprof(interval=.99)
10 invisible(prcomp(x))
11 Rprof(NULL)
12
13 summaryRprof()
```

Performance Profiling Tools: Rprof()

```
$by.self
                self.time self.pct total.time total.pct
"La.svd"             0.68    69.39       0.72     73.47
"%*%"                0.12    12.24       0.12     12.24
"aperm.default"      0.04     4.08       0.04      4.08
"array"              0.04     4.08       0.04      4.08
"matrix"             0.04     4.08       0.04      4.08
"sweep"              0.02     2.04       0.10     10.20
### output truncated by presenter

$by.total
                 total.time total.pct self.time self.pct
"prcomp"               0.98    100.00       0.00      0.00
"prcomp.default"       0.98    100.00       0.00      0.00
"svd"                  0.76     77.55       0.00      0.00
"La.svd"               0.72     73.47       0.68     69.39
### output truncated by presenter

$sample.interval
[1] 0.02

$sampling.time
[1] 0.98
```

Performance Profiling Tools: Rprof()

```
$by.self
[1] self.time  self.pct   total.time total.pct
<0 rows> (or 0-length row.names)

$by.total
[1] total.time total.pct  self.time  self.pct
<0 rows> (or 0-length row.names)

$sample.interval
[1] 0.99

$sampling.time
[1] 0
```

Performance Profiling Tools: rbenchmark

rbenchmark is a simple package that easily benchmarks different functions:

```
1  x <- matrix(rnorm(10000*500), nrow=10000, ncol=500)
2
3  f <- function(x) t(x) %*% x
4  g <- function(x) crossprod(x)
5
6  library(rbenchmark)
7  benchmark(f(x), g(x))
8
9  #   test replications elapsed relative
10 # 1 f(x)          100  64.153    2.063
11 # 2 g(x)          100  31.098    1.000
```

Other Profiling Tools

- perf
- PAPI
- MPI profiling: fpmpi, mpiP, TAU

Profiling MPI Codes with **pbdPROF**

1. Rebuild pbdR packages

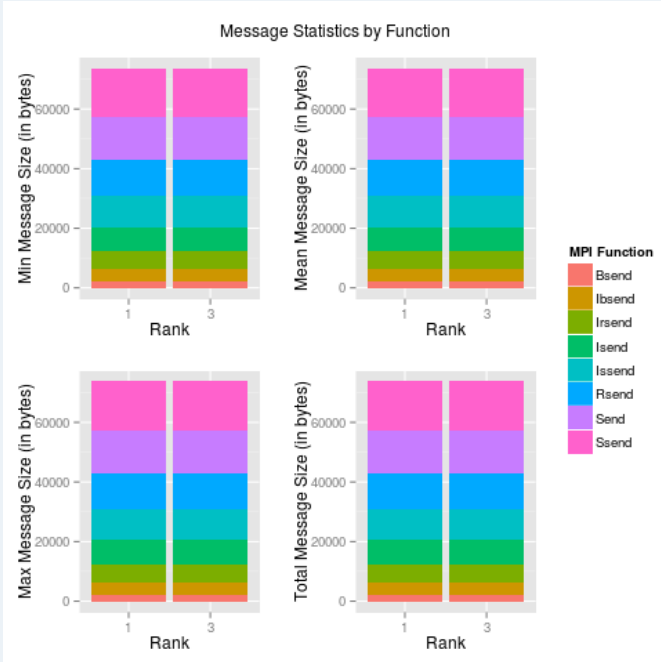
```
R CMD INSTALL pbdMPI_0.2-1.tar.gz \
    --configure-args= \
    "--enable-pbdPROF"
```

2. Run code

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

3. Analyze results

```
1 library(pbdPROF)
2 prof <- read.prof( "output.mpiP")
3 plot(prof, plot.type="messages2")
```



Profiling with **pbdPAPI**

- Bindings for Performance Application Programming Interface (PAPI)
- Gathers detailed hardware counter data.
- High and low level interfaces

Function	Description of Measurement
`system.flips()`	Time, floating point instructions, and Mflips
`system.flops()`	Time, floating point operations, and Mflops
`system.cache()`	Cache misses, hits, accesses, and reads
`system.epc()`	Events per cycle
`system.idle()`	Idle cycles
`system.cpuormem()`	CPU or RAM bound*
`system.utilization()`	CPU utilization*

Summary

- *Profile, profile, profile.*
- Use `system.time()` to get a general sense of a method.
- Use **rbenchmark**'s `benchmark()` to compare 2 methods.
- Use `Rprof()` for more detailed profiling.
- Other tools exist for more hardcore applications (**pbdPAPI** and **pbdPROF**).

Contents

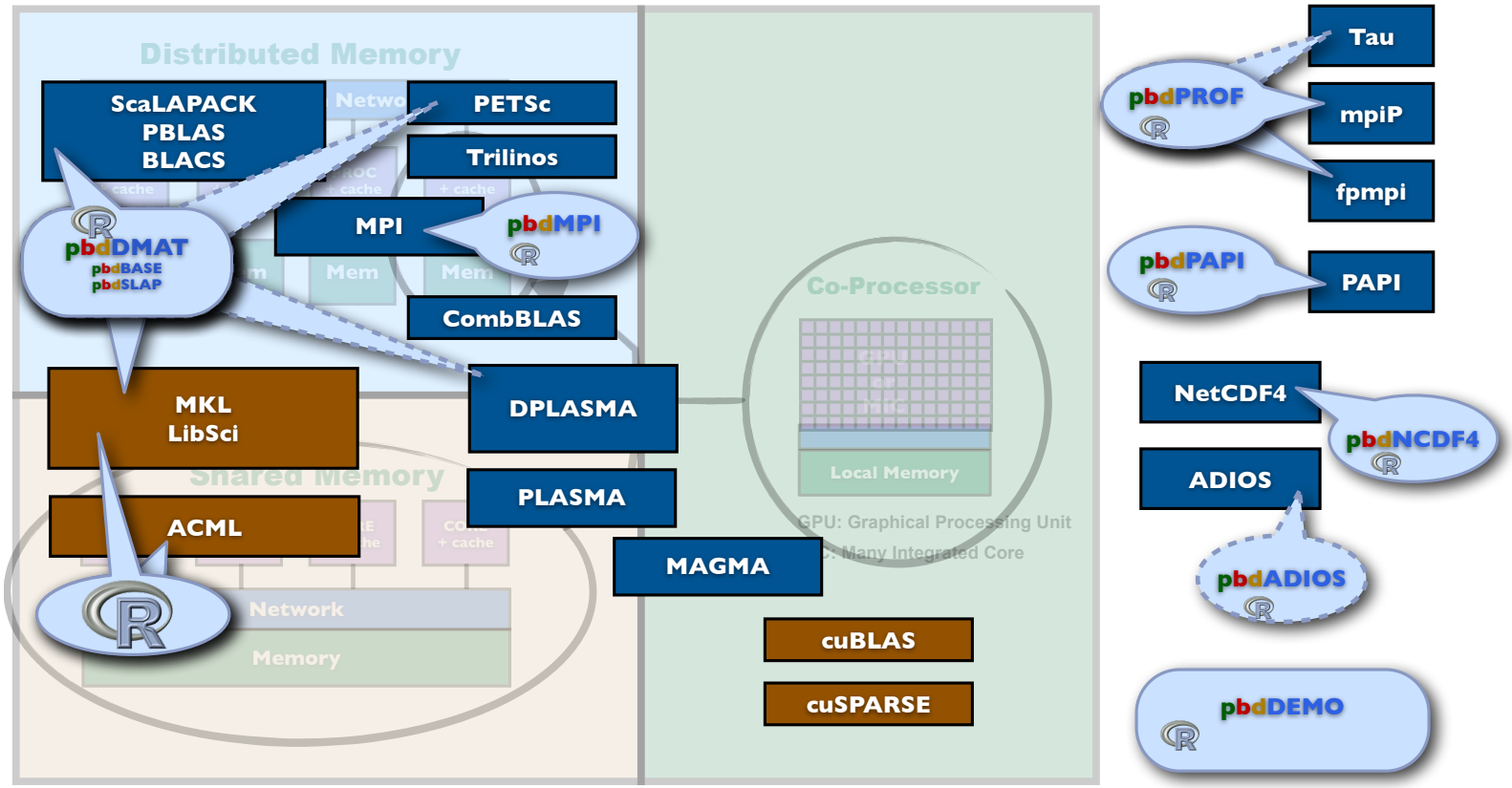
Programming with Big Data in R (pbdR)

Striving for *Productivity, Portability, Performance*

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

[a] MPL, BSD, and GPL licensed

pbdR Interfaces to Libraries: Sustainable Path



Why use HPC libraries?

- The HPC community has been at this for decades.
- *They're tested. They're fast. They're scalable.*
- Many science communities are invested in their API.
- You're not going to beat Jack Dongarra's lab at dense linear algebra!

Simple Interface for MPI Operations with **pbdMPI**

Rmpi

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

pbdMPI

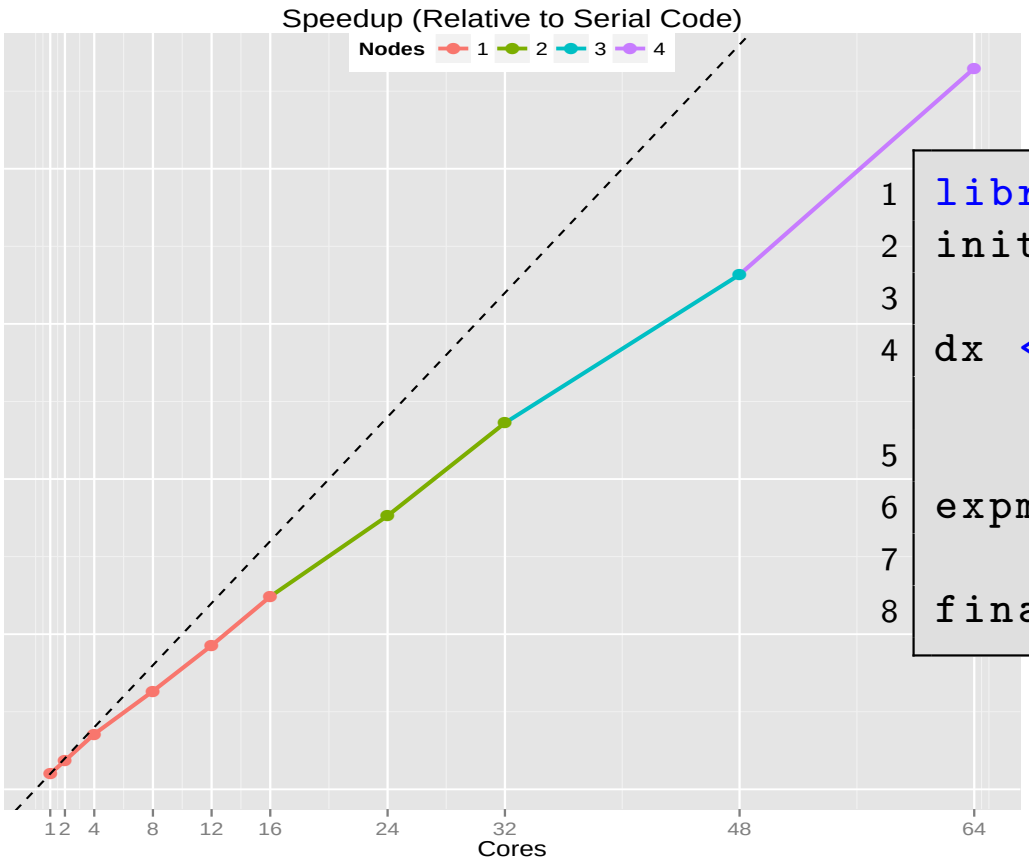
```
1 allreduce(x)
```

Types in R

```
1 > is.integer(1)
2 [1] FALSE
3 > is.integer(2)
4 [1] FALSE
5 > is.integer(1:2)
6 [1] TRUE
```

Distributed Matrices and Statistics with **pbdDMAT**

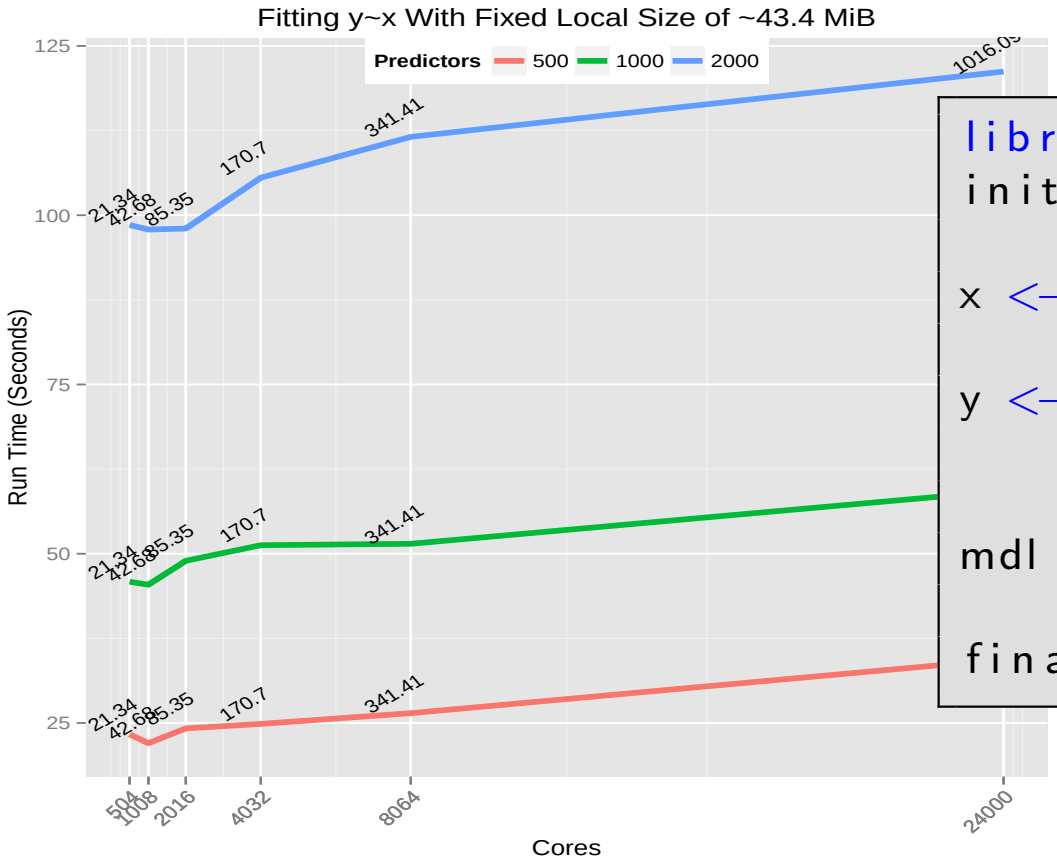
Matrix Exponentiation




```
1 library(pbdDMAT)
2 init.grid()
3
4 dx <- ddmatrix("rnorm",
      5000, 5000)
5
6 expm(dx)
7
8 finalize()
```

Distributed Matrices and Statistics with **pbdDMAT**

Least Squares Benchmark



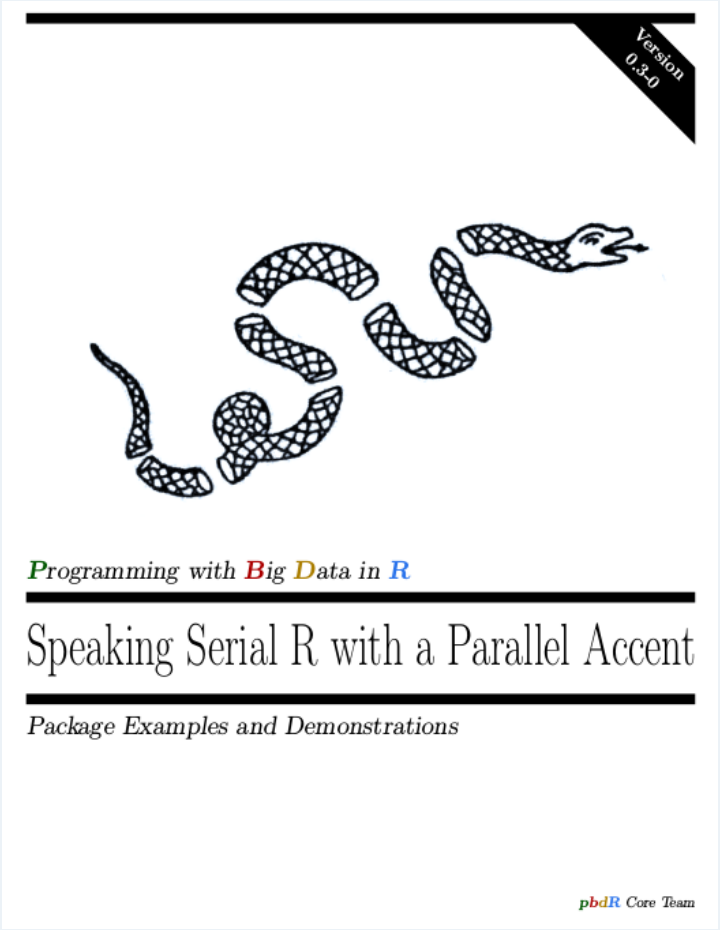

```
library(pbdDMAT)
init.grid()

x <- ddmatrix("rnorm",
    nrow=m, ncol=n)
y <- ddmatrix("rnorm",
    nrow=m, ncol=1)

mdl <- lm.fit(x=x, y=y)

finalize()
```

Getting Started with HPC for R Users: **pbdDEMO**



- 140 page, textbook-style vignette.
- Over 30 demos, utilizing all* packages.
- Not just a "hello world"!
- Demos include:
  - PCA
  - Regression
  - Parallel data input
  - Model-based clustering
  - Simple Monte Carlo simulation
  - Bayesian MCMC

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
OOP, Functional, SPMD, ...

SIMD

Hardware instructions
MMX, SSE, ...

Single Program/Multiple Data (SPMD)

SPMD is arguably the simplest extension of serial programming.

- Only one program is written, executed in batch on all processors.
- Different processors are autonomous; there is no manager.
- Dominant programming model for large machines for 30 years.

Summary

- pbdR connects R to scalable HPC libraries.
- The **pbdDEMO** package offers numerous examples and explanations for getting started with distributed R programming.
- pbdR programs are R programs.

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
```

Random Seeds

pbdMPI offers a simple interface for managing random seeds:

- `comm.set.seed(seed=1234, diff=TRUE)` — All processors generate different streams.
- `comm.set.seed(seed=1234, diff=FALSE)` — All processors generate same streams.

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()`

Summary

- Start by loading the package:

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

- 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

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

Distributing by Row: Load Balanced

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

Processors = 0 1 2 3 4 5

Distributing by Row: Local View

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

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

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

$$\left[\begin{array}{ccccccccc} x_{71} & x_{72} & x_{73} & x_{74} & x_{75} & x_{76} & x_{77} & x_{78} & x_{79} \end{array}\right]_{1\times 9}$$

$$\left[\begin{array}{ccccccccc} x_{81} & x_{82} & x_{83} & x_{84} & x_{85} & x_{86} & x_{87} & x_{88} & x_{89} \end{array}\right]_{1\times 9}$$

$$\left[\begin{array}{ccccccccc} x_{91} & x_{92} & x_{93} & x_{94} & x_{95} & x_{96} & x_{97} & x_{98} & x_{99} \end{array}\right]_{1\times 9}$$

Processors = 0 1 2 3 4 5

Distributing by Row: Non-Balanced

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

Processors = 0 1 2 3 4 5

Distributing by Row: 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

5 Distributing Data

- A Way to Distribute Your Data
- Summary

Summary

- Need to distribute your data? Try splitting by row.
- May not work well if your data is square (or wider than tall).

Contents

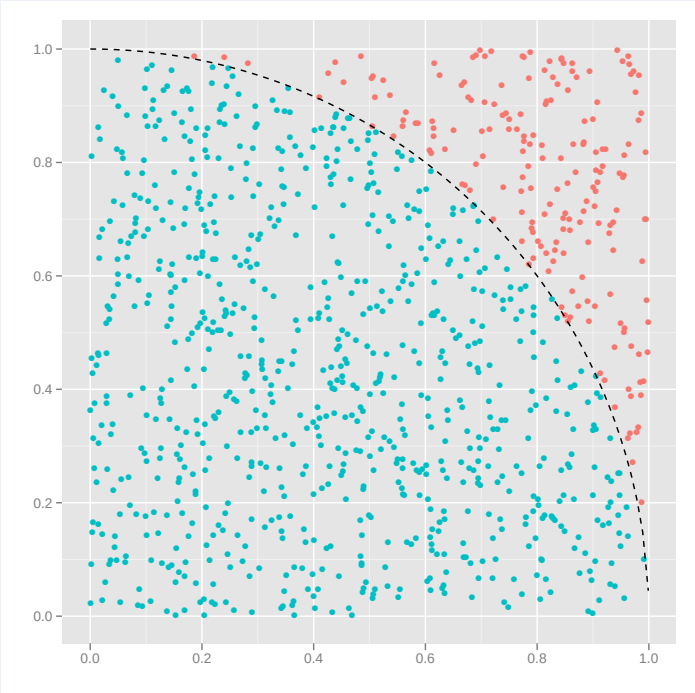
6 Basic Statistics Examples

- pbdMPI Example: Monte Carlo Simulation
- pbdMPI Example: Sample Covariance
- pbdMPI Example: Linear Regression
- Summary

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 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(seed=1234567, diff=TRUE)
4
5  N.loc <- 50000 / comm.size()
6  X.loc <- matrix(runif(N.loc * 2), ncol = 2)
7  r.loc <- sum(rowSums(X.loc^2) <= 1)
8  r <- allreduce(r.loc)
9  PI <- 4*r/(N.loc * comm.size())
10 comm.print(PI)
11
12 finalize()
```

Note

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

6 Basic Statistics Examples

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

6 Basic Statistics Examples

- pbdMPI Example: Monte Carlo Simulation
- pbdMPI Example: Sample Covariance
- pbdMPI Example: Linear Regression
- Summary

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

6 Basic Statistics Examples

- pbdMPI Example: Monte Carlo Simulation
- pbdMPI Example: Sample Covariance
- pbdMPI Example: Linear Regression
- Summary

Summary

- SPMD programming is (often) a natural extension of serial programming.
- More **pbdMPI** examples in **pbdDEMO**.

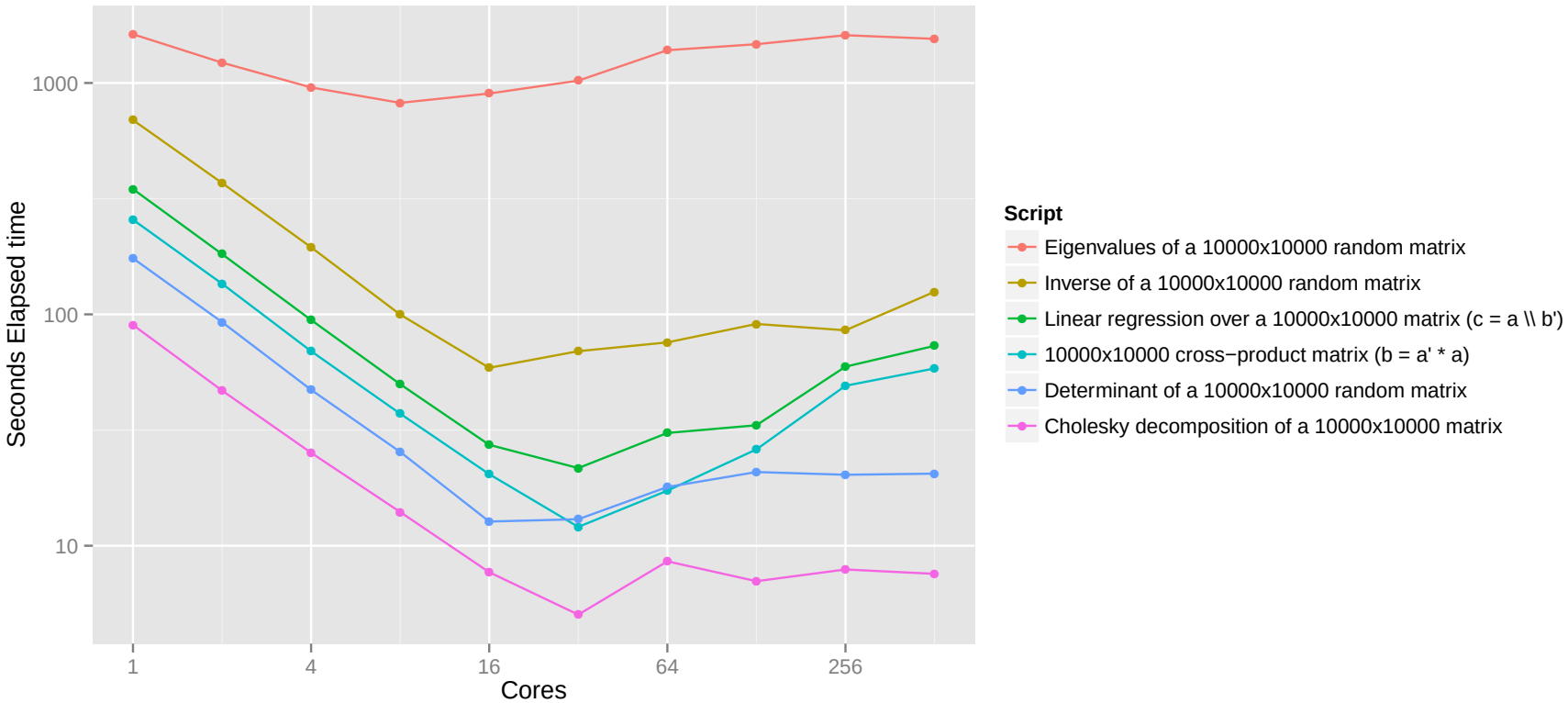
Contents

7 Introduction to pbdDMAT and the ddmatrix Structure

- Introduction to Distributed Matrices
- pbdDMAT
- Summary

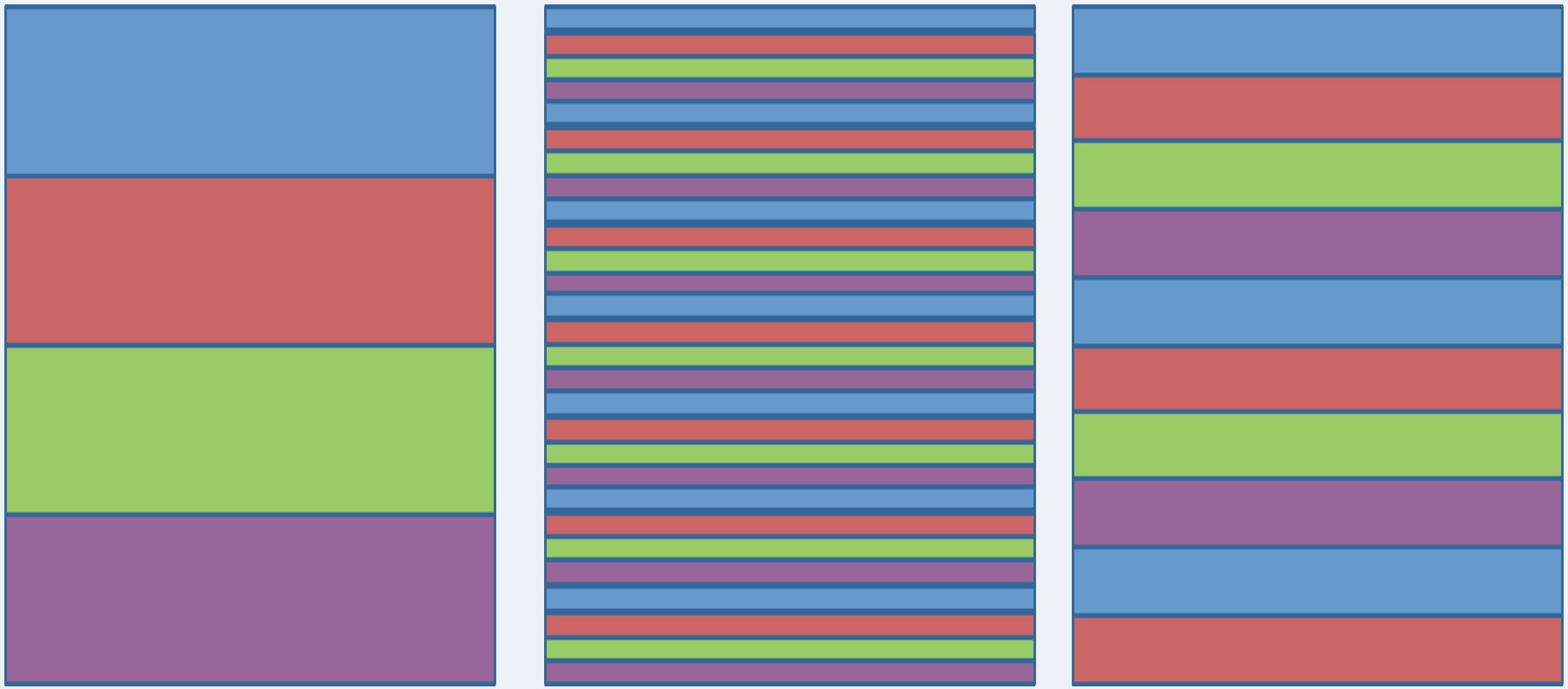
Distributed Matrices

You can only get so far with one node. . .



The solution is to distribute the data.

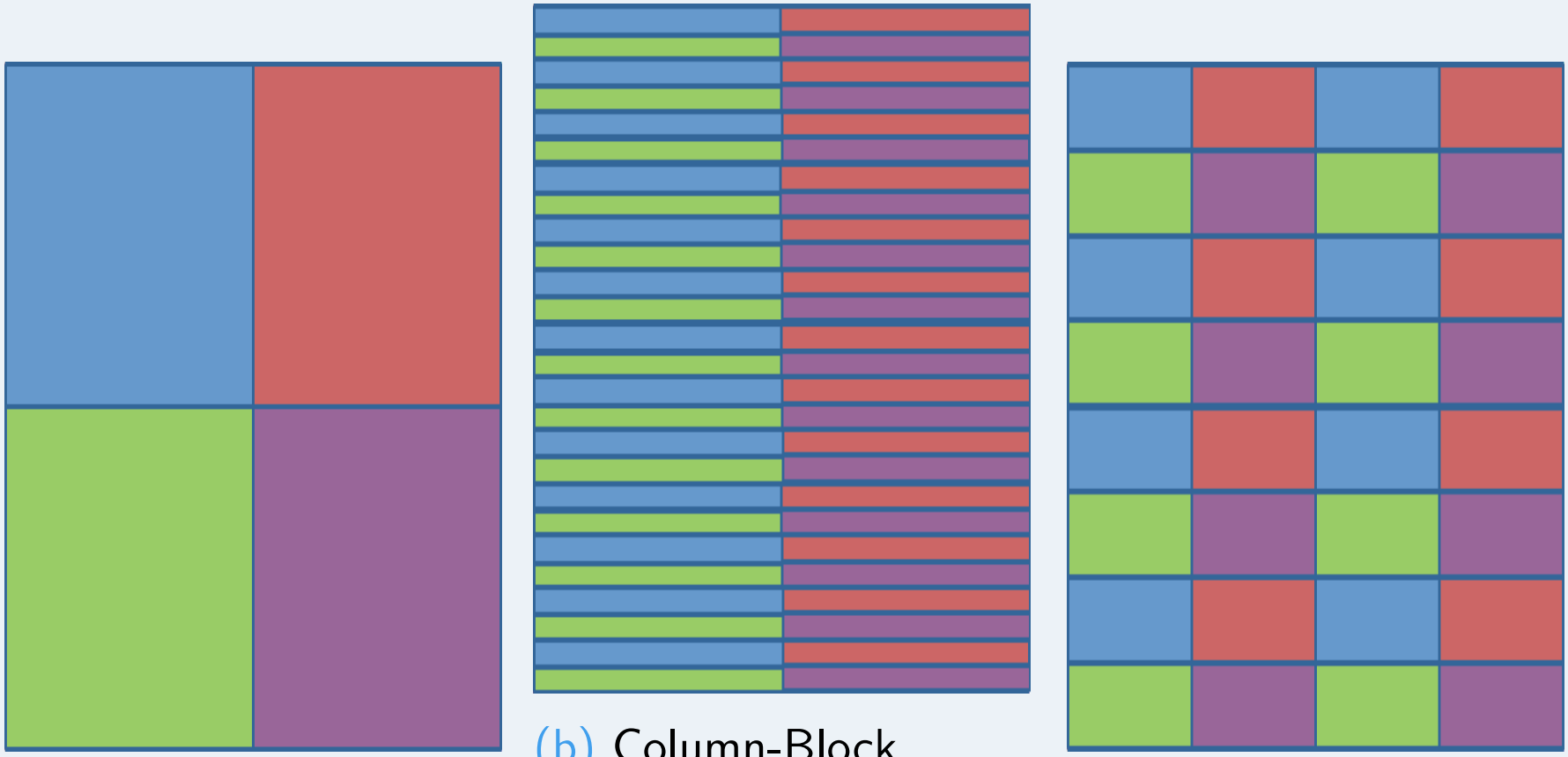
Distributed Matrices



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

Figure: Matrix Distribution Schemes Onto a 4×1 Processor Grid

Distributed Matrices



(a) Block

(b) Column-Block Row-Cyclic

(c) Block-Cyclic

Figure: Matrix Distribution Schemes Onto a 2×2 Processor 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

The ddmatrix Class

For **d**istributed **d**ense **matrix** objects, we use the special S4 class `ddmatrix`.

$$\texttt{ddmatrix} = \begin{cases} \textbf{Data} & \text{The local submatrix (an R matrix)} \\ \textbf{dim} & \text{Dimension of the global matrix} \\ \textbf{ldim} & \text{Dimension of the local submatrix} \\ \textbf{bldim} & \text{Dimension of the blocks} \\ \textbf{ICTXT} & \text{MPI Grid Context} \end{cases}$$

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: Row Block

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

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

ddmatrix: Row-Column 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: Row Cyclic

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

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

ddmatrix: Row-Cyclic Column-Block

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

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

ddmatrix: Block-Cyclic

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

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

7 Introduction to pbdDMAT and the ddmatrix Structure

- Introduction to Distributed Matrices
- pbdDMAT
- Summary

The ddmatrix Structure

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

ddmatrix: Block-Cyclic on a 2×3 Processor Grid

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

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

Understanding `ddmatrix`: Local View of Block-Cyclic on a 2×3 Processor Grid

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

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

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

Pros and Cons of This Data Structure

Pros

- Robust for matrix computations.

Cons

- Complex 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.

pbdDMAT:

- Distributed matrices + statistics.
- The `ddmatrix` structure *must* be used for **pbdDMAT**.
- If the data is not 2d block-cyclic compatible, `ddmatrix` will *definitely* give the wrong answer.

7 Introduction to pbdDMAT and the ddmatrix Structure

- Introduction to Distributed Matrices
- pbdDMAT
- Summary

Summary

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()
```

Contents

8 Examples Using pbdDMAT

- Manipulating ddmatrixObjects
- Statistics Examples with pbdDMAT
- RandSVD
- Summary

Example 1: ddmatrix Construction

Generate a global matrix and distribute it

```
1  library(pbdDMAT, quietly=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()
```

Example 2: ddmatrix Construction

Generate locally only what is needed

```
1  library(pbdDMAT, quietly=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()
```

Example 3: ddmatrix Operations

Generate locally only what is needed

```
1  library(pbdDMAT, quietly=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()
```

Example 4: More Basic Operations

```
1  library(pbdDMAT, quietly=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()
```

Example 5: Using apply()

```
1  library(pbdDMAT, quietly=TRUE)
2  init.grid()
3
4  x.dmat <- ddmatrix(1:30, nrow=10)
5
6  colsd <- apply(X=x.dmat, FUN=sd, MARGIN=2)
7  rowprods <- as.matrix(colsd)
8
9  comm.print(colsd)
10
11 finalize()
```

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 6: Dimension Reduction via PCA

```
1  library(pbdDMAT, quietly=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", quietly=T)
17 comm.cat("%Cols:", i/dim(x.dmat)[2], "\n\n", quietly=T)
18
19 finalize()
```

Distributed Matrices

pbdDEMO contains many other examples of reading and managing data.

Randomized truncated 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, Martinsson, and Tropp. 2011. Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions *SIAM Review* **53** 217–288

Randomized truncated SVD

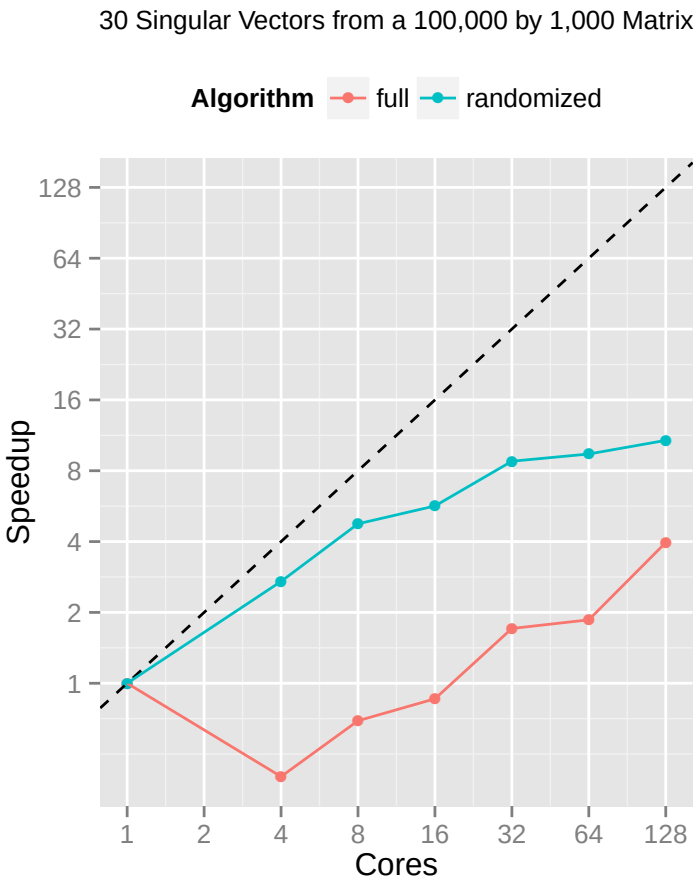
Serial R

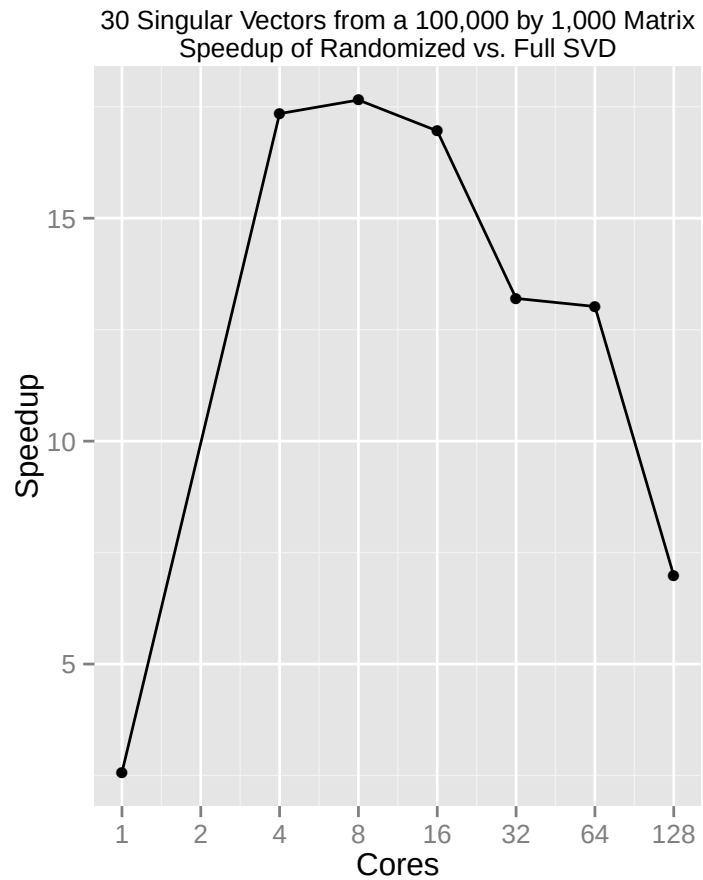
```
1  randSVD <- function(A, k, q=3)
2    {
3      ## Stage A
4      Omega <-  matrix(rnorm(n*2*k),
5          nrow=n, ncol=2*k)
6      Y <- A %*% Omega
7      Q <- qr.Q(qr(Y))
8      At <- t(A)
9      for(i in 1:q)
10       {
11         Y <- At %*% Q
12         Q <- qr.Q(qr(Y))
13         Y <- A %*%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

From journal to scalable code and scaling data in one day.





8 Examples Using pbdDMAT

- Manipulating `ddmatrix`Objects
- Statistics Examples with pbdDMAT
- RandSVD
- Summary

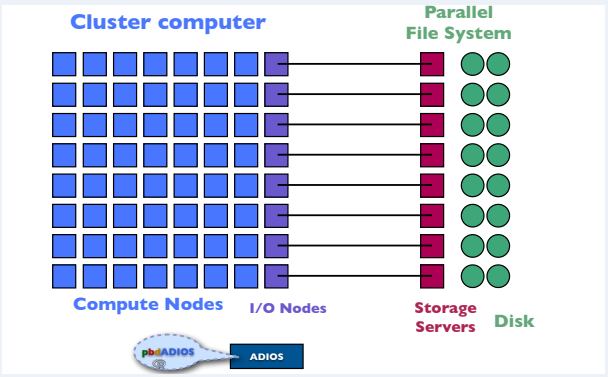
Summary

- **pbdDMAT** makes distributed (dense) linear algebra easier
- Enables rapid prototyping at large scale

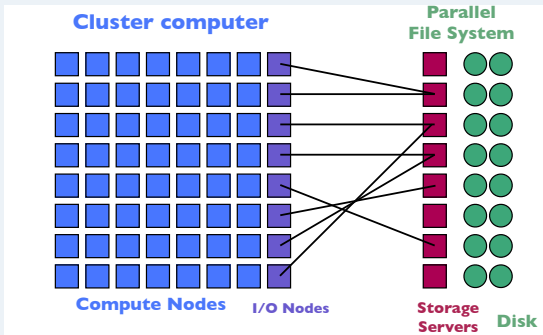
Contents

I/O Configurations

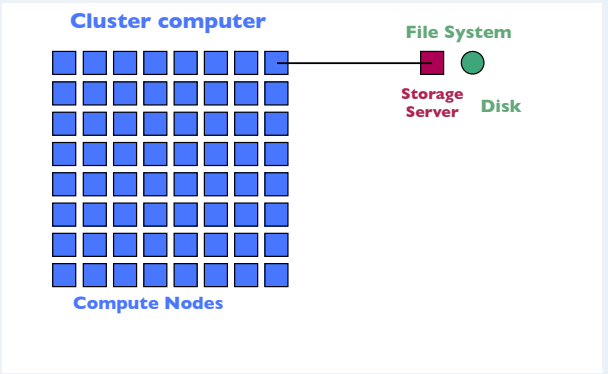
Best!



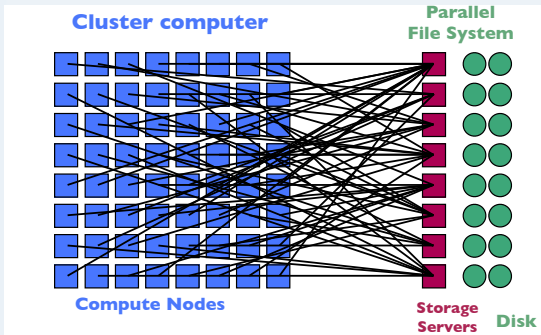
Good.



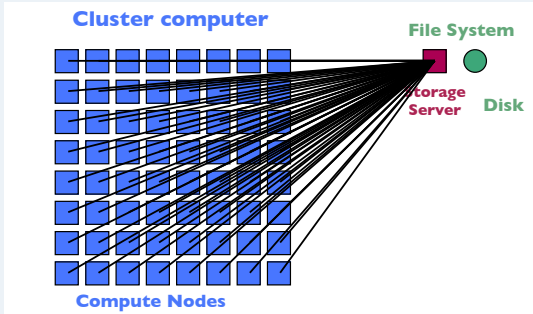
If you have to.



Can work.



Don't do this!



R has a separate manual for data import/export: http://r-project.org/

- scan(), readLines(), read.table(), read.csv(), ...
- seek(), readBin()
- CSV, SQL, NetCDF4, HDF, custom binary, ...

CSV Data: Read Serial then Distribute

0_readcsv.r

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

Parallel reading brings new issues

- How to partition data across nodes?
- How to structure for scalable libraries?
- Read directly into form needed or restructure?
- CSV, SQL, NetCDF4, HDF, ADIOS, custom binary
- ...
- Lot of work needed to make it intuitive!
- Currently under development.

CSV Data

Serial Code

```
1 x <- read.csv("x.csv")
2
3 x
```

Parallel Code

```
1 library(pbdDEMO, quiet = TRUE)
2 init.grid()
3
4 dx <- read.csv.ddmatrix("x.csv", header=TRUE, sep=",",
5           nrows=10, ncols=10, num.rdrs=2, ICTXT=0)
6
7 dx
8
9 finalize()
```

Binary Data: Vector

```
size <- 8 # bytes

my_ids <- get.jid(n)   # my index values from n

my_start <- (my_ids[1] - 1)*size    # my starting byte location
my_length <- length(my_ids)     # my number of bytes to read

con <- file("binary.vector.file", "rb")
seekval <- seek(con, where=my_start, rw="read")
x <- readBin(con, what="double", n=my_length, size=size)
```

Hadoop HDFS, RHadoop

ravro - read and write files in avro format

plyrmr - higher level plyr-like data processing for structured data, powered by rmr

rmr - functions providing Hadoop MapReduce functionality in R

rhdfs - functions providing file management of the HDFS from within R

rhbase - functions providing database management for the HBase distributed database from within R

Binary Data: Matrix

```
size <- 8 # bytes

my_ids <- get.jid(ncol)
my_ncol <- length(my_ids)
my_start <- (my_ids[1] - 1)*nrow*size
my_length <- my_ncol*nrow

con <- file("binary.matrix.file", "rb")
seekval <- seek(con, where=my_start, rw="read")
x <- readBin(con, what="double", n=my_length, size=size)

gdim <- c(nrow, ncol)
ldim <- c(nrow, my_ncol)
bldim <- c(nrow, allreduce(my_ncol, op="max"))
X <- new("ddmatrix", Data=matrix(x, nrow, my_ncol), dim=gdim,
    ldim=ldim, bldim=bldim, ICTXT=1)    # glue together as
    column-block ddmatrix

X <- redistribute(X, bldim=c(2, 2), ICTXT=0)  # redistribute as
    block-cyclic
Xprc <- prcomp(X)   # proceed as with serial code
```

NetCDF4 Data

```
### parallel read after determining start and length
nc <- nc_open_par(file_name)

nc_var_par_access(nc, "variable_name")
new.X <- ncvar_get(nc, "variable_name", start, length)
nc_close(nc)
```

ADIOS Data (.bp files)

pbdADIOS is under construction

```
### parallel read after determining start and length
file <- adios_open(file_name)

## Bounding box (start, length) access
## Staging capability with ADIOS configured simulation codes
## Streaming access

adios_close()
```

Summary

- Mostly "do it yourself" with bounding box
- Parallel file system for big data
  - Binary files for true parallel reads
  - Use correct number of readers vs number of storage servers
- Redistribution help from `ddmatrix` functions
- More intuitive input under development
- Staging and in situ capabilities via ADIOS soon

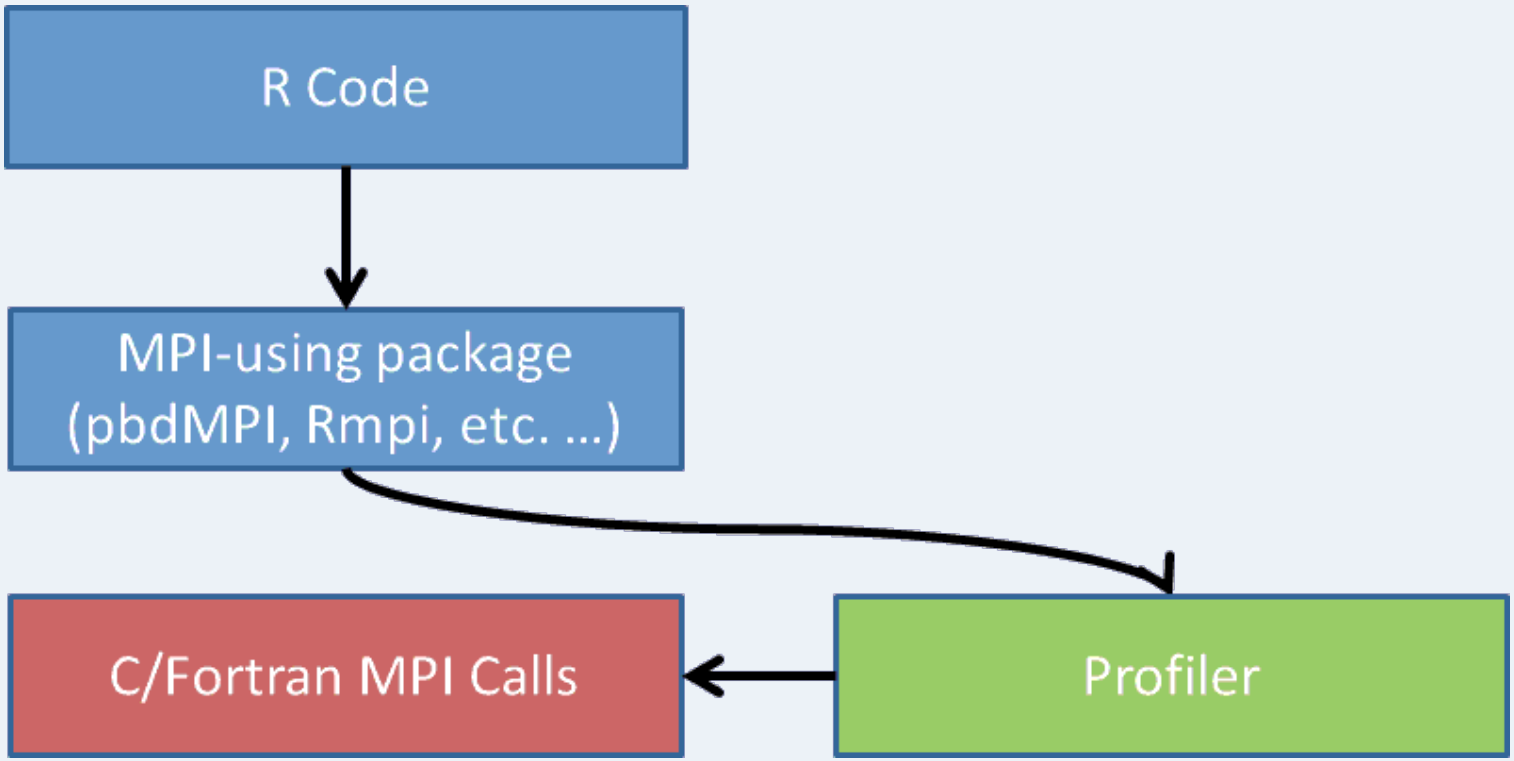
Contents

Introduction to **pbdPROF**

- Successful Google Summer of Code 2013 project.
- Available on the CRAN.
- Enables profiling of MPI-using R scripts.
- pbdR packages officially supported; can work with others...
- Also reads, parses, and plots profiler outputs.

How it works

MPI calls get hijacked by profiler and logged:



Introduction to **pbdPROF**

- Currently supports the profilers **fpmpi** and **mpiP**.
- **fpmpi** is distributed with **pbdPROF** and installs easily, but offers minimal profiling capabilities.
- **mpiP** is fully supported also, but you have to install and link it yourself.

Installing **pbdPROF**

1. Build **pbdPROF**.
2. Rebuild **pbdMPI** (linking with **pbdPROF**).
3. Run your analysis as usual.
4. Interactively analyze profiler outputs with **pbdPROF**.

This is explained at length in the **pbdPROF** vignette.

Rebuild **pbdMPI**

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

- Any package which explicitly links with an MPI library must be rebuilt in this way (**pbdMPI**, **Rmpi**, ...).
- Other pbdR packages link with **pbdMPI**, and so do not need to be rebuilt.
- See **pbdPROF** vignette if something goes wrong.

An Example from **pbdDMAT**

- Compute SVD in **pbdDMAT** package.
- Profile MPI calls with **mpiP**.

Example Script

my_svd.r

```
1  library(pbdMPI, quietly=TRUE)
2  library(pbdDMAT, quietly=TRUE)
3  init.grid()
4
5
6  n <- 1000
7  x <- ddmatrix("rnorm", n, n)
8
9  my.svd <- La.svd(x)
10
11
12 finalize()
```

Example Script

Run example with 4 ranks:

```
$ mpirun -np 4 Rscript my_svd.r
mpiP:
mpiP: mpiP: mpiP V3.3.0 (Build Sep 23 2013/14:00:47)
mpiP: Direct questions and errors to
    mpip-help@lists.sourceforge.net
mpiP:
Using 2x2 for the default grid size

mpiP:
mpiP: Storing mpiP output in [./R.4.5944.1.mpiP].
mpiP:
```

Read Profiler Data into R

Interactively (or in batch) Read in Profiler Data

```
1 library(pbdPROF)
2 prof.data <- read.prof("R.4.28812.1.mpiP")
```

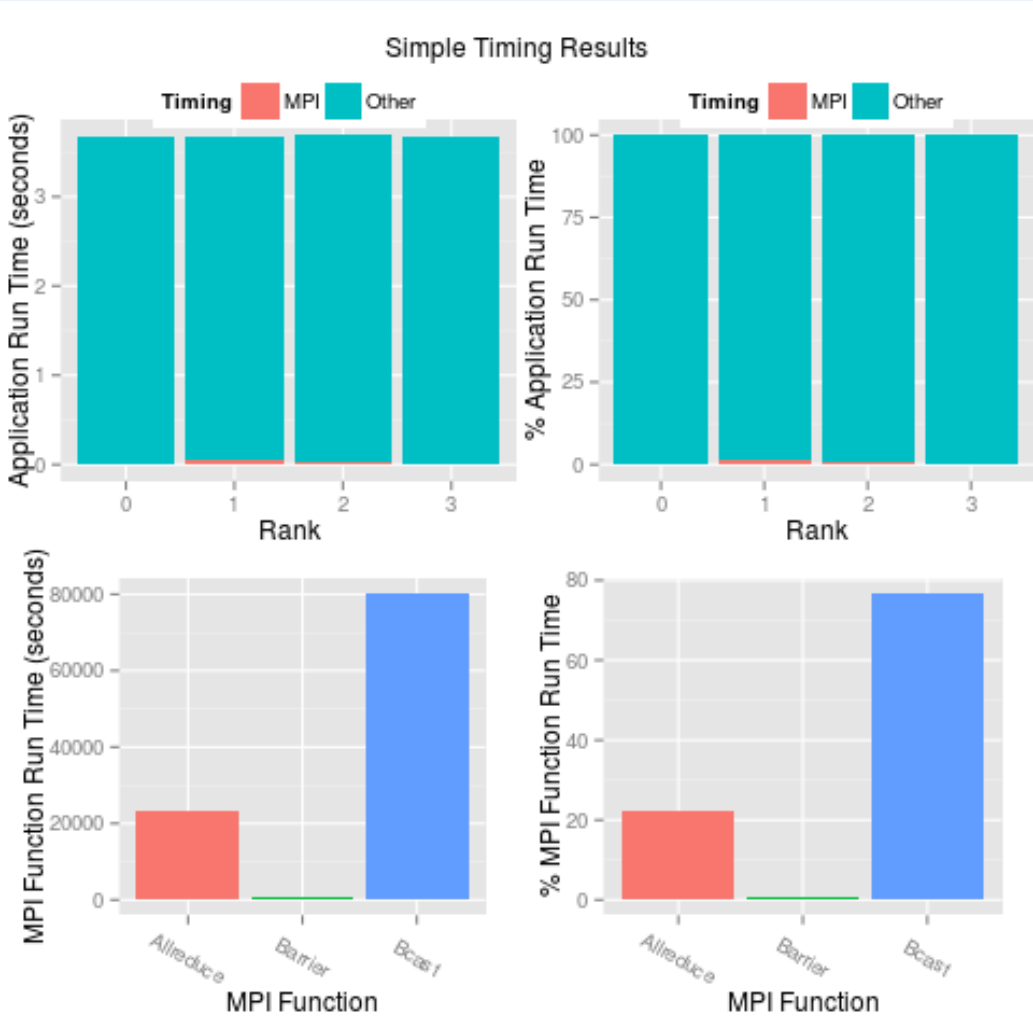
Partial Output of Example Data

```
> prof.data
An mpip profiler object:
[[1]]
  Task  AppTime  MPITime  MPI.
1    0     5.71   0.0387  0.68
2    1     5.70   0.0297  0.52
3    2     5.71   0.0540  0.95
4    3     5.71   0.0355  0.62
5    *    22.80   0.1580  0.69

[[2]]
   ID  Lev  File.Address  Line_Parent_Funct   MPI_Call
1   1    0  1.397301e+14          [unknown]  Allreduce
2   2    0  1.397301e+14          [unknown]      Bcast
```

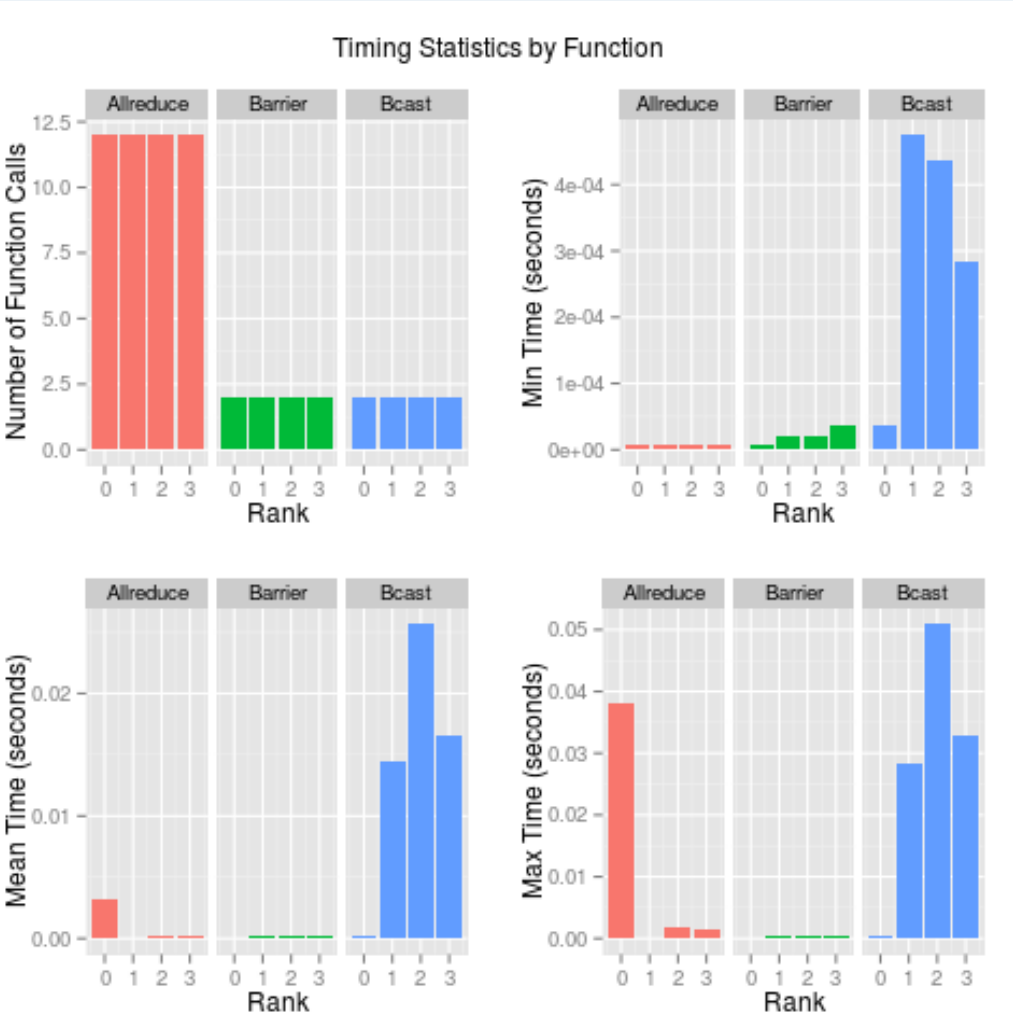
Generate plots

```
1 plot(prof.data)
```



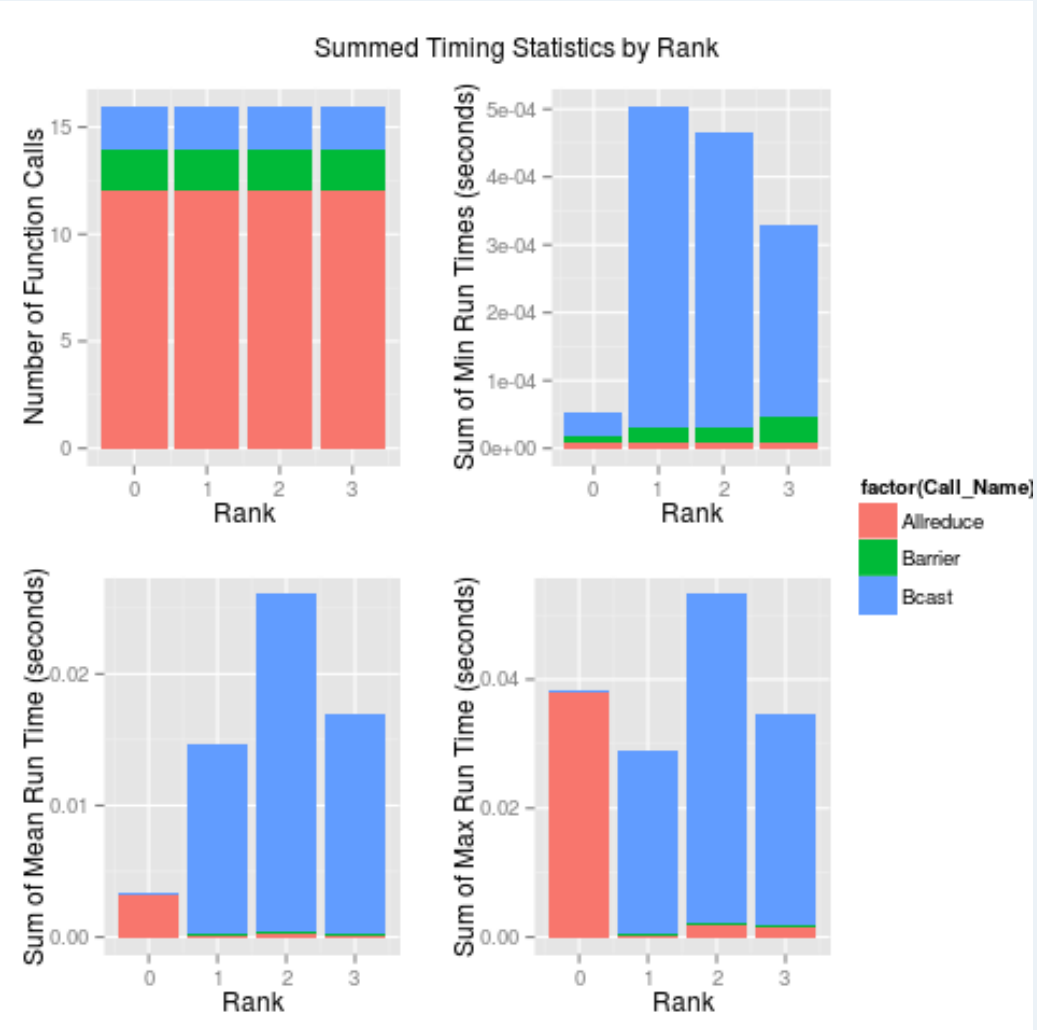
Generate plots

```
1 plot(prof.data, plot.type="stats1")
```



Generate plots

```
1 plot(prof.data, plot.type="stats2")
```

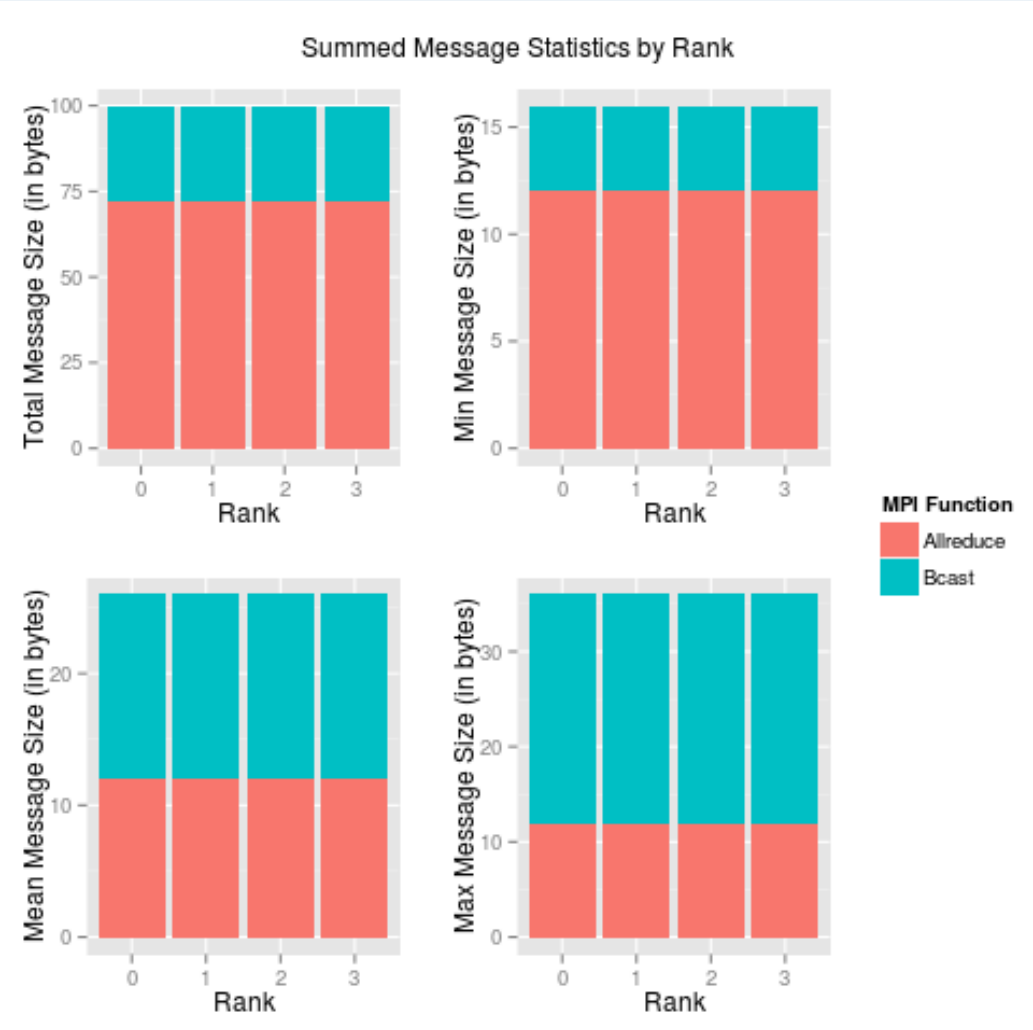


Generate plots

```
1 plot(prof.data, plot.type="messages1")
```

Generate plots

```
1 plot(prof.data, plot.type="messages2")
```



Summary

- **pbdPROF** offers tools for profiling R-using MPI codes.
- Easily builds **fpmpi**; also supports **mpiP**.

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

Coherent Modes in Turbulent Flow

Get and Redistribute the Data

```
library(pbdDMAT, quiet = TRUE)
init.grid()

## load local data (file assumes 4 processors!)
g.dim <- c(64, 64, 1024)
my.dim <- g.dim / c(1, 1, comm.size())
save.file <- paste("xyz.RData", comm.rank(), sep="") # assumes
    4 processors!
load(save.file)

## reshape 3d array into a matrix for PCA (EOF) computation
## first two dimensions become rows and third becomes columns
## local reshape dimensions
my.nrow <- prod(my.dim[1:2])
my.ncol <- my.dim[3]
ldim <- c(my.nrow, my.ncol)

## global reshape dimensions
g.nrow <- prod(g.dim[1:2])
g.ncol <- g.dim[3]
gdim <- c(g.nrow, g.ncol)

## now reshape local
X <- matrix(vx, nrow=my.nrow, ncol=my.ncol, byrow=FALSE)
Y <- matrix(vy, nrow=my.nrow, ncol=my.ncol, byrow=FALSE)
Z <- matrix(vz, nrow=my.nrow, ncol=my.ncol, byrow=FALSE)

## glue local pieces into a ddmatrix
X <- new("ddmatrix", Data=X, dim=gdim, ldim=ldim, bldim=ldim,
    ICTXT=1)
Y <- new("ddmatrix", Data=Y, dim=gdim, ldim=ldim, bldim=ldim,
    ICTXT=1)
Z <- new("ddmatrix", Data=Z, dim=gdim, ldim=ldim, bldim=ldim,
    ICTXT=1)

## transform to 2d block cyclic
X <- redistribute(X, bldim=c(8,8), ICTXT=0)
Y <- redistribute(Y, bldim=c(8,8), ICTXT=0)
Z <- redistribute(Z, bldim=c(8,8), ICTXT=0)
```

Coherent Modes in Turbulent Flow

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

```
E <- sqrt(X^2 + Y^2 + Z^2) # energy from velocity
E.pca <- prcomp(x=E, retx=TRUE, scale=FALSE)

# plot using one process
if(comm.rank() == 0)
  {
    ## scree plot for first 50 components
    variance <- E.pca$sdev^2  # note: all own sdev
    proportion <- variance[1:50]/sum(variance)
    cumulative <- cumsum(proportion)
    component <- 1:length(proportion)
    png("scree.png")
    plot(component, cumulative, ylim=c(0,1))
    points(component, proportion, type="h", col="blue")
    dev.off()
  }

finalize()
```

How can we plot in parallel?

- Several plots, one on each processor
- One plot by several processors

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 }
```

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

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

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   }
```

SimpleRedistributions

- 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 0_pca.r, 1_plot.r, 2_plot.r, 3_plot.r, 4_plot.r, 5ictxt.r, 6_ictxt.r, and 7_ictxt.r
- Experiment with other redistributions

Contents

12 Clustering Distributed Data

- Using **pmclust** for model-based clustering
- **pmclust** examples with various data locations
- Summary

Background

- Gaussian mixture model
- $X_n \overset{iid}{\sim} \sum_{i=1}^{k} \eta_i \text{MVN}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$
- For k-means, take $\boldsymbol{\Sigma}_i = \sigma^2 I$
- Estimate with four possible variants of EM algorithm EM (Dempster et al. 1977), AECM (Meng and Van Dyk 1997), APECM (Chen and Maitra 2011), APECMa (Chen et al. 2013), and K-means (Lloyd 1982)
- Sophisticated starting values generation
- Result still a random variable but stable for good k

Data starts on single instance

iris_single.r

```
1  library(pmclust, quietl = TRUE)
2
3  if(comm.rank() == .SPMD.CT$rank.source){
4    ### Load data
5    X <- as.matrix(iris[, -5])
6
7    ### Standardize
8    X.std <- scale(X)
9  } else X.std <- NULL
10
11 ### Cluster
12 library(pmclust, quietly = TRUE)
13 comm.set.seed(123, diff = TRUE)
14
15 ret.mb1 <- pmclust(X.std, K = 3, method.own.X = "single")
16 comm.print(ret.mb1)
17
18 ret.kms <- pkmeans(X.std, K = 3, method.own.X = "single")
19 comm.print(ret.kms)
20
21 finalize()
```

All instances have complete data

iris_common.r

```
1  library(pmclust, quietly = TRUE)
2
3  ### Load data
4  X <- as.matrix(iris[, -5])
5
6  ### Standardize
7  X.std <- scale(X)
8
9  ### Cluster
10 library(pmclust, quietly = TRUE)
11 comm.set.seed(123, diff = TRUE)
12
13 ret.mb1 <- pmclust(X.std, K = 3, method.own.X = "common")
14 comm.print(ret.mb1)
15
16 ret.kms <- pkmeans(X.std, K = 3, method.own.X = "common")
17 comm.print(ret.kms)
18
19 finalize()
```

Data distributed by blocks of rows

iris_gbdr.r

```
1  library(pmclust, quietly = TRUE)
2
3  ### Load data
4  X <- as.matrix(iris[, -5])
5
6  ### Distribute data
7  jid <- get.jid(nrow(X))
8  X.gbd <- X[jid,]
9
10 ### Standardize
11 N <- allreduce(nrow(X.gbd))
12 p <- ncol(X.gbd)
13 mu <- allreduce(colSums(X.gbd / N))
14 X.std <- sweep(X.gbd, 2, mu, FUN = "-")
15 std <- sqrt(allreduce(colSums(X.std^2 / (N - 1))))
16 X.std <- sweep(X.std, 2, std, FUN = "/")
17
18 ### Cluster
19 library(pmclust, quietly = TRUE)
20 comm.set.seed(123, diff = TRUE)
21
22 ret.mb1 <- pmclust(X.std, K = 3)
```

Data in block-cyclic ddmatrix layout

iris_dmat.r

```
1  library(pbdDMAT, quiet = TRUE)
2  library(pmclust, quiet = TRUE)
3  init.grid()
4
5  ### Load data
6  X <- as.matrix(iris[, -5])
7
8  ### Convert to ddmatrix
9  X.dmat <- as.ddmatrix(X)
10
11 ### Standardize
12 X.std <- scale(X.dmat)
13
14 ### Cluster
15 library(pmclust, quietly = TRUE)
16 comm.set.seed(123, diff = TRUE)
17
18 ret.mb1 <- pmclust(X.std, K = 3)
19 comm.print(ret.mb1)
20
21 ret.kms <- pkmeans(X.std, K = 3)
22 comm.print(ret.kms)
```

12 Clustering Distributed Data

- Using **pmclust** for model-based clustering
- **pmclust** examples with various data locations
- Summary

Summary

- **pmclust** handles model-based clustering of distributed data
- Enables rapid prototyping at large scale

Contents

Summary

- Profile your code to understand your bottlenecks
- pbdR makes distributed parallelism with R easier
- pbdR connects R to HPC scalable libraries
- Distributing data to multiple nodes
- For truly large data, I/O must be parallel

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?

http://r-pbd.org/