Introducing R:
From Your Laptop to HPC and Big Data

George Ostrouchov and Drew Schmidt

SC14

The pbdR Core Team

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

pbdR
Programming with Big Data in R

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]Department of Ecology and Evolutionary Biology
University of Tennessee, Knoxville TN, 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

1. Introduction
2. Profiling and Benchmarking
3. The pbdR Project
4. Introduction to pbdMPI
5. Distributing Data
6. Basic Statistics Examples
7. Introduction to pbdDMAT and the ddmatrix Structure
8. Examples Using pbdDMAT
9. Data Input
10. MPI Profiling
11. Wrapup

Contents

1 Introduction
- What is R?
- An R and pbdR View of Parallel Hardware and Software
- Summary

1 Introduction
- What is R?
- An R and pbdR View of Parallel Hardware and Software
- Summary

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 5000 user-contributed packages.

Who uses R?

ORACLE
MERCK
KICKSTARTER
Bank of America
ebay
The New York Times
JOHN DEERE
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 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: $-\left(2^{-31}-1\right)$
  - 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

Language Rank	Types	Spectrum Ranking
1. Java		100.0
2. C		99.3
3. C++		95.5
4. Python		93.4
5. C#		92.4
6. PHP		84.7
7. Javascript		84.4
8. Ruby		78.8
9. R		74.2
10. MATLAB		72.9

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

Resources for Learning R

- *Advanced R* by Hadley Wickham: http://adv-r.had.co.nz/
- *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
- *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

Big Data and Little Data

Analysis Workflow

- Get Big Data
  - Parallel data reader
  - Parallel data generator
- Write analysis script
- Graphics to display results
- Profile and optimize code

R and pbdR Interfaces to Libraries: Harness 30+ Years of HPC Research

Hadoop: Will it merge with HPC in the future?

Three Hadoop components

- File system: HDFS
- Resource manager: Yarn
- Algorithm limitation: Map Reduce, Manager+Workers

R and pbdR Interfaces to Libraries: "Standing on the Shoulders of Giants"

1 Introduction
- What is R?
- An R and pbdR View of Parallel Hardware and Software
- Summary

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

2 Profiling and Benchmarking
- Profiling R Code
- Advanced R Profiling
- Summary

2 Profiling and Benchmarking
- Profiling R Code
- Advanced R Profiling
- Summary

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

```
x <- matrix(rnorm(20000*750), nrow=20000, ncol=750)

system.time(t(x) %*% x)
#    user  system elapsed
#   2.187   0.032   2.324

system.time(crossprod(x))
#    user  system elapsed
#   1.009   0.003   1.019

system.time(cov(x))
#    user  system elapsed
#   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)
```

```
x <- matrix(rnorm(10000*250), nrow=10000, ncol=250)

Rprof()
invisible(prcomp(x))
Rprof(NULL)

summaryRprof()

Rprof(interval=.99)
invisible(prcomp(x))
Rprof(NULL)

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:

```
x <- matrix(rnorm(10000*500), nrow=10000, ncol=500)

f <- function(x) t(x) %*% x
g <- function(x) crossprod(x)

library(rbenchmark)
benchmark(f(x), g(x))

#   test replications elapsed relative
# 1 f(x)          100  64.153    2.063
# 2 g(x)          100  31.098    1.000
```

2 Profiling and Benchmarking
- Profiling R Code
- Advanced R Profiling
- Summary

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

```
library(pbdPROF)
prof <- read.prof( "output.mpiP")
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*

2 Profiling and Benchmarking
- Profiling R Code
- Advanced R Profiling
- Summary

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

3 The pbdR Project
- The pbdR Project
- pbdR Connects R to HPC Libraries
- Using pbdR
- Summary

3 The pbdR Project
- The pbdR Project
- pbdR Connects R to HPC Libraries
- Using pbdR
- Summary

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 Packages

pbdR Motivation

Why HPC libraries (MPI, ScaLAPACK, PETSc, . . .)?

- The HPC community has been at this for decades.
- *They're tested. They work. They're fast.*
- You're not going to beat Jack Dongarra at dense linear algebra.

Simple Interface for MPI Operations with **pbdMPI**

Rmpi

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

pbdMPI

```
allreduce(x)
```

Types in R

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

Distributed Matrices and Statistics with **pbdDMAT**

Matrix Exponentiation

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

dx <- ddmatrix("rnorm", 5000, 5000)

expm(dx)

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

3 The pbdR Project
- The pbdR Project
- pbdR Connects R to HPC Libraries
- Using pbdR
- Summary

R and pbdR Interfaces to HPC Libraries

3 The pbdR Project
- The pbdR Project
- pbdR Connects R to HPC Libraries
- Using pbdR
- Summary

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:

```
Rscript my_script.r
```

or

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

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

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

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

4 Introduction to pbdMPI
- Managing a Communicator
- Reduce, Gather, Broadcast, and Barrier
- Other pbdMPI Tools
- Summary

4 Introduction to pbdMPI
- Managing a Communicator
- Reduce, Gather, Broadcast, and Barrier
- Other pbdMPI Tools
- Summary

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

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

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

finalize()
```

Execute this script via:

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

Sample Output:

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

Quick Example 2

Hello World: 2_hello.r

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

comm.print("Hello, world")

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

finalize()
```

Execute this script via:

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

Sample Output:

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

4 Introduction to pbdMPI
- Managing a Communicator
- Reduce, Gather, Broadcast, and Barrier
- Other pbdMPI Tools
- Summary

MPI Operations (2 of 2)

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

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

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

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

Quick Example 3

Reduce and Gather: 3_gt.r

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

comm.set.seed(diff=TRUE)

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

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

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

finalize()
```

Execute this script via:

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

Sample Output:

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

Quick Example 4

Broadcast: 4_bcast.r

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

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

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

comm.print(y, rank=1)

finalize()
```

Execute this script via:

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

Sample Output:

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

4 Introduction to pbdMPI
- Managing a Communicator
- Reduce, Gather, Broadcast, and Barrier
- Other pbdMPI Tools
- Summary

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()`

4 Introduction to pbdMPI
- Managing a Communicator
- Reduce, Gather, Broadcast, and Barrier
- Other pbdMPI Tools
- Summary

Summary

- Start by loading the package:

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

- Always initialize before starting and finalize when finished:

```
init()

# ...

finalize()
```

Contents

5 Distributing Data
- A Way to Distribute Your Data
- Summary

5 Distributing Data
- A Way to Distribute Your Data
- Summary

Distributing Data

Problem: How to distribute the data

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

?

Distributing a Matrix Across 4 Processors: Block Distribution

Data

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

Processors

0
1
2
3

Distributing a Matrix Across 4 Processors: Local Load Balance

Data

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

Processors

0
1
2
3

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

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

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

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

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

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

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

Processors = 0 1 2 3 4 5

Distributing by Row: Non-Balanced

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

Processors = 0 1 2 3 4 5

Distributing by Row: 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

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

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

Parallel Code

```
library(pbdMPI, quiet = TRUE)
init()
comm.set.seed(seed=1234567, diff=TRUE)

N.loc <- 50000 / comm.size()
X.loc <- matrix(runif(N.loc * 2), ncol = 2)
r.loc <- sum(rowSums(X.loc^2) <= 1)
r <- allreduce(r.loc)
PI <- 4*r/(N.loc * comm.size())
comm.print(PI)

finalize()
```

Note

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

6 Basic Statistics Examples
- pbdMPI Example: Monte Carlo Simulation
- pbdMPI Example: Sample Covariance
- pbdMPI Example: Linear Regression
- Summary

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

```
N <- nrow(X)
mu <- colSums(X) / N

X <- sweep(X, STATS=mu, MARGIN=2)
Cov.X <- crossprod(X) / (N-1)

print(Cov.X)
```

Parallel Code

```
N <- allreduce(nrow(X.loc), op="sum")
mu <- allreduce(colSums(X.loc) / N, op="sum")

X.loc <- sweep(X.loc, STATS=mu, MARGIN=2)
Cov.X <- allreduce(crossprod(X.loc), op="sum") / (N-1)

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

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

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

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

Example 3: Linear Regression 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

```
tX <- t(X)
A <- tX %*% X
B <- tX %*% y

ols <- solve(A) %*% B
```

Parallel Code

```
tX.loc <- t(X.loc)
A <- allreduce(tX.loc %*% X.loc, op = "sum")
B <- allreduce(tX.loc %*% y.loc, op = "sum")

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

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. . .

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

(a) 1×6	(b) 2×3	(c) 3×2	(d) 6×1
$\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}^T$	$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$

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: 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

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

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

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

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

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

ddmatrix: 2-dimensional Block-Cyclic

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

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

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 distribution.

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

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

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

Understanding `ddmatrix`: Local View

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

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

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

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

```
cov(x)
```

Parallel Code

```
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:

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

2. Always initialize before starting and finalize when finished:

```
init.grid()

# ...

finalize()
```

Contents

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

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

Example 1: ddmatrix Construction

Generate a global matrix and distribute it

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

# Common global on all processors --> distributed
x <- matrix(1:100, nrow=10, ncol=10)
x.dmat <- as.ddmatrix(x)

x.dmat

# Global on processor 0 --> distributed
if (comm.rank()==0){
  y <- matrix(1:100, nrow=10, ncol=10)
} else {
  y <- NULL
}
y.dmat <- as.ddmatrix(y)

y.dmat

finalize()
```

Example 2: ddmatrix Construction

Generate locally only what is needed

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

zero.dmat <- ddmatrix(0, nrow=100, ncol=100)
zero.dmat

id.dmat <- diag(1, nrow=100, ncol=100, type="ddmatrix")
id.dmat

comm.set.seed(diff=T)
rand.dmat <- ddmatrix("rnorm", nrow=100, ncol=100, mean=10, sd=100)
rand.dmat

finalize()
```

Example 3: ddmatrix Operations

Generate locally only what is needed

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

x.dmat <- ddmatrix(1:30, nrow=10)
y.dmat <- x.dmat[c(1, 3, 5, 7, 9), -3]

comm.print(y.dmat)
y <- as.matrix(y.dmat)
comm.print(y)

finalize()
```

Example 4: More Basic Operations

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

x.dmat <- ddmatrix(1:30, nrow=10)
y.dmat <- x.dmat + 1:7
z.dmat <- scale(x.dmat, center=TRUE, scale=TRUE)

y <- as.matrix(y.dmat)
z <- as.matrix(z.dmat)

comm.print(y)
comm.print(z)

finalize()
```

Example 5: Using apply()

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

x.dmat <- ddmatrix(1:30, nrow=10)

colsd <- apply(X=x.dmat, FUN=sd, MARGIN=2)
rowprods <- as.matrix(colsd)

comm.print(colsd)

finalize()
```

Sample Covariance

Serial Code

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

Parallel Code

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

Linear Regression

Serial Code

```
tX <- t(X)
A <- tX %*% X
B <- tX %*% y

ols <- solve(A) %*% B

# or
ols <- lm.fit(X, y)
```

Parallel Code

```
tX <- t(X)
A <- tX %*% X
B <- tX %*% y

ols <- solve(A) %*% B

# or
ols <- lm.fit(X, y)
```

Example 6: Dimension Reduction via PCA

```
library(pbdDMAT, quietly=T)
init.grid()

n <- 1e4
p <- 250

comm.set.seed(diff=T)
x.dmat <- ddmatrix("rnorm", nrow=n, ncol=p, mean=100, sd=25)

pca <- prcomp(x=x.dmat, retx=TRUE, scale=TRUE)
prop_var <- cumsum(pca$sdev)/sum(pca$sdev)
i <- max(min(which(prop_var > 0.9)) - 1, 1)

y.dmat <- pca$x[, 1:i]

comm.cat("\nCols: ", i, "\n", quietly=T)
comm.cat("%Cols:", i/dim(x.dmat)[2], "\n\n", quietly=T)

finalize()
```

Distributed Matrices

pbdDEMO contains many other examples of reading and managing data.

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

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: $\boldsymbol{B} = \widetilde{\boldsymbol{U}}\boldsymbol{\Sigma V}^*$.
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 $\boldsymbol{Y}_j = \boldsymbol{Q}_j\boldsymbol{R}_j$.

Serial R

```
randSVD <- function(A, k, q=3)
  {
    ## Stage A
    Omega <-  matrix(rnorm(n*2*k),
                     nrow=n, ncol=2*k)
    Y <- A %*% Omega
    Q <- qr.Q(qr(Y))
    At <- t(A)
    for(i in 1:q)
      {
        Y <- At %*% Q
        Q <- qr.Q(qr(Y))
        Y <- A %*% Q
        Q <- qr.Q(qr(Y))
      }

    ## Stage B
    B <- t(Q) %*% A
    U <- La.svd(B)$u
    U <- Q %*% U
    U[, 1:k]
  }
```

Randomized SVD

Serial R

```
randSVD <- function(A, k, q=3)
  {
    ## Stage A
    Omega <- matrix(rnorm(n*2*k),
              nrow=n, ncol=2*k)
    Y <- A %*% Omega
    Q <- qr.Q(qr(Y))
    At <- t(A)
    for(i in 1:q)
      {
        Y <- At %*% Q
        Q <- qr.Q(qr(Y))
        Y <- A %*% Q
        Q <- qr.Q(qr(Y))
      }

    ## Stage B
    B <- t(Q) %*% A
    U <- La.svd(B)$u
    U <- Q %*% U
    U[, 1:k]
  }
```

Parallel pbdR

```
randSVD <- function(A, k, q=3)
  {
    ## Stage A
    Omega <- ddmatrix("rnorm",
              nrow=n, ncol=2*k)
    Y <- A %*% Omega
    Q <- qr.Q(qr(Y))
    At <- t(A)
    for(i in 1:q)
      {
        Y <- At %*% Q
        Q <- qr.Q(qr(Y))
        Y <- A %*% Q
        Q <- qr.Q(qr(Y))
      }

    ## Stage B
    B <- t(Q) %*% A
    U <- La.svd(B)$u
    U <- Q %*% U
    U[, 1:k]
  }
```

Randomized SVD

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

Summary

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

Contents

9 Data Input
- Considering I/O Configurations
- Serial Data Input
- Parallel Data Input
- Summary

9 Data Input
- Considering I/O Configurations
- Serial Data Input
- Parallel Data Input
- Summary

I/O Configurations

Best!

Good.

If you have to . . .

Can work.

Don't do this!

9 Data Input
- Considering I/O Configurations
- Serial Data Input
- Parallel Data Input
- Summary

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

- scan()
- read.table()
- read.csv()
- socket

CSV Data: Read Serial then Distribute

Listing :

```
library(pbdDMAT)
if(comm.rank() == 0) { # only read on process 0
  x <- read.csv("myfile.csv")
} else {
  x <- NULL
}

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

9 Data Input
- Considering I/O Configurations
- Serial Data Input
- Parallel Data Input
- Summary

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
- ...
- A lot of work needed here!
- Currently under development.

CSV Data

Serial Code

```
x <- read.csv("x.csv")

x
```

Parallel Code

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

dx <- read.csv.ddmatrix("x.csv", header=TRUE, sep=",",
          nrows=10, ncols=10, num.rdrs=2, ICTXT=0)

dx

finalize()
```

Binary Data: Vector

```
## set up start and length for reading a vector of n doubles
size <- 8 # bytes

my_ids <- get.jid(n, method="block")

my_start <- (my_ids[1] - 1)*size
my_length <- length(my_ids)

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

Binary Data: Matrix

```
## read an nrow by ncol matrix of doubles split by columns
size <- 8 # bytes

my_ids <- get.jid(ncol, method="block")
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)

## glue together as a column-block ddmatrix
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)

## redistribute for ScaLAPACK's block-cyclic
```

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)

finalize()
```

9 Data Input
- Considering I/O Configurations
- Serial Data Input
- Parallel Data Input
- Summary

Summary

- Mostly "do it yourself"
- Parallel file system for big data
  - Binary files for true parallel reads
  - Know number of readers vs number of storage servers
- Redistribution help from `ddmatrix` functions
- More help under development

Contents

10 MPI Profiling
- Profiling with the pbdPROF Package
- Installing pbdPROF
- Example
- Summary

10 MPI Profiling
- Profiling with the pbdPROF Package
- Installing pbdPROF
- Example
- Summary

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.

10 MPI Profiling
- Profiling with the pbdPROF Package
- Installing pbdPROF
- Example
- Summary

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.

10 MPI Profiling
- Profiling with the pbdPROF Package
- Installing pbdPROF
- Example
- Summary

An Example from **pbdDMAT**

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

Example Script

my_svd.r

```
library(pbdMPI, quietly=TRUE)
library(pbdDMAT, quietly=TRUE)
init.grid()


n <- 1000
x <- ddmatrix("rnorm", n, n)

my.svd <- La.svd(x)


finalize()
```

Example Script

Run example with 4 ranks:

```
$ mpirun -np 4 Rscript my_svd.r
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

```
library(pbdPROF)
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

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

Generate plots

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

Generate plots

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

Generate plots

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

Generate plots

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

10 MPI Profiling
- Profiling with the pbdPROF Package
- Installing pbdPROF
- Example
- Summary

Summary

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

Contents

11 Wrapup

Summary

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

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/