Programming with Big Data in R

Drew Schmidt and George Ostrouchov

useR! 2014

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

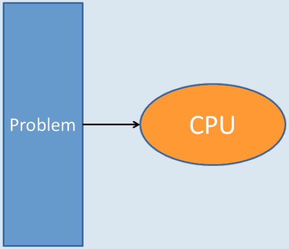
This includes instructions for installing R, MPI, and pbdR.

Contents

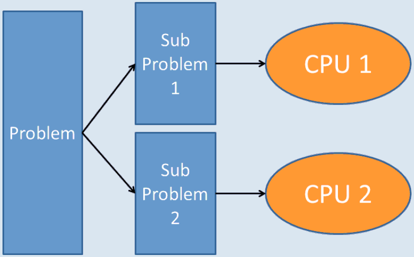
Contents

Parallelism

Serial Programming



Parallel Programming



Difficulty in Parallelism

1. *Implicit parallelism*: Parallel details hidden from user
2. *Explicit parallelism*: Some assembly required...
3. *Embarrassingly Parallel*: Also called *loosely coupled*. Obvious how to make parallel; lots of independence in computations.
4. *Tightly Coupled*: Opposite of embarrassingly parallel; lots of dependence in computations.

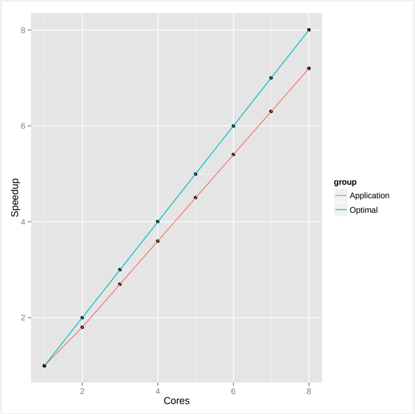
Speedup

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

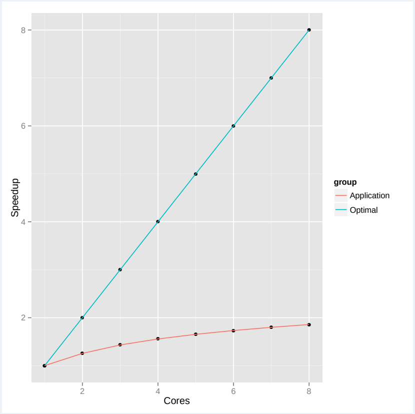
$$S_{n_1,n_2} = \frac{\text{Run time for } n_1 \text{ cores}}{\text{Run time for } n_2 \text{ cores}}$$

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

Good Speedup



Bad Speedup

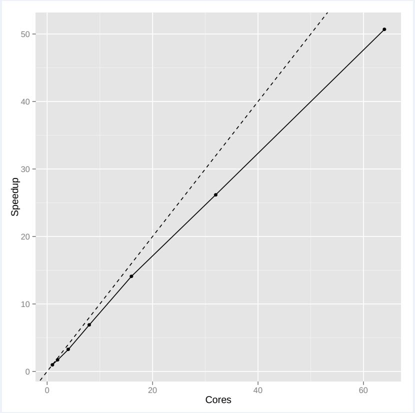


Scalability and Benchmarking

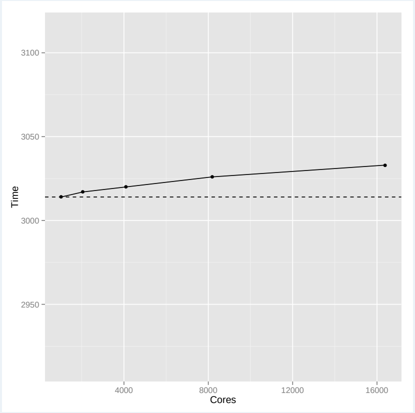
1. *Strong*: Fixed **total** problem size.
   Less work per core as more cores are added.

2. *Weak*: Fixed **local** (per core) problem size.
   Same work per core as more cores are added.

Good Strong Scaling



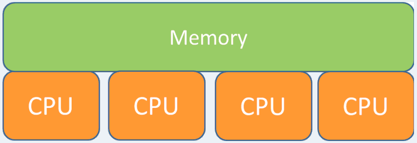
Good Weak Scaling



Shared and Distributed Memory Machines

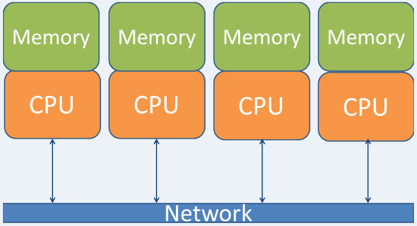
Shared Memory

Direct access to read/change memory (one node)



Distributed

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



Shared and Distributed Memory Machines

Shared Memory Machines

Thousands of cores

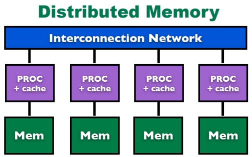
Nautilus, University of Tennessee
1024 cores
4 TB RAM

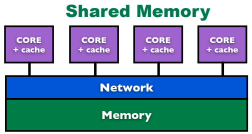
Distributed Memory Machines

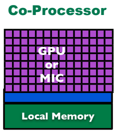
Hundreds of thousands of cores

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

Three Basic Flavors of Hardware



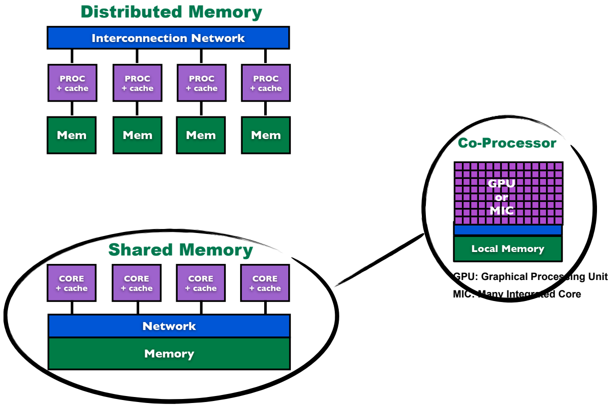




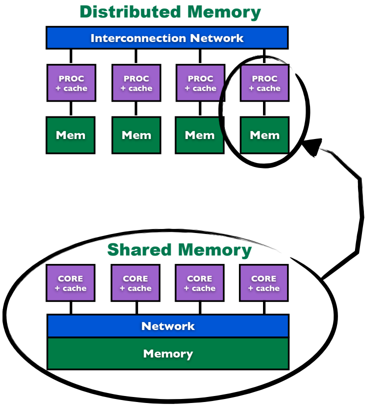
GPU: Graphical Processing Unit

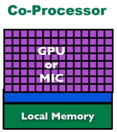
MIC: Many Integrated Core

Your Laptop or Desktop



A Server or Cluster

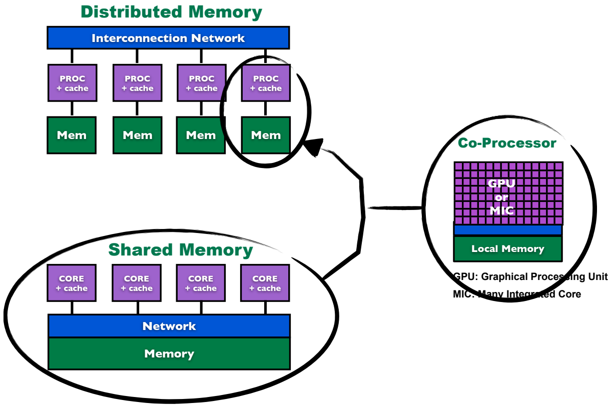




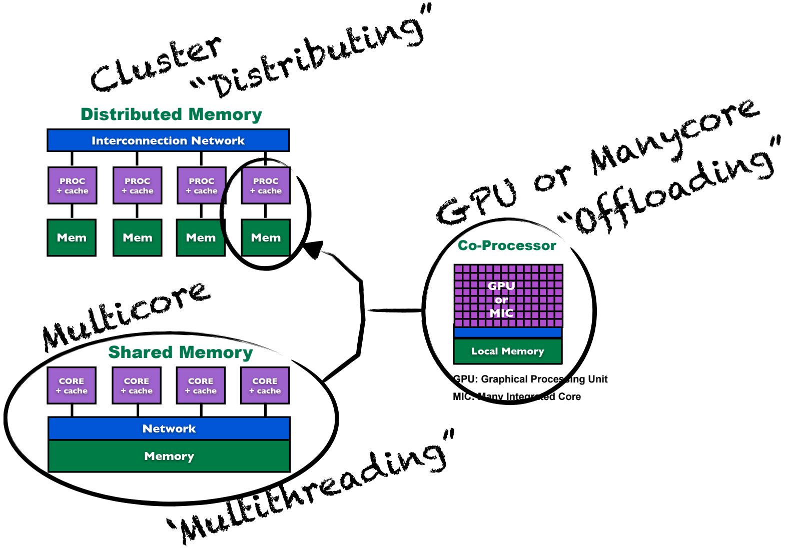
GPU: Graphical Processing Unit

MIC: Many Integrated Core

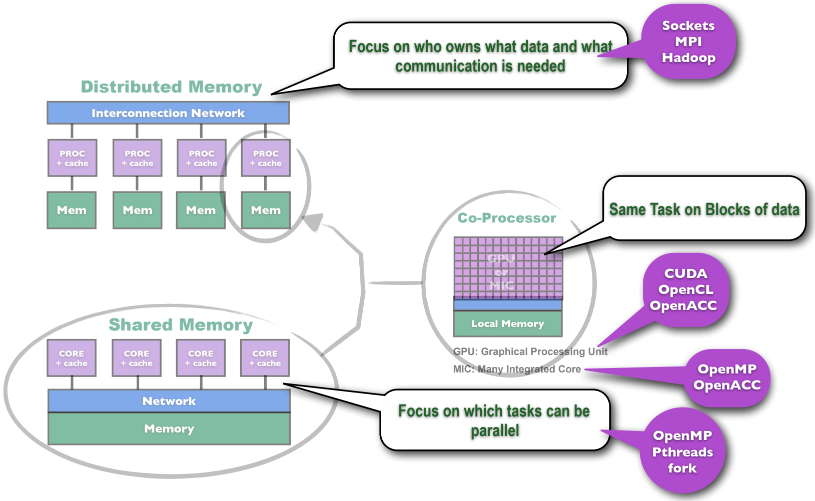
Server to Supercomputer



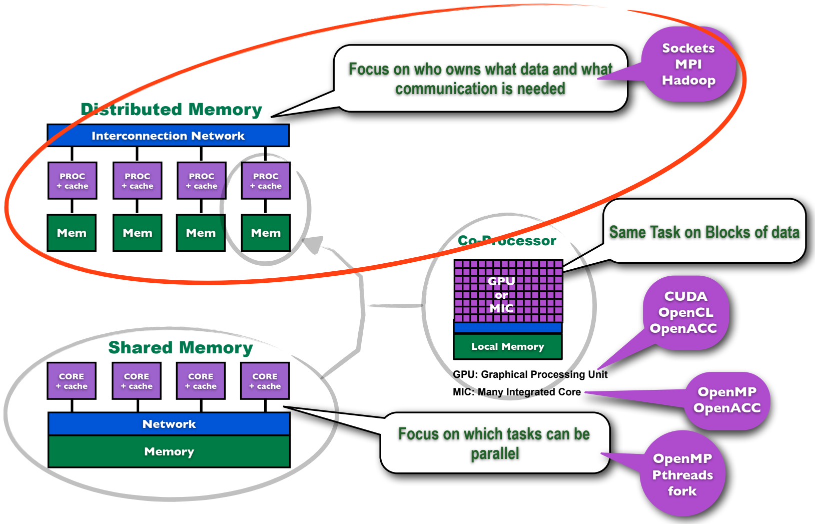
Knowing the Right Words



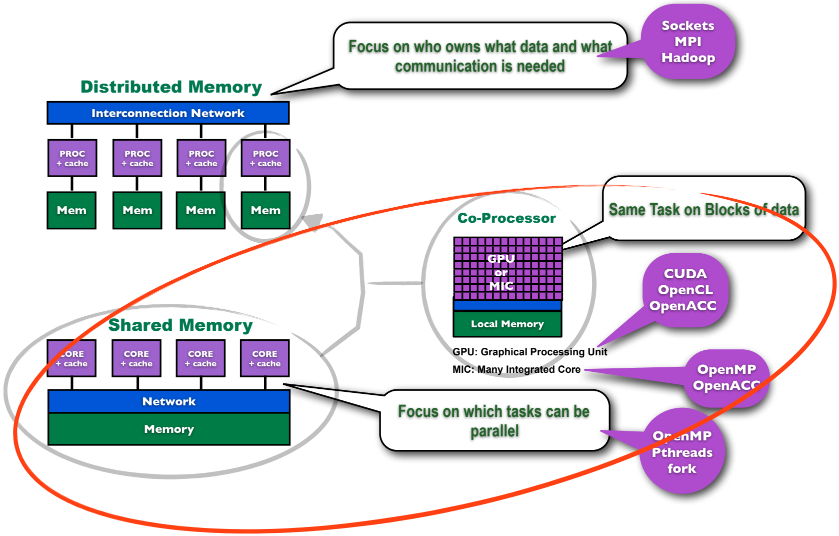
"Native" Programming Models and Tools



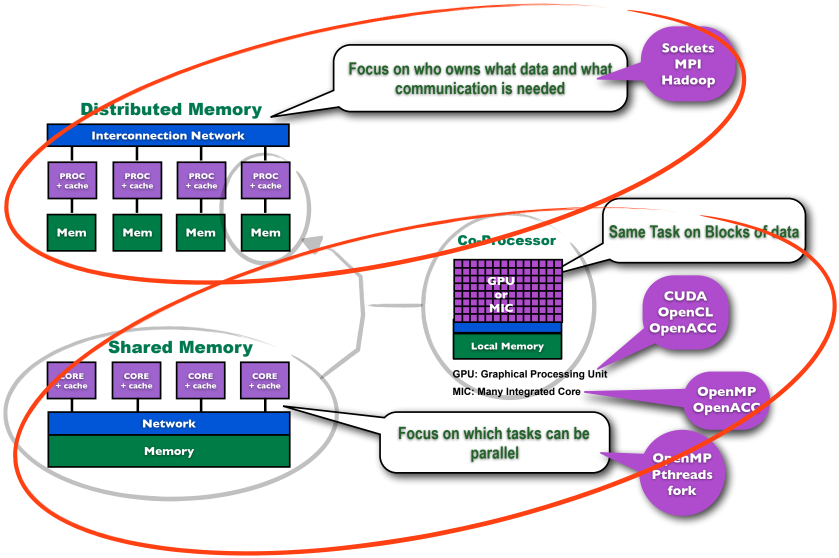
30+ Years of Parallel Computing Research



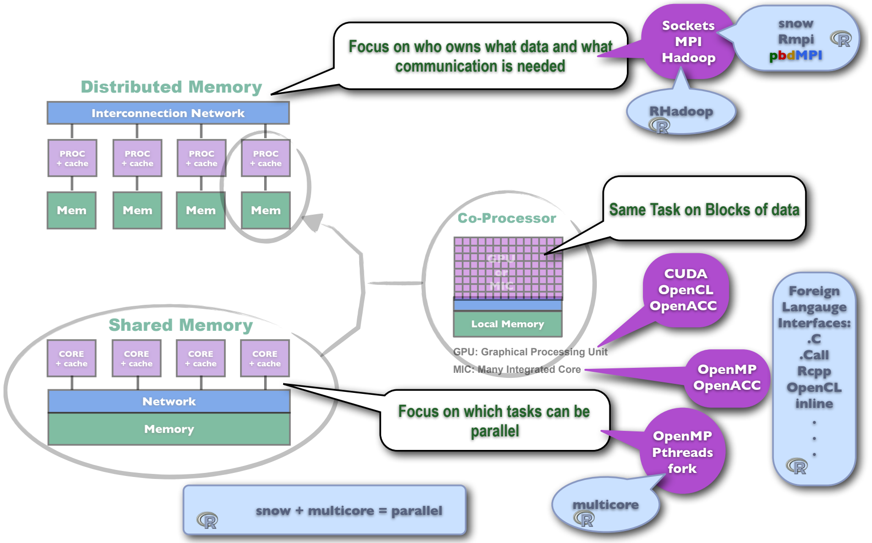
Last 10 years of Advances



Putting It All Together Challenge



R Interfaces to Native Tools


Focus on who owns what data and what communication is needed
Sockets MPI Hadoop
snow Rmpi pbdMPI
RHadoop
Distributed Memory
Interconnection Network
PROC + cache
Mem
Co-Processor
GPU or MIC
Local Memory
GPU: Graphical Processing Unit
MIC: Many Integrated Core
Same Task on Blocks of data
CUDA OpenCL OpenACC
OpenMP OpenACC
Foreign Langauge Interfaces: .C .Call Rcpp OpenCL inline . . .
Shared Memory
CORE + cache
Network
Memory
Focus on which tasks can be parallel
OpenMP Pthreads fork
multicore
snow + multicore = parallel

Summary

- Three flavors of hardware
  - Distributed is stable
  - Multicore and co-processor are evolving
  - Two memory models
  - Distributed works in multicore
- Parallelism hierarchy
- Medium to big machines have all three

Contents

Performance and Accuracy

Sometimes $\pi = 3.14$ is (a) infinitely faster than the "correct" answer and (b) the difference between the "correct" and the "wrong" answer is meaningless. . . . The thing is, some specious value of "correctness" is often irrelevant because it doesn't matter. While performance almost always matters. And I absolutely detest the fact that people so often dismiss performance concerns so readily.

— Linus Torvalds, August 8, 2008

Why Profile?

- Because performance matters.
- Bad practices scale up!
- Your bottlenecks may surprise you.
- Because R is dumb.
- R users claim to be data people. . . so act like it!

Compilers often correct bad behavior. . .

A Really Dumb Loop

```c
int main(){
    int x, i;
    for (i=0; i<10; i++)
        x = 1;
    return 0;
}
```

clang -O3 example.c

```
main:
        .cfi_startproc
# BB#0:
        xorl    %eax,
            %eax
        ret
```

clang example.c

```
main:
        .cfi_startproc
# BB#0:
        movl    $0, -4(%rsp)
        movl    $0, -12(%rsp)
.LBB0_1:
        cmpl    $10, -12(%rsp)
        jge     .LBB0_4
# BB#2:
        movl    $1, -8(%rsp)
# BB#3:
        movl    -12(%rsp), %eax
        addl    $1, %eax
        movl    %eax, -12(%rsp)
        jmp     .LBB0_1
.LBB0_4:
        movl    $0, %eax
        ret
```

R will not!

Dumb Loop

```
for (i in 1:n){
  tA <- t(A)
  Y <- tA %*% Q
  Q <- qr.Q(qr(Y))
  Y <- A %*% Q
  Q <- qr.Q(qr(Y))
}

Q
```

Better Loop

```
tA <- t(A)

for (i in 1:n){
  Y <- tA %*% Q
  Q <- qr.Q(qr(Y))
  Y <- A %*% Q
  Q <- qr.Q(qr(Y))
}

Q
```

Example from a Real R Package

Exerpt from Original function

```
while(i<=N){
  for(j in 1:i){
     d.k <- as.matrix(x)[l==j,l==j]
       ...
```

Exerpt from Modified function

```
x.mat <- as.matrix(x)

while(i<=N){
  for(j in 1:i){
     d.k <- x.mat[l==j,l==j]
       ...
```

By changing just 1 line of code, performance of the main method improved by **over 350%**!

Some Thoughts

- R is slow.
- Bad programmers are slower.
- R isn't very clever (compared to a compiler).
- The Bytecode compiler helps, but not nearly as much as a compiler.

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.

Other Profiling Tools

- perf (Linux)
- PAPI
- MPI profiling: fpmpi, mpiP, TAU

Profiling MPI Codes with **pbdPROF**

1. Rebuild pbdR packages

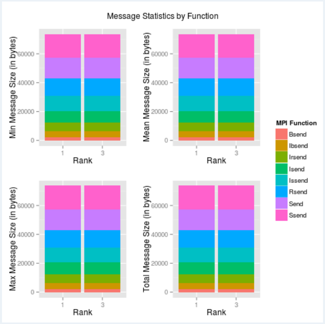
```
R CMD INSTALL pbdMPI_0.2-1.tar.gz \
    --configure-args= \
    "--enable-pbdPROF"
```

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**

- Performance Application Programming Interface
- High and low level interfaces
- Linux only :(

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

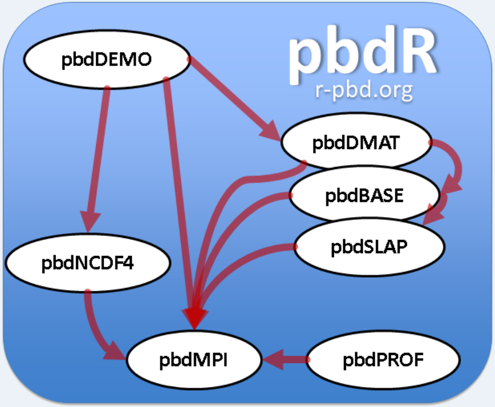
Programming with Big Data in R (pbdR)

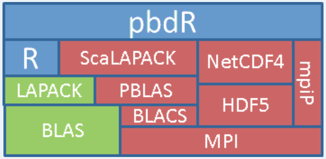
Striving for *Productivity, Portability, Performance*

- *Free*[a] R packages.
- Bridging high-performance compiled code with high-productivity of R
- 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

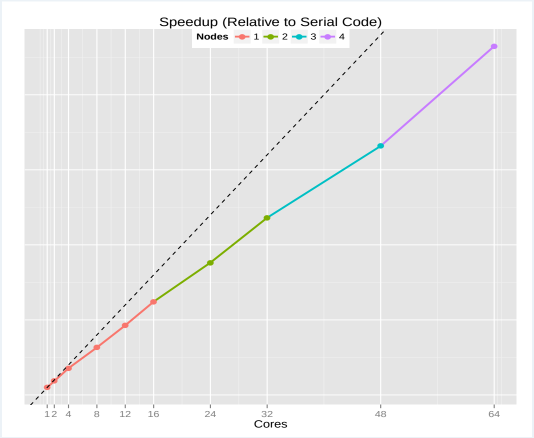
```
allreduce(x)
```

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**

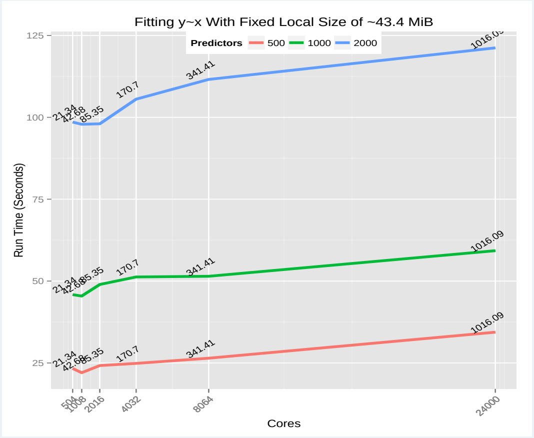
Matrix Exponentiation



```r
library(pbdDMAT)

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

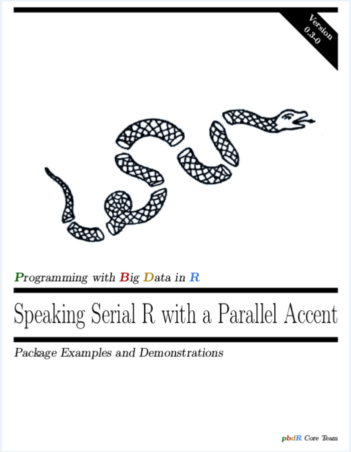
Distributed Matrices and Statistics with **pbdDMAT**

Least Squares Benchmark



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

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


Version 0.3-0

Programming with Big Data in R

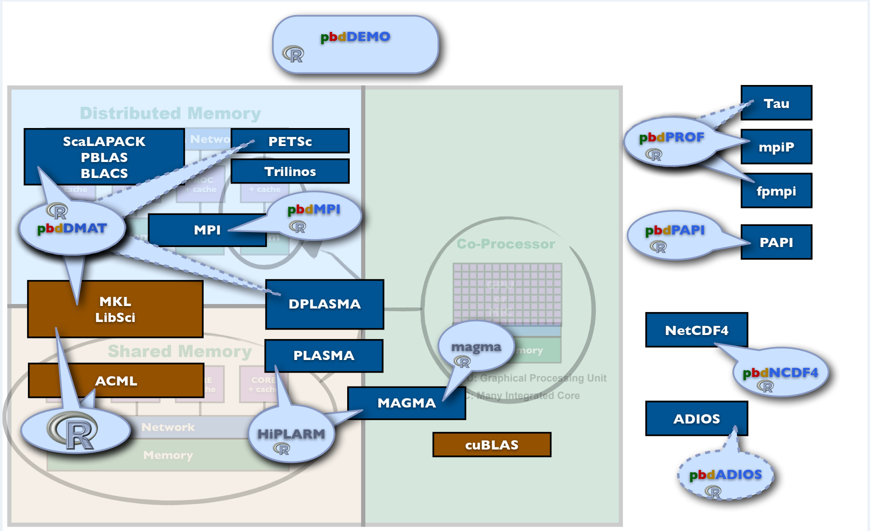
Speaking Serial R with a Parallel Accent

Package Examples and Demonstrations

pbdR Core Team

- 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

R and pbdR Interfaces to HPC Libraries



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

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

```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

```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
```

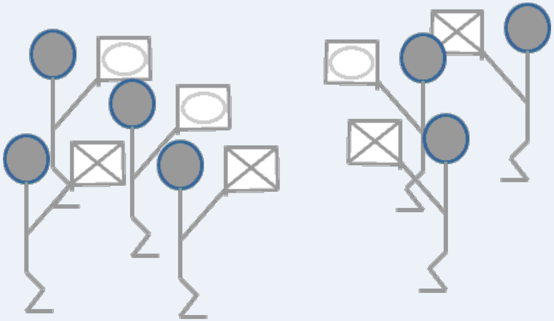
Sample Output:

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

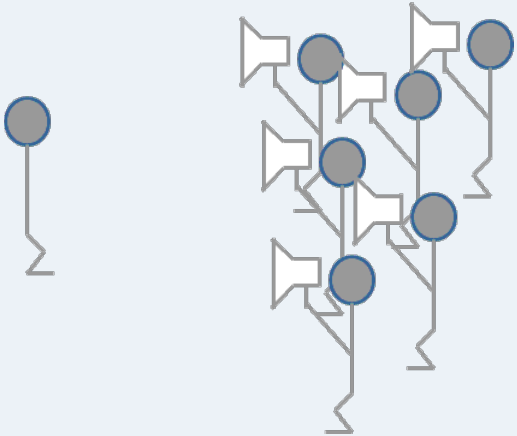
MPI Operations

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

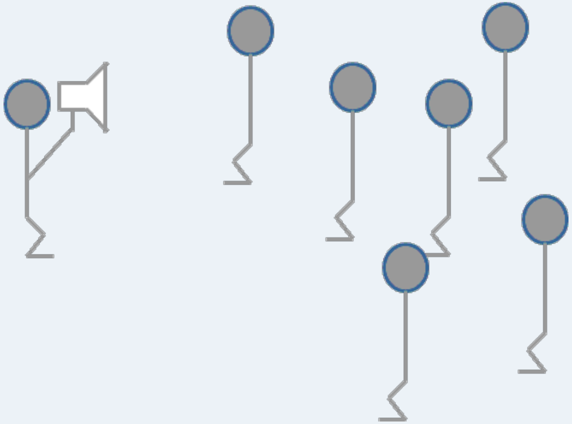
Reductions — Combine results into single result



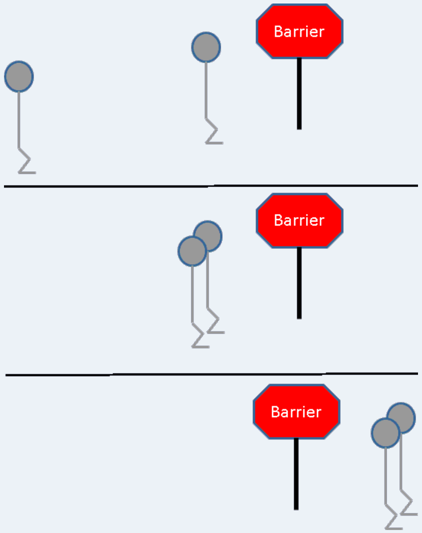
Gather — Many-to-one



Broadcast — One-to-many



Barrier — Synchronization



MPI Operations (2 of 2)

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

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

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

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

Quick Example 3

Reduce and Gather: 3_gt.r

```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

```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
```

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:

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

- Always initialize before starting and finalize when finished:

```r
init()

# ...

finalize()
```

Contents

Distributing Data

Problem: How to distribute the data

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

?

Distributing a Matrix Across 4 Processors: Block Distribution

Data

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

Processors

0
1
2
3

Distributing a Matrix Across 4 Processors: Local Load Balance

Data

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

Processors

0

1

2

3

The GBD Data Structure

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

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

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

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

Understanding GBD: Global Matrix

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

$$\text{Processors} = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5$$

Understanding GBD: Load Balanced GBD

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

$$\text{Processors} = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5$$

Understanding GBD: Local View

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

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

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

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

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

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

Processors = 0 1 2 3 4 5

Understanding GBD: Non-Balanced GBD

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

$$\text{Processors} = \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5$$

Understanding GBD: Local View

$$\left[\begin{array}{ccccccccc} \end{array}\right]_{0\times 9}$$

$$\left[\begin{array}{ccccccccc} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & x_{17} & x_{18} & x_{19} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} & x_{27} & x_{28} & x_{29} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & x_{37} & x_{38} & x_{39} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} & x_{47} & x_{48} & x_{49} \end{array}\right]_{4\times 9}$$

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

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

$$\left[\begin{array}{ccccccccc} \end{array}\right]_{0\times 9}$$

$$\left[\begin{array}{ccccccccc} x_{81} & x_{82} & x_{83} & x_{84} & x_{85} & x_{86} & x_{87} & x_{88} & x_{89} \\ x_{91} & x_{92} & x_{93} & x_{94} & x_{95} & x_{96} & x_{97} & x_{98} & x_{99} \end{array}\right]_{2\times 9}$$

$$\text{Processors} = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5$$

Summary

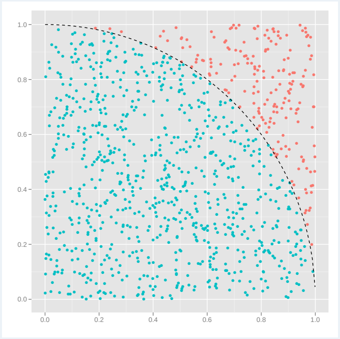
- Need to distribute your data? Try splitting by row.
- May not work well if your data is square (or longer than tall).

Contents

Example 1: Monte Carlo Simulation

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

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



Example 1: Monte Carlo Simulation GBD Algorithm

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

Example 1: Monte Carlo Simulation Code

Serial Code

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

Parallel Code

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

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

finalize()
```

Note

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

Example 2: Sample Covariance

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

Example 2: Sample Covariance GBD Algorithm

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

Example 2: Sample Covariance Code

Serial Code

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

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

comm.print(Cov.X)
```

Example 3: Linear Regression

Find $\boldsymbol{\beta}$ such that

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

When X is full rank,

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

Example 3: Linear Regression GBD Algorithm

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

Example 3: Linear Regression Code

Serial Code

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

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

Parallel Code

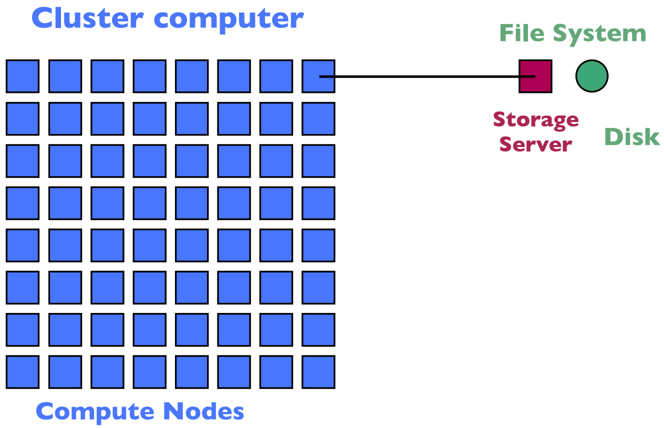
```r
tX.gbd <- t(X.gbd)
A <- allreduce(tX.gbd %*% X.gbd, op = "sum")
B <- allreduce(tX.gbd %*% y.gbd, op = "sum")

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

Summary

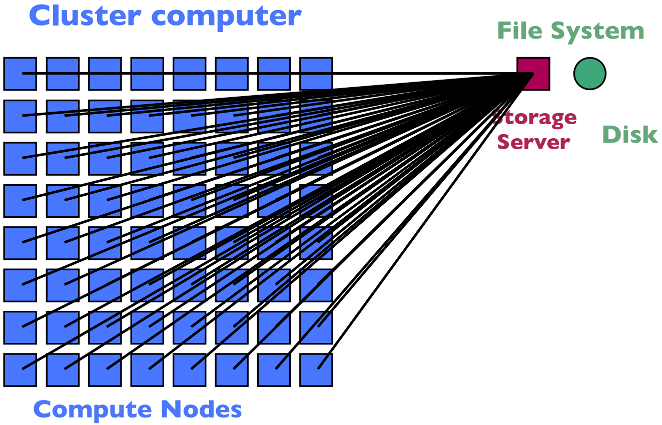
- SPMD programming is (often) a natural extension of serial programming.
- More **pbdMPI** examples in **pbdDEMO**.

Contents

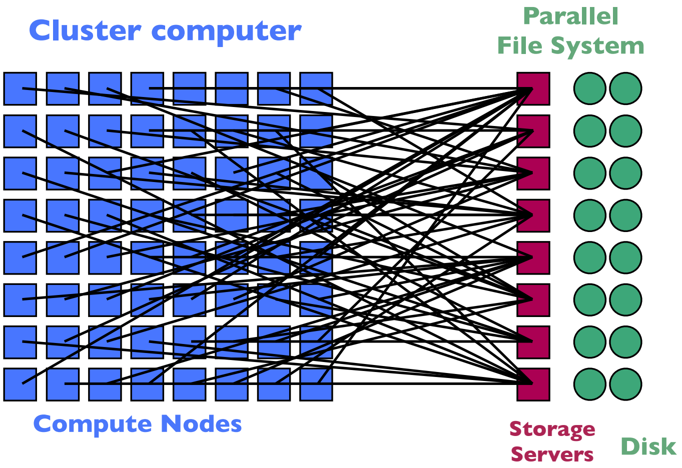
One Node to One Storage Server



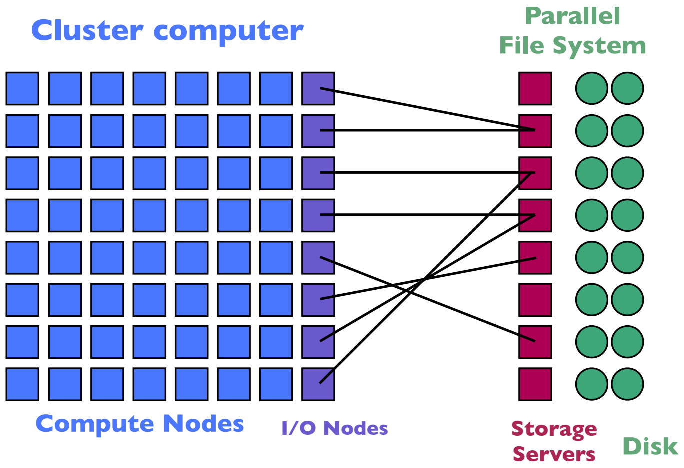
Many Nodes to One Storage Server



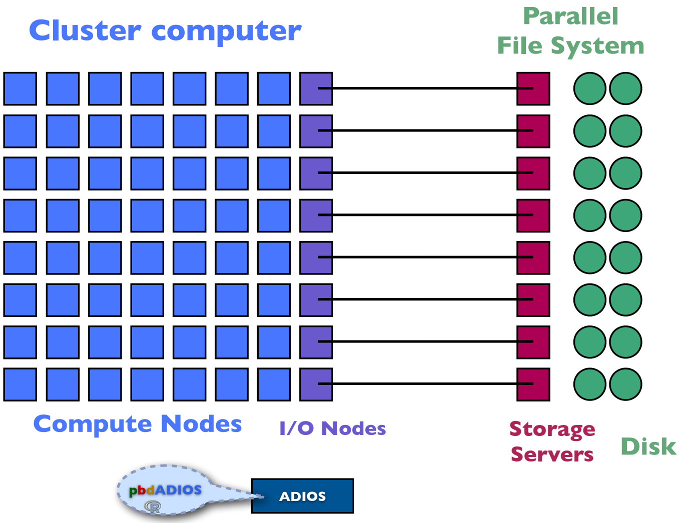
Many Nodes to Few Storage Servers



Few Nodes to Few Storage Servers - Default



Few Nodes to Few Storage Servers - Coordinated



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

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

CSV Data: Read Serial then Distribute

Listing:

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

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

New Issues

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

CSV Data

Serial Code

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

x
```

Parallel Code

```r
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

```r
## 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

```r
## 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
X <- redistribute(X, bldim=c(2, 2), ICTXT=0)
Xprc <- prcomp(X)
```

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()
```

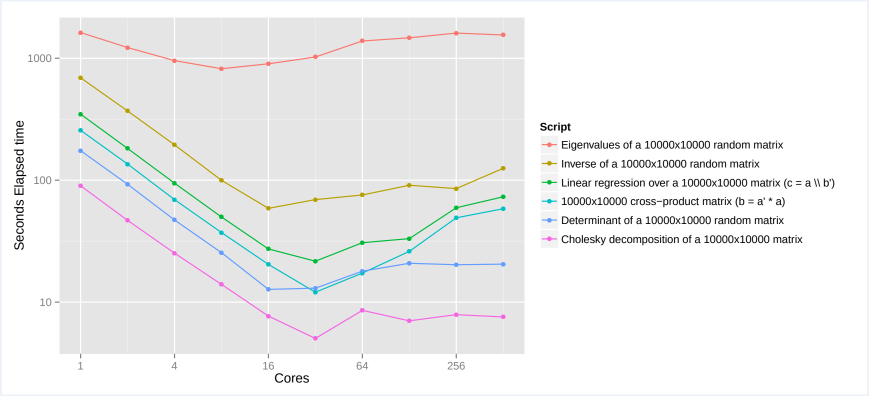
Summary

- 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

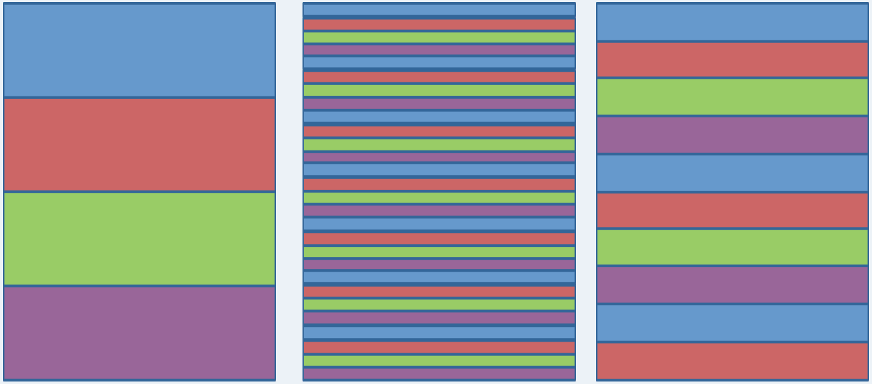
Distributed Matrices

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



The solution is to distribute the data.

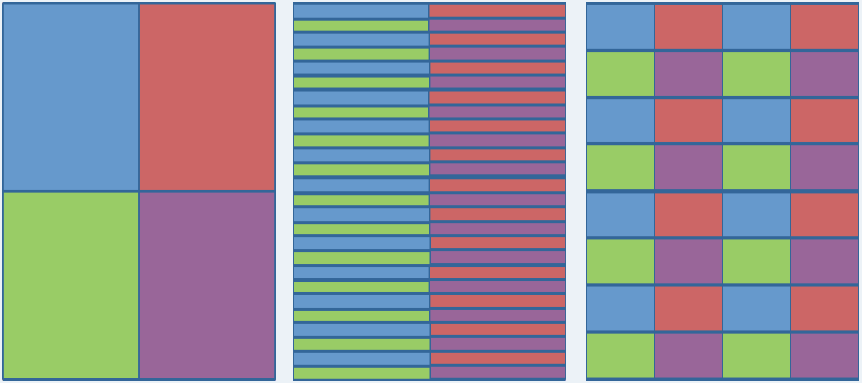
Distributed Matrices



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

Figure: Matrix Distribution Schemes

Distributed Matrices



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

Figure: Matrix Distribution Schemes Onto a 2-Dimensional Grid

Processor Grid Shapes

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

(a) 1×6

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

(b) 2×3

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

(c) 3×2

$$\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

(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`: 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

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

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

ddmatrix: 2-dimensional Row Cyclic

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

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

ddmatrix: 2-dimensional Block-Cyclic

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

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

The ddmatrix Data Structure

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

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

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

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

Understanding `ddmatrix`: Local View

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

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

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

The `ddmatrix` Data Structure

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

---

3. `ddmatrix` can be row-contiguous or not, depending on the processor grid and blocking factor used.
4. `ddmatrix` is locally column-major and globally, it depends. . .

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

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

Pros and Cons of This Data Structure

Pros

- Robust for matrix computations.

Cons

- Confusing layout.

This is why we hide most of the distributed details.

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

Methods for class `ddmatrix`

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

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

Serial Code

```r
cov(x)
```

Parallel Code

```r
cov(x)
```

Comparing pbdMPI and pbdDMAT

pbdMPI:

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

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.

Summary

1. Start by loading the package:

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

2. Always initialize before starting and finalize when finished:

```r
init.grid()

# ...

finalize()
```

Contents

Randomized SVD[1]

```
PROTOTYPE FOR RANDOMIZED SVD
Given an $m \times n$ matrix $A$, 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 $U\Sigma V^*$, where $U$ and $V$ are orthonormal, and $\Sigma$ is
nonnegative and diagonal.
Stage A:
1  Generate an $n \times 2k$ Gaussian test matrix $\Omega$.
2  Form $Y = (AA^*)^q A\Omega$ by multiplying alternately with $A$ and $A^*$.
3  Construct a matrix $Q$ whose columns form an orthonormal basis for
   the range of $Y$.
Stage B:
4  Form $B = Q^*A$.
5  Compute an SVD of the small matrix: $B = \widetilde{U}\Sigma V^*$.
6  Set $U = Q\widetilde{U}$.
Note: The computation of $Y$ in step 2 is vulnerable to round-off errors.
When high accuracy is required, we must incorporate an orthonormalization
step between each application of $A$ and $A^*$; see Algorithm 4.4.
```

```
ALGORITHM 4.4: RANDOMIZED SUBSPACE ITERATION
Given an $m \times n$ matrix $A$ and integers $\ell$ and $q$, this algorithm computes an
$m \times \ell$ orthonormal matrix $Q$ whose range approximates the range of $A$.
1  Draw an $n \times \ell$ standard Gaussian matrix $\Omega$.
2  Form $Y_0 = A\Omega$ and compute its QR factorization $Y_0 = Q_0 R_0$.
3  for $j = 1, 2, \dots, q$
4      Form $\widetilde{Y}_j = A^* Q_{j-1}$ and compute its QR factorization $\widetilde{Y}_j = \widetilde{Q}_j \widetilde{R}_j$.
5      Form $Y_j = A\widetilde{Q}_j$ and compute its QR factorization $Y_j = Q_j R_j$.
6  end
7  $Q = Q_q$.
```

Serial R

```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]
  }
```

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

Randomized SVD

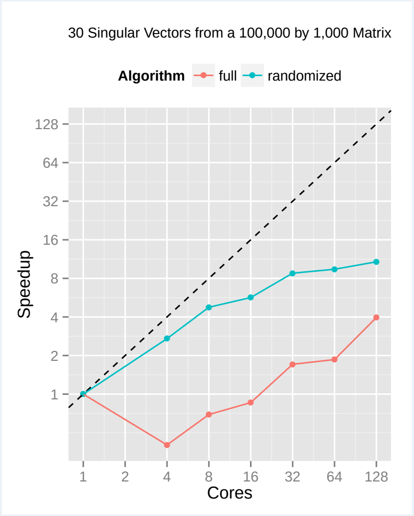
Serial R

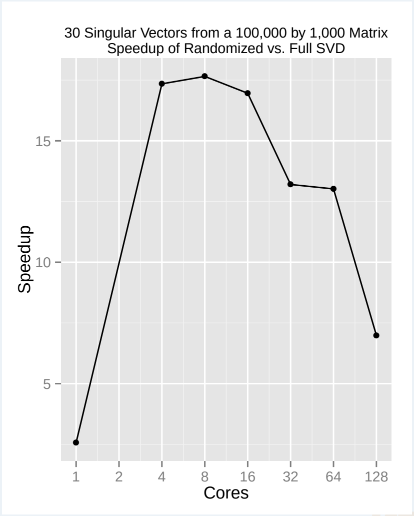
```
randSVD <- function(A, k, q=3)
  {
    ## Stage A
    Omega <-  matrix(rnorm(n*2*k),
                nrow=n, ncol=2*k)
    Y <- A %*% Omega
    Q <- qr.Q(qr(Y))
    At <- t(A)
    for(i in 1:q)
      {
        Y <- At %*% Q
        Q <- qr.Q(qr(Y))
        Y <- A %*% Q
        Q <- qr.Q(qr(Y))
 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

    ## 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





Summary

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

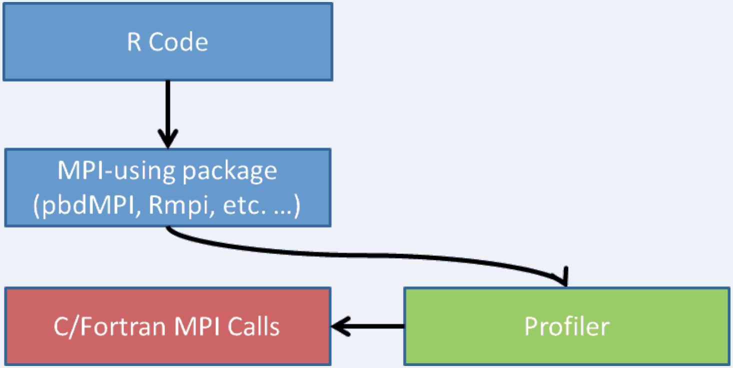
Contents

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

```
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: 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

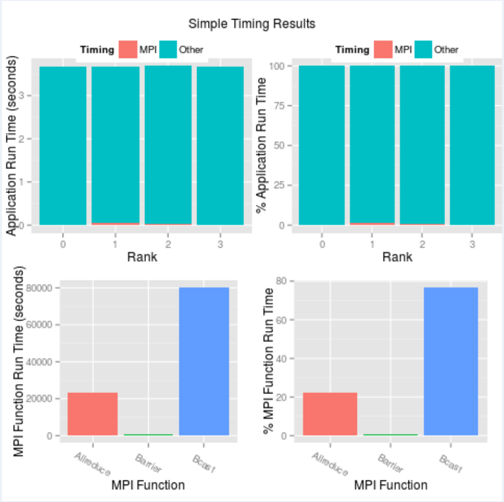
```r
library(pbdPROF)
prof.data <- read.prof("R.4.28812.1.mpiP")
```

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
```

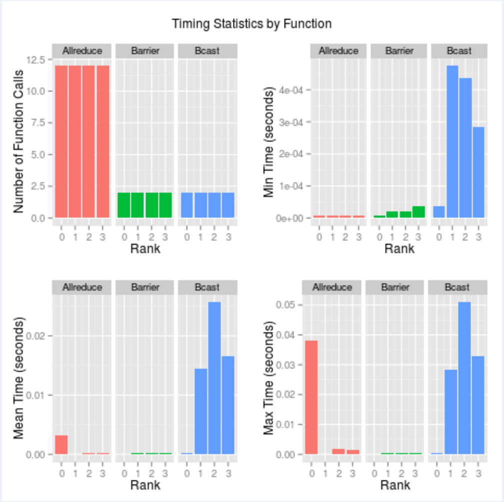
Generate plots

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



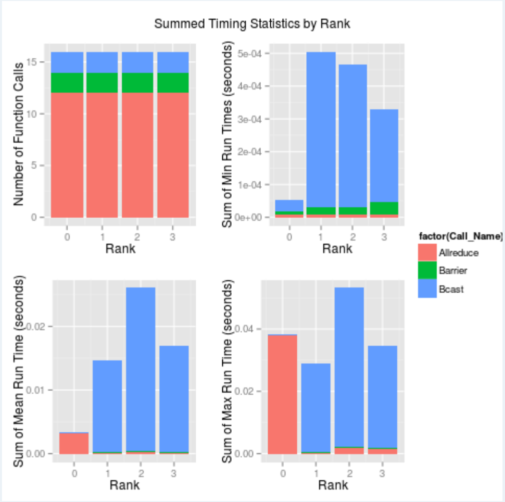
Generate plots

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



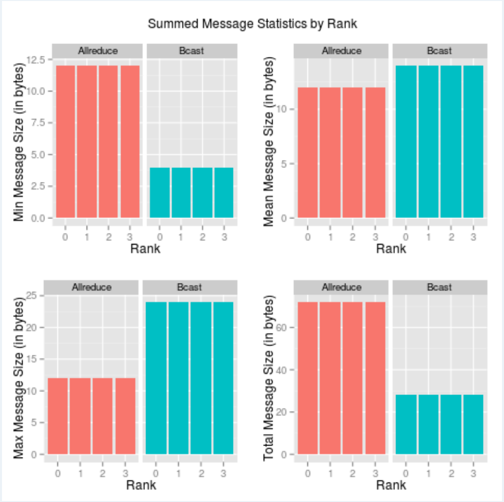
Generate plots

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



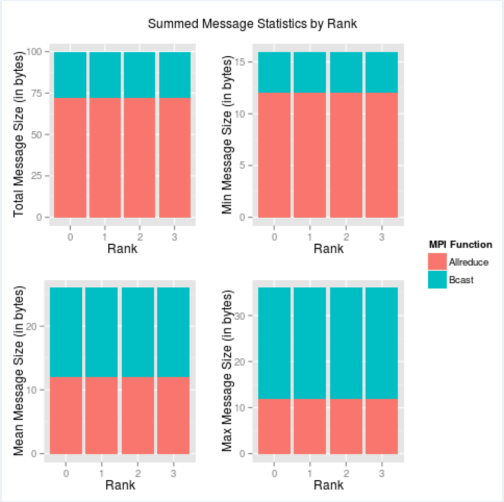
Generate plots

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



Generate plots

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



Summary

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

Contents

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/

Come see our poster on Wednesday at 5:30!