Programming with Big Data in R

Drew Schmidt and George Ostrouchov

July 8, 2013

Affiliations and Support

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

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

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

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

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

About This Presentation

Downloads

This presentation and supplemental materials are available at:

http://r-pbd.org/user2013

About This Presentation

Speaking Serial R with a Parallel Accent

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

https://github.com/wrathematics/pbdDEMO/blob/master/inst/doc/pbdDEMO-guide.pdf?raw=true

It contains more examples, and sometimes added detail.

About This Presentation

Installation Instructions

Installation instructions for setting up a pbdR environment are available:

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

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

About This Presentation

Conventions

We use:

- "." as a decimal mark
- "," as order of magnitude separator

Example	Yes	No
One million	1,000,000	1.000.000
One half	0.5	0,5
One thousand and one half	1,000.5	1.000,5

Contents

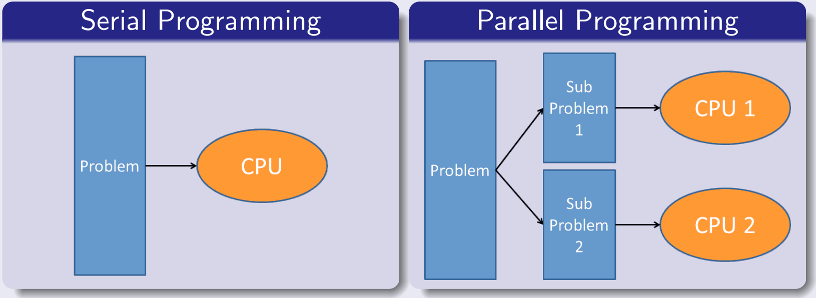
Contents

What is Parallelism?

Broadly, *doing more than one thing at a time*.

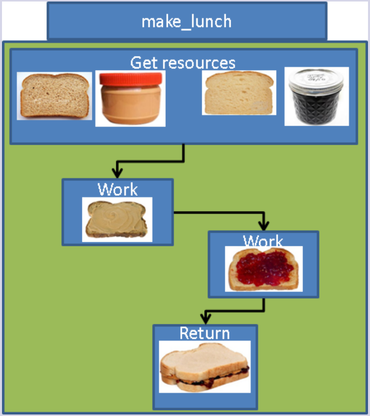
The simultaneous use of multiple compute resources to solve a computational problem:

Parallelism

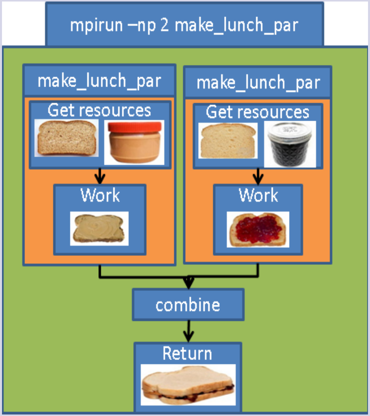


Parallelism

Serial Programming



Parallel Programming



Kinds of Parallelism

- *Data Parallelism*: Data is distributed
- *Task Parallelism*: Tasks are distributed

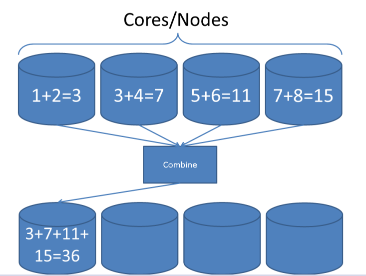
pbdR Paradigms: Data Parallelism

With data parallelism:

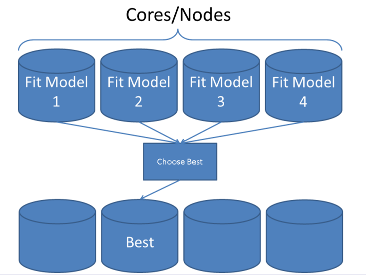
- No one processor/node owns all the data.
- Processors own local pieces of a (conceptually) global object

Data vs Task Parallelism

Data Parallelism



Task Parallelism



Difficulty

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.

Scalability

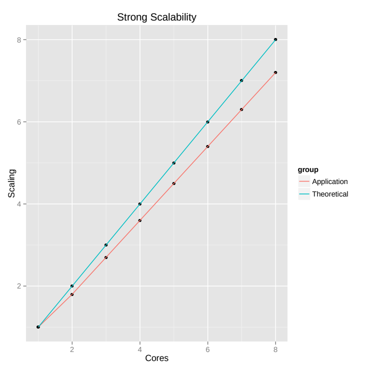
Scalability: unitless measure of performance;

$$\frac{\tau_i}{\tau_0}$$

Types of Scalability: Strong and Weak

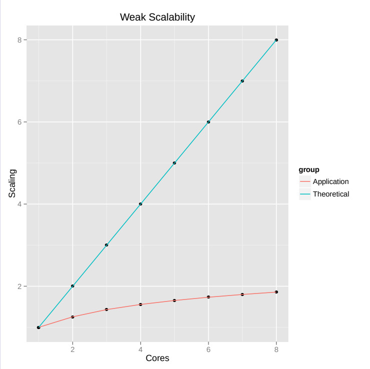
Strong

Fix *total* data size



Weak

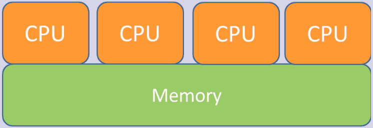
Fix *local* data size



Shared and Distributed Memory Machines

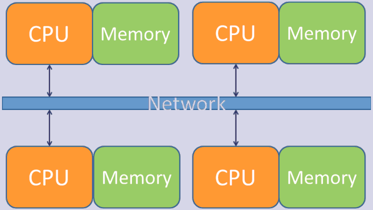
Shared Memory

Different processors can directly access and modify each others' memory. There is only one node.



Distributed

Different processors/nodes can not directly access/modify different processors'/nodes' memory.



Shared and Distributed Memory Machines

Shared Memory Machines

Thousands of cores

Nautilus, University of Tennessee

1024 cores

Distributed Memory Machines

Hundreds of thousands of cores

Kraken, University of Tennessee

112,896 cores

R and Parallelism

What about R?

Problems with Serial R

1. Slow.
2. If you don't know what you're doing, it's *really* slow.
3. Performance improvements usually for small machines.
4. Very ram intensive.
5. Chokes on big data.

Parallel R Packages

Shared Memory

1. **foreach**
2. **parallel**
3. **snow**
4. **multicore**

Distributed

1. **Rmpi**
2. R+Hadoop
3. **pbdR**

R and Parallelism

The solution to many of R's problems is parallelism. However ...

What we have

1. Mostly serial.
2. Mostly not distributed
3. Data parallelism mostly explicit

What we want

1. Mostly parallel.
2. Mostly distributed.
3. Mostly implicit.

Why We Need Parallelism

1. Saves time (long term).
2. Data size is skyrocketing.
3. Necessary for many problems.
4. Like it or not, it's coming.
5. *It's really cool.*

Contents

Programming with Big Data in R (pbdR)

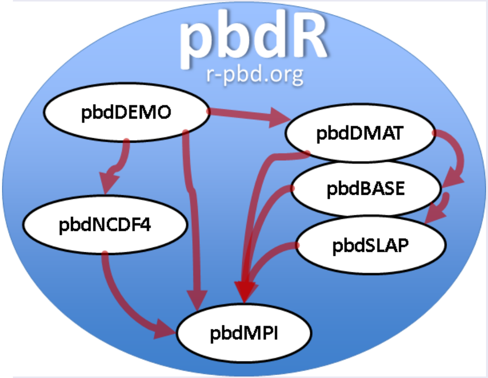
Striving for *Productivity, Portability, Performance*

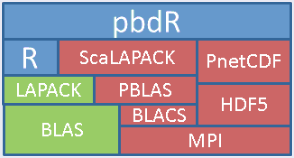
Programming with Big Data in R

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

[a] MPL, BSD, and GPL licensed

pbdR Packages





pbdR Packages — http://code.r-pbd.org

Released to CRAN:

- **pbdMPI**: MPI bindings (explicit, low-level)
- **pbdSLAP**: Foreign library (just install it, nothing to use)
- **pbdBASE**: Compiled code (used by DMAT, also for devs)
- **pbdDMAT**: Distributed matrices (mostly implicit, high-level)
- **pbdNCDF4**: Parallel NetCDF4 reader
- **pbdDEMO**: Package demonstrations, examples, vignette written in textbook style

Future Development:

- Updates and expansions
- Profiling Tools for Parallel Computing with R
- . . .

Example Syntax

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

Look familiar?

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

pbdR Paradigms

Programs that use pbdR are utilize:

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

And generally utilize:

- Data Parallelism

Batch Execution

- Non-interactive
- Use

```
Rscript my_script.r
```

or

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

- In parallel:

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

Single Program/Multiple Data (SPMD)

- SPMD is a programming *paradigm*.
- Not to be confused with SIMD.
- SPMD utilizes MIMD architecture computers.
- Arguably the simplest extension of serial programming.
- Difficult to describe, easy to do. . .
- Only one program is written, executed in batch on all processors.
- Different processors are autonomous; there is no manager.
- The dominant programming model for large machines.

SPMD

Manager/Worker

SPMD

Contents

Message Passing Interface (MPI)

- *MPI*: Standard for managing communications (data and instructions) between different nodes/computers.
- *Implementations*: OpenMPI, MPICH2, Cray MPT, ...
- Enables parallelism (via communication) on distributed machines.
- *Communicator*: manages communications between processors.

MPI Operations (1 of 2)

- **Managing a Communicator**: Create and destroy communicators.
  `init()` — initialize communicator
  `finalize()` — shut down communicator(s)

- **Rank query**: determine the processor's position in the communicator.
  `comm.rank()` — "who am I?"
  `comm.size()` — "how many of us are there?"

- **Printing**: Printing output from various ranks.
  `comm.print(x)`
  `comm.cat(x)`
  WARNING: only use these functions on *results*, never on yet-to-be-computed things.

Quick Example 1

Rank Query: 1_rank.r

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

comm.print("Hello, world")

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

finalize()
```

Execute this script via:

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

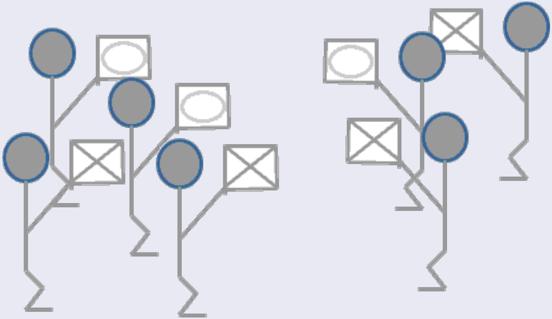
Sample Output:

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

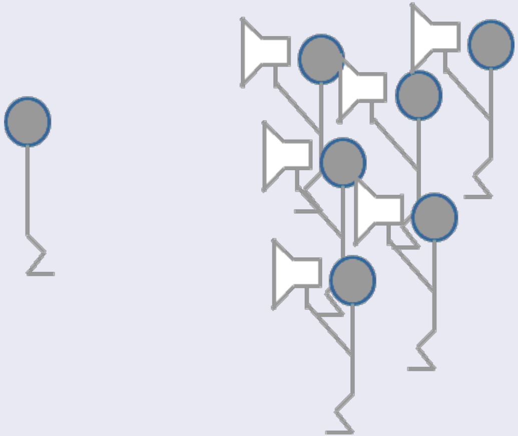
MPI Operations

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

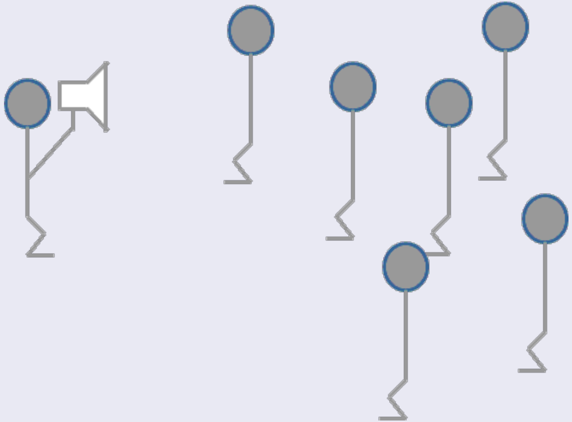
Reductions — Combine results into single result



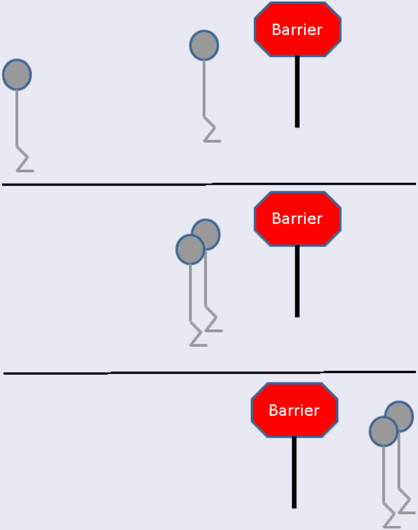
Gather — Many-to-one



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, quiet = 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, quiet=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
```

MPI Package Controls

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

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

Quick Example 5

Barrier: 5_barrier.r

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

.SPMD.CT$msg.barrier <- TRUE
.SPMD.CT$print.quiet <- TRUE

for (rank in 1:comm.size()-1){
  if (comm.rank() == rank){
    cat(paste("Hello", rank+1, "of", comm.size(), "\n"))
  }
  barrier()
}

comm.cat("\n")

comm.cat(paste("Hello", comm.rank()+1, "of",
    comm.size(), "\n"), all.rank=TRUE)

finalize()
```

Execute this script via:

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

Sample Output:

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

Random Seeds

pbdMPI offers a simple interface for managing random seeds:

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

Quick Example 6

Timing: 6_timer.r

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

comm.set.seed(diff=T)

test <- function(timed)
{
  ltime <- system.time(timed)[3]

  mintime <- allreduce(ltime, op='min')
  maxtime <- allreduce(ltime, op='max')
  meantime <- allreduce(ltime, op='sum')/comm.size()

  return(data.frame(min=mintime, mean=meantime,
      max=maxtime))
}

times <- test(rnorm(1e6)) # ~7.6MiB of data
comm.print(times)

finalize()
```

Execute this script via:

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

Sample Output:

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

Other Helper Tools

pbdMPI Also contains useful tools for Manager/Worker and task parallelism codes:

- **Task Subsetting**: Distributing a list of jobs/tasks
  `get.jid(n)`
- ***ply**: Functions in the *ply family.
  `pbdApply(X, MARGIN, FUN, ...)` — analogue of `apply()`
  `pbdLapply(X, FUN, ...)` — analogue of `lapply()`
  `pbdSapply(X, FUN, ...)` — analogue of `sapply()`

Quick Comments for Using pbdMPI

1. Start by loading the package:

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

2. Always initialize before starting and finalize when finished:

```r
init()

# ...

finalize()
```

Contents

Distributing a Matrix Across 4 Processors: Block Distribution

Data	Processors
$$x = \left[\begin{array}{ccc} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \\ \hline x_{4,1} & x_{4,2} & x_{4,3} \\ x_{5,1} & x_{5,2} & x_{5,3} \\ x_{6,1} & x_{6,2} & x_{6,3} \\ \hline x_{7,1} & x_{7,2} & x_{7,3} \\ x_{8,1} & x_{8,2} & x_{8,3} \\ x_{9,1} & x_{9,2} & x_{9,3} \\ \hline x_{10,1} & x_{10,2} & x_{10,3} \end{array}\right]_{10\times 3}$$	0 1 2 3

Distributing a Matrix Across 4 Processors: Local Load Balance

Data	Processors

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

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

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

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

Understanding GBD: Local View

$$\left[\ \right]_{0\times 9}$$

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

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

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

$$\left[\ \right]_{0\times 9}$$

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

Processors = 0 1 2 3 4 5

Quick Comment for GBD

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

Brief Intermission

Brief Intermission

Questions? Comments?

Don't forget to talk to us at our discussion group:
http://group.r-pbd.org/

If you have an affiliation at a United States institution (university, research lab, etc.), consider getting an allocation with us:
http://www.nics.tennessee.edu/getting-an-allocation

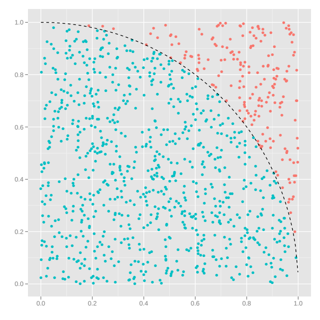
Come to the talk *Elevating R to Supercomputers*, Friday, July 12th at 10:00 at the High Performance Computing session

Contents

Example 1: Monte Carlo Simulation

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

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



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(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

```r
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

```r
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 β such that

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

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

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

Example 3: Linear Regression GBD Algorithm

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

Example 3: Linear Regression Code

Serial Code

```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
```

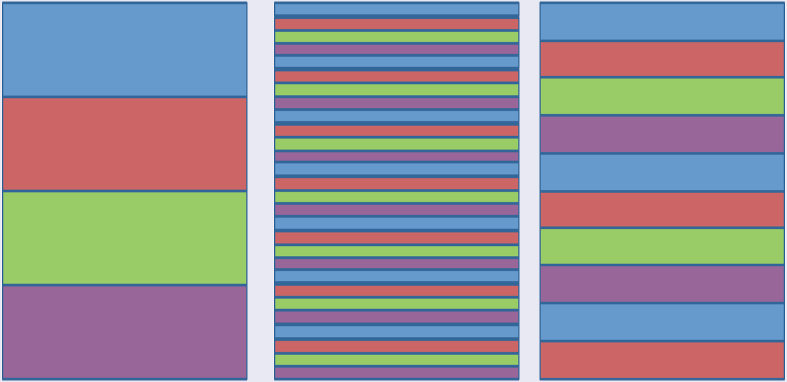
Contents

Distributed Matrices

Most problems in data science are matrix algebra problems

- Data structure: block-cyclic matrix distributed across a 2-dimensional grid of processors.
- No single processor should hold all of the data.
- Very robust, but very confusing data structure.

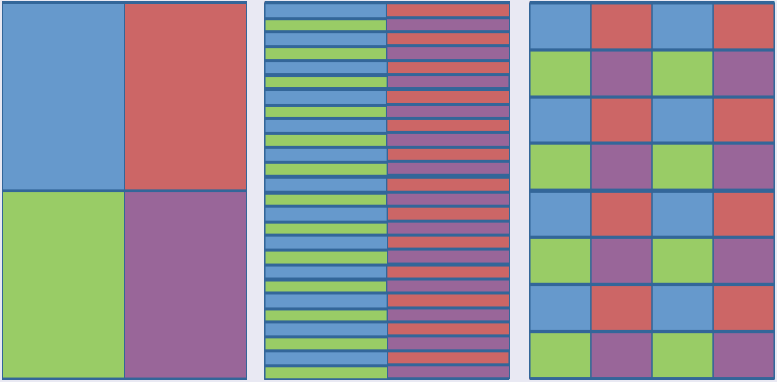
Distributed Matrices



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

Figure: Matrix Distribution Schemes

Distributed Matrices



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

Figure: Matrix Distribution Schemes Onto a 2-Dimensional Grid

Processor Grid Shapes

(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

Distributed Matrices

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

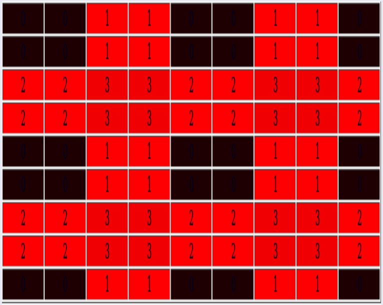
$$\texttt{ddmatrix} = \begin{cases} \textbf{Data} & \text{S4 local submatrix, an R matrix} \\ \textbf{dim} & \text{S4 dimension of the global matrix, a numeric pair} \\ \textbf{ldim} & \text{S4 dimension of the local submatrix, a numeric pair} \\ \textbf{bldim} & \text{S4 ScaLAPACK blocking factor, a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

with prototype

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

Distributed Matrices: The Data Structure

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



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

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

Understanding Dmat: Global Matrix

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

DMAT: 1-dimensional Row Block

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

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

DMAT: 2-dimensional Row Block

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

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

DMAT: 1-dimensional Row Cyclic

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

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

DMAT: 2-dimensional Row Cyclic

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

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

DMAT: 2-dimensional Block-Cyclic

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

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

The DMAT Data Structure

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

DMAT: 2-dimensional Block-Cyclic with 6 Processors

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

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

Understanding DMAT: Local View

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

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

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

The DMAT Data Structure

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

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

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

Pros and Cons of This Data Structure

Pros

- Fast for distributed matrix computations

Cons

- Literally everything else

This is why we hide most of the distributed details.

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

Distributed Matrix Methods

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

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

Serial Code

```
cov(x)
```

Parallel Code

```
cov(x)
```

Comparing pbdMPI and pbdDMAT

- **pbdMPI** is MPI + some sugar.
- The `GBD` data structure is not the only thing **pbdMPI** can handle (just a useful convention).
- **pbdDMAT** is more of a software package.
- The block-cyclic `DMAT` structure *must* be used for **pbdDMAT**.

Quick Comments for Using pbdDMAT

1. Start by loading the package:

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

2. Always initialize before starting and finalize when finished:

```r
init.grid()

# ...

finalize()
```

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

Sample Covariance

Serial Code

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

Parallel Code

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

Linear Regression

Serial Code

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

ols <- solve(A) %*% B

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

Parallel Code

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

ols <- solve(A) %*% B

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

Quick Example 3

PCA: pca.r

```
library(pbdDMAT, quiet=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", quiet=T)
comm.cat("%Cols:", i/dim(x.dmat)[2], "\n\n", quiet=T)

finalize()
```

Execute this script via:

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

Sample Output:

```
Cols: 221
%Cols: 0.884
```

Generating Random Data

Using randomly generated matrices is the best way to "get your feet wet" with the pbd tools. You can do this in 2 ways:

1. Generate a global matrix and distribute it.
2. Generate locally only what is needed.

Example 1: Random Distributed Matrix Generation

Generate a global matrix and distribute it

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

# Common global on all processors --> distributed
comm.set.seed(diff=FALSE)
x <- matrix(rnorm(100), nrow=10, ncol=10)
x.dmat <- as.ddmatrix(x)

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

finalize()
```

Example 2: Random Distributed Matrix Generation

Generate locally only what is needed

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

comm.set.seed(diff = TRUE) # good seeds via rlecuyer
x.dmat <- ddmatrix("rnorm", nrow=10, ncol=10)

finalize()
```

Example 3: Random Distributed Matrix Generation

Generate locally only what is needed

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

zero.dmat <- ddmatrix(0, nrow=100, ncol=100)
id.dmat <- diag(1, nrow=100, ncol=100)

finalize()
```

Example 4: Random Distributed Matrix Generation

Convert between GBD and DMAT

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

comm.set.seed(diff = TRUE)

N.gbd <- 1 + comm.rank()
X.gbd <- matrix(rnorm(N.gbd * 3), ncol = 3)

# convert GBD to DMAT
X.dmat <- gbd2dmat(X.gbd)

# convert DMAT to GBD
new.X.gbd <- dmat2gbd(X.dmat)

# undistribute
X <- as.matrix(X.dmat)

finalize()
```

Distributed Matrices

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

Contents

Where to Learn More

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

Thanks for coming!

Questions? Comments?

Don't forget to come to the talk:

Elevating R to Supercomputers
Friday, July 12th at 10:00
at the High Performance Computing session!