Programming with Big Data in R Workshop I
Introduction and Basics

Whoever

Whenever, 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

"Course Notes"

The content of this presentation is largely based on the **pbdDEMO** vignette:

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

It contains more examples, and sometimes more detail. Loosely, it is this presentation's lecture notes.

About This Presentation

Conventions

- We use "." as a decimal mark, not ",". E.g., "one thousand and one half" is written "1,000.5", not "1.000,5".
- We will use special suffixes to denote distributed objects (ones not stored entirely on a single processor).
  `.spmd` denotes a distributed object, while
  `.dmat` denotes a distributed object which is of class `ddmatrix`
  No suffix means the object is global (common to all processors)

  Neither of these suffices carries semantic meaning.

Contents

1 Introduction

2 Introduction to MPI

3 pbdMPI Examples

4 Brief Intermission

5 Introduction to pbdDMAT

6 pbdDMAT Examples

7 Dénouement

Contents

1 Introduction
- Problems with R and a Concise Introduction to Parallelism
- The pbdR Project
- pbdR Focus and Paradigms

Problems with 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.

What is Parallelism?

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

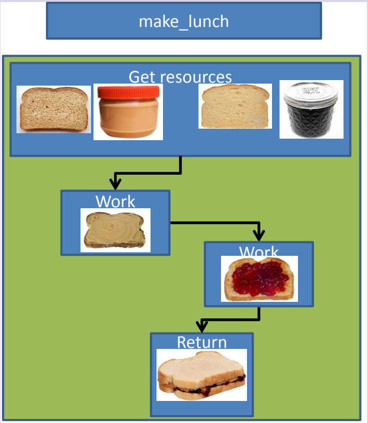
- *Task Parallelism*: Many really small tasks.
  e.g. Make one sandwich for every person on earth to eat.
- *Data Parallelism*: One really big task.
  e.g. Make one sandwich so large that every person on earth could eat from it.

More Common Terms

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

(Data) Parallelism

Serial Programming



make_lunch

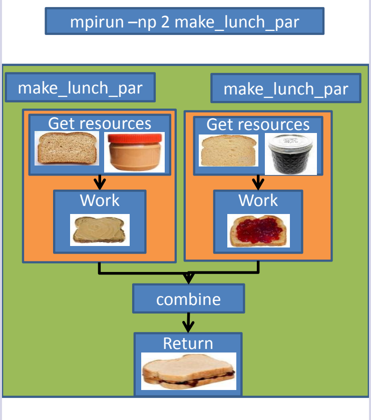
Get resources

Work

Work

Return

Parallel Programming



mpirun –np 2 make_lunch_par

make_lunch_par

make_lunch_par

Get resources

Get resources

Work

Work

combine

Return

R and Parallelism

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

What we have

1. Mostly serial.
2. Parallelism mostly not distributed (foreach, parallel/snow/multicore, . . .)
3. Data parallelism mostly explicit (Rmpi, R+Hadoop, . . .)

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.*

Programming with Big Data in R (pbdR)

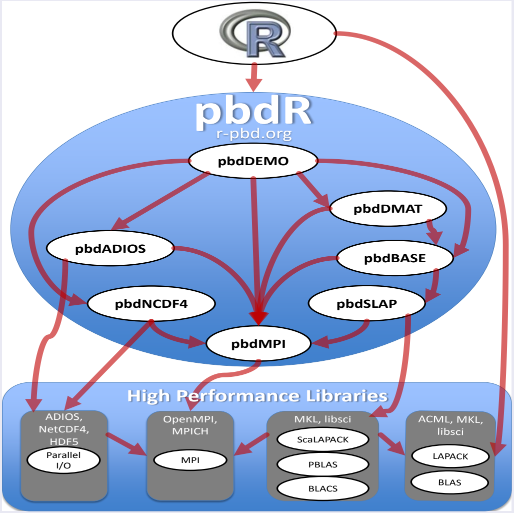
Goals: *Productivity, Portability, Performance*

Our Approach:

- Series of *free*[a] R packages.
- Enables SPMD style programming.
- Scalable, big data analytics with high-level syntax.
- Implicit management of distributed data details.
- Methods have syntax *identical* to R.
- Powered by state of the art numerical libraries (MPI, ScaLAPACK, PBLAS, BLACS, LAPACK, BLAS, . . .)

[a] GPL and BSD licensed

pbdR Packages



pbdR Packages

- **pbdMPI**: MPI bindings (explicit, low-level)
- **pbdSLAP**: Foreign library (just install it, nothing to use)
- **pbdBASE**/**pbdDMAT**: Distributed matrices (mostly implicit, high-level)
- **pbdNCDF4**: Parallel NetCDF4 reader (mostly implicit, mid-level)
- **pbdADIOS**: Interface to ADIOS I/O middleware (mostly explicit, low-level)
- **pbdDEMO**: Package demonstrations, examples, lengthy vignette

Beginners should focus on **pbdDEMO**, **pbdMPI**, and **pbdDMAT**

Example Syntax

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

Look familiar?

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

pbdR Focus: Distributed 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

pbdR Paradigms

Programs that use pbdR are meant to utilize the:

- Data Parallelism method
- Single Program/Multiple Data (SPMD) style

pbdR Paradigms: Data Parallelism

With data parallelism:

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

pbdR Paradigms: SPMD

- Natural extension of writing serial codes.
- Different from Manager/Worker.
- No one processor is in charge. Each thinks it's the boss ("it's like academia").
- One program written, executed independently by all processors.
- Each processor owns a local sub-piece of data from the (conceptual) whole.

Manager/Worker vs SPMD

Graphics will go here

Manager/Worker: Fascism

SPMD: Democracy

Contents

2 Introduction to MPI
- MPI Basics
- pbdMPI vs Rmpi

Message Passing Interface (MPI)

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

Common 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?"

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

Quick Example 1

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

myRank <- comm.rank() + 1 # comm index starts at 0, not 1
print(myRank)

finalize()
```

Common MPI Operations (2 of 2)

- **Reduction**: each processor has a number `x.spmd`; add all of them up, find the largest/smallest, ...
  `reduce(x.spmd, op='sum')` — only one processor gets result
  `allreduce(x.spmd, op='sum')` — every processor gets result
- **Gather**: each processor has a number; create a new object on some processor containing all of those numbers.
  `gather(x.spmd)` — only one processor gets result
  `allgather(x.spmd)` — every processor gets result
- **Broadcast**: one processor has a number `x.spmd` that every other processor should also have.
  `bcast(x.spmd)`

Quick Example 2

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

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

sm <- allreduce(n) # default op is 'sum'
print(sm)

gt <- allgather(n)
print(gt)

finalize()
```

pbdMPI Sugar

- **Print**: printing with control over which processor prints.
  `comm.print(x, ...)`

- **Apply**: *ply-like functions.
  `pbdApply(X, MARGIN, FUN, ...)` — analogue of `apply()`
  `pbdLapply(X, FUN, ...)` — analogue of `lapply()`
  `pbdSapply(X, FUN, ...)` — analogue of `sapply()`

For more details, see the **pbdMPI** Reference Manual:
http://goo.gl/9oFRd

Quick Example 3

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

n <- 100
x <- split((1:n) + n * comm.rank(), rep(1:10, each = 10))
sm <- pbdLapply(x, sum)
comm.print(unlist(sm))

finalize()
```

pbdMPI vs **Rmpi**: Overview

- (+) **pbdMPI** is easier to install than **Rmpi**
- (+) **pbdMPI** is easier to use than **Rmpi**
- (+) **pbdMPI** has *way* better documentation and examples than **Rmpi**.
- (+) **pbdMPI** can often outperform **Rmpi**
- (+) **pbdMPI** integrates with the rest of pbd
- (−) **Rmpi** can be used with **foreach** via **doMPI**
- (−) **Rmpi** can be used in the master/worker paradigm

pbdMPI vs Rmpi: Syntax

Rmpi

```
# integer data
mpi.allreduce(x, type = 1)

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

pbdMPI

```
# whatever
allreduce(x)
```

Think That's Not a Problem?

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

pbdMPI vs **Rmpi**: Performance

We compared[a] the performance between **Rmpi** and **pbdMPI** in an allgather() operation on a for 10000 × 10000 distributed matrix

Cores	**Rmpi**	**pbdMPI**	Speedup
32	24.6	6.7	3.67
64	25.2	7.1	3.55
128	22.3	7.2	3.10
256	22.4	7.1	3.15

Table: Runtimes (seconds) for allgather()

[a] D. Schmidt, G. Ostrouchov, W.-C. Chen, and P. Patel. *Tight coupling of R and distributed linear algebra for high-level programming with big data.* SC Companion: High Performance Computing, Networking Storage and Analysis. IEEE Computer Society, 2012.

Contents

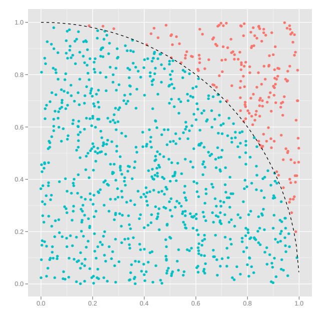
3 pbdMPI Examples

- Monte Carlo Simulation
- Linear Regression
- Clustering

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 SPMD Algorithm

1. Let n be big-ish; we'll take $n = 1000$.
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 $n \times nprocs$
6. If my rank is 0, print the result.

Example 1: Monte Carlo Simulation Code

```
N.spmd <- 1000
X.spmd <- matrix(runif(N.spmd * 2), ncol = 2)
r.spmd <- sum(rowSums(X.spmd^2) <= 1)
ret <- allreduce(c(N.spmd, r.spmd), op = "sum")
PI <- 4 * ret[2] / ret[1]
comm.print(PI)
```

Example 1: Monte Carlo Simulation Batch Execution

Locate the **pbdDEMO** example script `monte_carlo.r` and execute:

```
### At the shell prompt, run the demo with 4 processors
### Use Rscript.exe for Windows systems
mpirun -np 4 Rscript monte_carlo.r
```

Sample output:

```
COMM.RANK = 0
[1] 3.171
```

Example 2: 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 2: Linear Regression SPMD Algorithm

1. Compute $tx = x^T$
2. Compute $A = tx \times x$. Ask everyone else what they got for this and sum all the answers up.
3. Compute $B = tx \times yx$. Ask everyone else what they got for this and sum all the answers up.
4. Compute $A^{-1} \times B$

Example 2: Linear Regression Code

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

solve(matrix(A, ncol = ncol(X.spmd))) %*% B
```

Example 2: Linear Regression Batch Execution

Locate the **pbdDEMO** example script `ols.r` and execute:

```
### At the shell prompt, run the demo with 4 processors
### Use Rscript.exe for Windows systems
mpirun -np 4 Rscript ols.r
```

Sample output:

```
COMM.RANK = 0
          [,1]
[1,] 0.9652591
[2,] 2.0166145
```

Example 3: Clustering

Brief Intermission

Brief Intermission

Questions? Comments?

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

Contents

5 Introduction to pbdDMAT
- Distributed Matrices
- Generating Data
- Reading Distributed Matrices

Distributed Matrices

- `ddmatrix`: distributed analogue of R's `matrix` class.
- No single processor holds all of the data (unless you messed up)

Distributed Matrices: The Data Structure

`ddmatrix` is an S4 class object containing a block-cyclically distributed data onto a 2-dimensional processor grid.

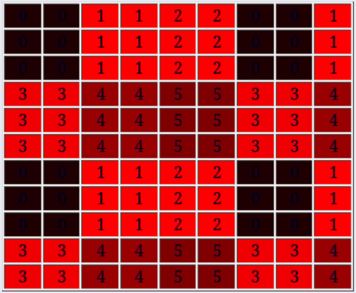
$$\texttt{ddmatrix} = \begin{cases} \textbf{Data} & \text{S4 slot containing the object's submatrix, an R matrix} \\ \textbf{dim} & \text{S4 slot containing the dimension of the global matrix, a numeric pair} \\ \textbf{ldim} & \text{S4 slot containing the dimension of the local submatrix, a numeric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

with prototype

$$\texttt{new("ddmatrix")} = \begin{cases} \textbf{Data} & = \texttt{matrix(0)} \\ \textbf{dim} & = \texttt{c(1,1)} \\ \textbf{ldim} & = \texttt{c(1,1)} \\ \textbf{bldim} & = \texttt{c(1,1)} \\ \textbf{CTXT} & = 0 \end{cases}$$

Distributed Matrices: The Data Structure

Example: an 11 × 9 matrix is distributed with a “block-cycling” factor of 3 × 2 on a 2 × 3 processor grid:



0	0	1	1	2	2	0	0	1
0	0	1	1	2	2	0	0	1
0	0	1	1	2	2	0	0	1
3	3	4	4	5	5	3	3	4
3	3	4	4	5	5	3	3	4
3	3	4	4	5	5	3	3	4
0	0	1	1	2	2	0	0	1
0	0	1	1	2	2	0	0	1
0	0	1	1	2	2	0	0	1
3	3	4	4	5	5	3	3	4
3	3	4	4	5	5	3	3	4

$$= \begin{cases} \textbf{Data} & = \texttt{matrix(...)} \\ \textbf{dim} & = \texttt{c(11, 9)} \\ \textbf{ldim} & = \texttt{c(...)} \\ \textbf{bldim} & = \texttt{c(3, 2)} \\ \textbf{CTXT} & = 0 \end{cases}$$

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

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.

Distributed Matrix Methods

The **pbdBASE** and **pbdDMAT** packages have nearly 100 (and counting) methods with identical syntax to core R, including:

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

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:

- Global matrix → distributed matrix
- Generate locally only what is needed

Example 1 : Random Distributed Matrix Generation

```r
# Common global --> distributed
set.seed(1234)
x <- matrix(rnorm(100), nrow=10, ncol=10)
dx <- as.ddmatrix(x)

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

Example 2 : Random Distributed Matrix Generation

```r
# Using pbdDEMO
comm.set.seed(diff = TRUE) # good seeds via rlecuyer
dx <- Hnorm(dim=c(10, 10))
```

Distributed Matrices

- .

Contents

6 pbdDMAT Examples
- Compression with Principal Components Analysis
- Predictions with Linear Regression

Example 1: PCA

Compute the principal components of a distributed matrix. Retain only a subset of the rotated data, the greatest number of columns which will retain no more than 90% of the variation of the original dataset.

Example 1: PCA SPMD Algorithm

1. Set good random seed and generate $10,000 \times 250$ `ddmatrix`
2. Compute PCA rotation with scaling using `prcomp()`.
3. Determine the first *i* columns which retain no more than 90% of the original variation.
4. Retain only the first *i* columns of the rotated data.

Example 1: PCA Code

```
n <- 1e4
p <- 250

comm.set.seed(diff=T)
dx <- Hnorm(dim=c(n, p), bldim=c(4,4), mean=100, sd=25)

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

new_dx <- pca$x[, 1:i]
```

Example 1: PCA Batch Execution

Locate the **pbdDEMO** example script pca.r and execute:

```
### At the shell prompt, run the demo with 4 processors
### Use Rscript.exe for Windows systems
mpirun -np 4 Rscript pca.r
```

Sample output:

```
DENSE DISTRIBUTED MATRIX
---------------------------
@Data: -9.81e-01, -8.39e-01, -1.33e-01,  6.33e-02, ...
Process grid: 2x2
Global dimension: 10000x221
(max) Local dimension: 5000x112
Blocking: 4x4
BLACS CTXT: 0


Number of columns retained:  221
Percentage of columns retained: 0.884
```

Example 2: Regression

Fit the linear model $\mathbf{y} = \mathbf{X}\beta + \epsilon$ and make a prediction on new x data using this model.

Example 2: Regression SPMD Algorithm

1. Set good random seed and generate 1250 × 40 `ddmatrix` x and 1250 × 1 `ddmatrix` y
2. Fit the linear model using `lm.fit()`.
3. Generate new x data.
4. Compute the estimated $\hat{y} = x_{\text{new}} * \beta$.

Example 2: Regression Code

```
comm.set.seed(1234, diff=TRUE)
dx <- Hnorm(c(n, p), bldim=bldim, mean=mean, sd=sd)
dy <- Hunif(c(n, 1), bldim=bldim, min=ymin, max=ymax)

mdl <- lm.fit(dx, dy)

dx.new <- Hnorm(c(1, p), bldim=bldim, mean=mean, sd=sd)

pred <- dx.new %*% mdl$coefficients
```

Example 2: Regression Batch Execution

Locate the **pbdDEMO** example script `ols_dmat.r` and execute:

```
### At the shell prompt, run the demo with 4 processors
### Use Rscript.exe for Windows systems
mpirun -np 4 Rscript ols_dmat.r
```

Sample output:

```
The predicted y value is: 84.7432227923963
```

Contents

7 Dénouement

Where to Learn More

1. The **pbdDEMO** vignette: see http://r-pbd.org
2. Our Google Group: group.r-pbd.org

Thanks for coming!

Questions? Comments?

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