Programming with Big Data in R

Drew Schmidt
Remote Data Analysis and Visualization Center,
University of Tennessee, Knoxville

Wei-Chen Chen, PhD
Computer Science and Mathematics Division,
Oak Ridge National Laboratory

George Ostrouchov, PhD
Computer Science and Mathematics Division,
Oak Ridge National Laboratory

Pragneskumar Patel
Remote Data Analysis and Visualization Center,
University of Tennessee, Knoxville

Contents

Acknowledgement

Schmidt, Ostrouchov, and Patel 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.

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. This work used resources of the Newton HPC Program at the University of Tennessee, Knoxville.

We also thank Brian D. Ripley, Kurt Hornik, and Uwe Ligges from the R Core Team for discussing package release issues and helping us solve portability problems on different platforms.

Warning: This document is written to explain the main functions of **pbdDEMO** (Schmidt *et al.*, 2012b), version 0.1-0. Every effort will be made to ensure future versions are consistent with these instructions, but features in later versions may not be explained in this document.

Information about the functionality of this package, and any changes in future versions can be found on website: "Programming with Big Data in R" at `http://r-pbd.org/`.

Part I

Preliminaries

1

Introduction

This vignette is to explain some pbdR (Ostrouchov *et al.*, 2012) examples which are higher level applications and may be commonly found in basic Statistics. The purposes is to show how to reuse the pre-exist functions, and quickly solve problems in an efficient way. The functions built for examples may not be exactly same idea of original R (R Core Team, 2012) functions, but can be adjusted in similar wa. You are very welcome to use these as templates and rewrite your own functions or packages.

We presume that the reader already has an idea about SPMD programming. If not, please read **pbdMPI**'s vignette (Chen *et al.*, 2012b) first. Ideally, readers should run the demos of the **pbdMPI** package, and go through the code step by step.

1.1 Installation

One can download **pbdDEMO** from CRAN at `http://cran.r-project.org`, and the intallation can be done with the following commands

```
tar zxvf pbdDEMO_0.1-0.tar.gz
R CMD INSTALL pbdDEMO
```

Since **pbdEMO** depends on other **pbdR** packages, please read the corresponding vignettes if installations did not succeed. We also provide several demos for the capability of **pbdR** packages which are explained correspondingly in the next few sections.

1.2 List of Demos

Shell Script

```
### Under command mode, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
```

```
# ----------------- #
# II Ad Hoc Methods #
# ----------------- #

# Monte carlo simulation
mpiexec -np 4 Rscript -e "demo(monte_carlo,
   package=’pbdDMAT’,ask=F,echo=F)"
# Sample mean and variance
mpiexec -np 4 Rscript -e "demo(sample_stat,
   package=’pbdDMAT’,ask=F,echo=F)"
# Binning
mpiexec -np 4 Rscript -e "demo(binning, package=’pbdDMAT’,ask=F,echo=F)"
# Quantile
mpiexec -np 4 Rscript -e "demo(quantile,
   package=’pbdDMAT’,ask=F,echo=F)"
# OLS
mpiexec -np 4 Rscript -e "demo(ols, package=’pbdDMAT’,ask=F,echo=F)"

# -------------------- #
# III ddmatrix Methods #
# -------------------- #

# Random matrix generation
mpiexec -np 4 Rscript -e "demo(randmat_global,
   package=’pbdDMAT’,ask=F,echo=F)"
mpiexec -np 4 Rscript -e "demo(randmat_local,
   package=’pbdDMAT’,ask=F,echo=F)"
# SPMD to DMAT
mpiexec -np 4 Rscript -e "demo(spmd_dmat,
   package=’pbdDMAT’,ask=F,echo=F)"
# Parallel CSV read
mpiexec -np 4 Rscript -e "demo(read_csv,
   package=’pbdDMAT’,ask=F,echo=F)"
# Parallel SQL read
mpiexec -np 4 Rscript -e "demo(read_sql,
   package=’pbdDMAT’,ask=F,echo=F)"
```

2 Background

2.1 Notation

Note that we tend to use suffix `.spmd` for an object when we wish to indicate that the object is distributed. This is purely for pedagogical convenience, and has no semantic meaning. Since the code is written in SPMD style, you can think of such objects as referring to either a large, global object, or to a processor's local piece of the whole (depending on context). This is less confusing that it might first sound.

We will not use this suffix to denote a global object common to all processors. As a simple example, you could imagine having a large matrix with (global) dimensions $m \times n$ with each processor owning different collections of rows of the matrix. All processors might need to know the values for m and n; however, m and n do not depend on the local process, and so these do not receive the `.spmd` suffix. In many cases, it may be a good idea to invent an S4 class object and a corresponding set of methods. Doing so can greatly improve the usability and readability of your code, but is never strictly necessary. However, these constructions are the foundation of the **pbdBASE** (Schmidt *et al.*, 2012a) and **pbdDMAT** (Schmidt *et al.*, 2012c) packages.

On that note, depending on your requirements in distributed computing with R, it may be beneficial to you to use higher pbdR toolchain. If you need to perform dense matrix operations, or statistical analyses which depend heavily on matrix algebra (linear modeling, principal components analysis, . . .), then the **pbdBASE** and **pbdDMAT** packages are a natural choice. The major hurdle to using these tools is getting the data into the appropriate `ddmatrix` format, although we provide many tools to ease the pains of doing so. Learning how to use these packages can greatly improve code performance, and take your statistical modeling in R to previously unimaginable scales.

Again for the sake of understanding, we will at times append the suffix `.dmat` to objects of class `ddmatrix`. As with `.spmd`, this carries no semantic meaning, and is merely used to improve the readability of example code (especially when managing both "`.spmd`" and `ddmatrix` objects).

2.2 Timing Jobs

Measuring run time is a fundamental performance measure in computing. However, in parallel computing, not all “parallel components” (e.g. threads, or MPI processes) will take the same amount of time to complete a task, even when all tasks are given completely identical jobs. So measuring “total run time” begs the question, run time of what?

To help, we offer a timing function `demo.timer()` which can wrap segments of code much in the same way that `system.time()` does. However, the three numbers reported by `demo.timer()` are: (1) the minimum elapsed time measured across all processes, (2) the average elapsed time measured across all processes, and (3) the maximum elapsed time across all processes. The code for this function is listed below:

Timer Function

```
demo.timer <- function(timed)
{
  ltime <- system.time(timed)[3]
  barrier()

  mintime <- allreduce(ltime, op=’min’)
  maxtime <- allreduce(ltime, op=’max’)

  meantime <- allreduce(ltime, op=’sum’) / comm.size()

  return( c(min=mintime, mean=meantime, max=maxtime) )
}
```

Part II

Direct MPI Methods

3 Basic Statistics Examples

This section introduces four simple examples and explains a little about computing with distributed data. These implemented examples/functions are partly selected from the Cookbook of HPSC website (Chen and Ostrouchov, 2011) at `http://thirteen-01.stat.iastate.edu/snoweye/hpsc/?item=cookbook`. Please see more details there.

3.1 Monte Carlo Simulation

Example: compute a numerical approximation for π.

The demo command is

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(monte_carlo,'pbdDEMO',ask=F,echo=F)"
```

This is a simple Monte Carlo simulation example for numerically estimating π. Suppose we sample N uniform observations (x_i, y_i) inside (or perhaps on the border of) the unit square $(0, 1) \times (0, 1)$, where $i = 1, 2, \ldots, N$. Then

$$\pi \approx 4\frac{L}{N} \tag{3.1}$$

where $0 \leq L \leq N$ is the number of observations sampled satisfying

$$x_i^2 + y_i^2 \leq 1$$

The intuitive explanation for this is strategy which is sometimes given belies a misunderstanding of infinite cardinalities, and infinite processes in general. We are not *approximating* an area, because to do so with points would be madness requiring a transfinite process. Yes, indeed, something with areas is involved, but some care should be taken in the discussion as to what exactly is going on.

In reality, we are evaluating the probability that someone throwing a 0-dimensional "dart" at the unit square will have that "dart" also land below the arc of the unit circle contained within the unit square. Formally, let U_1 and U_2 be random uniform variables, each from the closed unit interval $[0, 1]$. Define the random variable

$$X := \begin{cases} 1, & U_1^2 + U_2^2 \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

Let $V_i = U_i^2$ for $i = 1, 2$. Then the expected value

$$\begin{aligned} E[X] &= P(V_1 + V_2 \leq 1) \\ &= \int_0^1 \int_0^{1-V_1} p(V_1, V_2) dV_2 dV_1 \\ &= \int_0^1 \int_0^{1-V_1} \left(\frac{1}{2\sqrt{V_1}}\right)\left(\frac{1}{2\sqrt{V_2}}\right) dV_2 dV_1 \\ &= \frac{1}{2}\int_0^1 \left(\frac{1-V_1}{V_1}\right)^{1/2} dV_1 \\ &= \frac{1}{2}\left[V_1\left(\frac{1-V_1}{V_1}\right)^{1/2} - \frac{1}{2}\arctan\left(\frac{\left(\frac{1-V_1}{V_1}\right)^{1/2}(2V_1 - 1)}{2(V_1 - 1)}\right)\right]_{V_1\to 0}^{V_1\to 1} \\ &= \frac{1}{2}\left[\frac{\pi}{4} + \frac{\pi}{4}\right] \end{aligned}$$

and by sampling observations X_i for $i = 1, \ldots, N$, by the Strong Law of Large Numbers

$$\bar{X}_N \xrightarrow{a.s.} \frac{\pi}{4} \quad \text{as } N \to \infty$$

In other words,

$$P\left(\lim_{N\to\infty} \bar{X}_N = \frac{\pi}{4}\right) = 1$$

Whence,

$$\frac{L}{N} \xrightarrow{a.s.} \frac{\pi}{4} \quad \text{as } N \to \infty$$

But because no one is going to read that, and if they do they'll just call me a grumpy old man, the misleading picture you desire can be found in Figure 3.1. And to everyone who found this looking for a homework solution, you're welcome.

The key step of the demo code is in the following block:

R Code

```r
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)
```

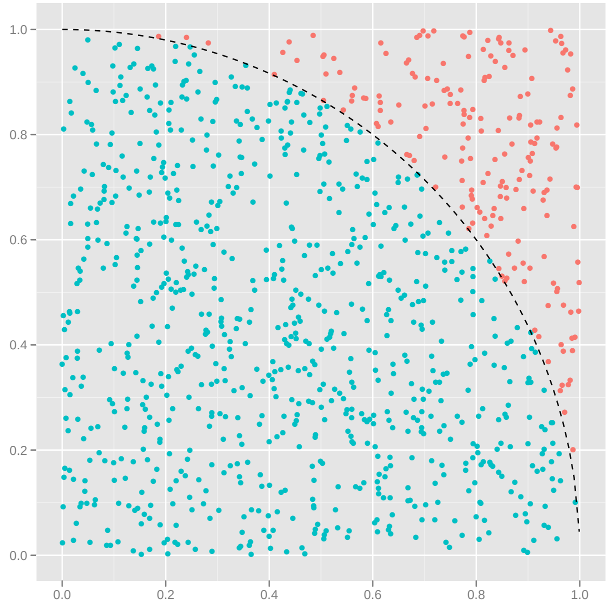

Figure 3.1: Approximating π by Monte Carlo methods

In line 1, we specify sample size in `N.spmd` for each processor, and $N = D \times$ `N.spmd` if D processors are executed. In line 2, we generate samples in `X.spmd` for every processor. In line 3, we compute how many of radios are less than or equal to 1 for each processors. In line 4, we call `allreduce` to obtain total numbers across all processors. In line 5, we use the Equation (3.1). Since SPMD, `ret` is common on all processors, and so is `PI`.

3.2 Sample Mean and Sample Variance

Example: compute sample mean/variance for distributed data.

The demo command is

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(sample_stat,'pbdDEMO',ask=F,echo=F)"
```

Suppose $\boldsymbol{x} = \{x_1, x_2, \ldots, x_N\}$ are observed samples, and N is potentially very large. We can distribute $\boldsymbol{x}$ in 4 processors, and each processor receives a proportional amount of data. One simple way to compute sample mean $\bar{x}$ and sample variance s_x is based on the formulas:

$$\begin{aligned}\bar{x} &= \frac{1}{N}\sum_{n=1}^{N} x_n \\ &= \sum_{n=1}^{N} \frac{x_n}{N} \qquad (3.2)\end{aligned}$$

and

$$\begin{aligned}s_x &= \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \bar{x})^2 \\ &= \frac{1}{N-1}\sum_{n=1}^{N} x_n^2 - \frac{2\bar{x}}{N-1}\sum_{n=1}^{N} x_n + \frac{1}{N-1}\sum_{n=1}^{N}\bar{x}^2 \\ &= \sum_{n=1}^{N}\left(\frac{x_n^2}{N-1}\right) - \frac{N\bar{x}^2}{N-1} \qquad (3.3)\end{aligned}$$

where expressions (3.2) and (3.3) are one-pass algorithms, which are potentially faster and more stable than the first expressions, especially for large N. Here, only the first and second moments are implemented, while the extension of one-pass algorithms to higher order moments is also possible.

The demo command is

```
### At the shell prompt, run the demo with 2 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 2 Rscript -e "demo(sample_stat,'pbdDEMO',ask=F,echo=F)"
```

The demo `sample_stat` generates fake data on 2 processors, then utilize `mpi.stat` function as

The demo `sample_stat` generates fake data on 4 processors, then utilizes `mpi.stat` function as

R Code

```
mpi.stat <- function(x.spmd){
  ### For mean(x).
  N <- allreduce(length(x.spmd), op = "sum")
  bar.x.spmd <- sum(x.spmd / N)
  bar.x <- allreduce(bar.x.spmd, op = "sum")

  ### For var(x).
  s.x.spmd <- sum(x.spmd^2 / (N - 1))
  s.x <- allreduce(s.x.spmd, op = "sum") - bar.x^2 * (N / (N -
     1))

  list(mean = bar.x, s = s.x)
} # End of mpi.stat().
```

where `allreduce` in **pbdMPI** (Chen *et al.*, 2012a) can be utilized in this examples to aggregate local information across all processors.

3.3 Binning

The demo command is

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(binning,'pbdDEMO',ask=F,echo=F)"
```

Binning is a classical statistics and can quickly summarize the data structure by setting some breaks between max and min of data. This is particularly a useful tool for constructing histograms and categorical data analysis.

The demo `binning` generates fake data on 4 processors, then utilize `mpi.bin` function as

R Code

```
mpi.bin <- function(x.spmd, breaks = pi / 3 * (-3:3)){
  bin.spmd <- table(cut(x.spmd, breaks = breaks))
  bin <- as.array(allreduce(bin.spmd, op = "sum"))
  dimnames(bin) <- dimnames(bin.spmd)
  class(bin) <- class(bin.spmd)
  bin
} # End of mpi.bin().
```

An easy implementation is to utilize `table` function to obtain local counts, then call `allreduce` to obtain global counts.

3.4 Quantile

Example: compute sample quantile order statistics for distributed data.

The demo command is

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(quantile,'pbdDEMO',ask=F,echo=F)"
```

Quantile is the other useful tool from fundamental statistics which provides data distribution for the desired value. This example can be extended to construct Q-Q plot, compute cumulative density function and nonparametric statistics, solve maximum likelihood estimators.

This is only an inefficient implementation to approximate a quantile and is not equivalent to the original `quantile` function in R. But in some sense, it should work well in large scale. The demo `quantile` generates fake data on 4 processors, then utilizes `mpi.quantile` function as

R Code

```r
mpi.quantile <- function(x.spmd, prob = 0.5){
  if(sum(prob < 0 | prob > 1) > 0){
    stop("prob should be in (0, 1)")
  }

  N <- allreduce(length(x.spmd), op = "sum")
  x.max <- allreduce(max(x.spmd), op = "max")
  x.min <- allreduce(min(x.spmd), op = "min")

  f.quantile <- function(x, prob = 0.5){
    allreduce(sum(x.spmd <= x), op = "sum") / N - prob
  }

  uniroot(f.quantile, c(x.min, x.max), prob = prob[1])$root
} # End of mpi.quantile().
```

where a numerical function is solved by `uniroot` to find out the appropriate value such that cumulated probability is less than or equal to the specified quantile.

This simple example shows that the SPMD is greatly applicable on large scale data analysis and likelihood computing. Note that the `uniroot` call is working in parallel and on distributed data, i.e. other optimization functions such as `optim` and `nlm` can be utilized in the same way, since SPMD simply assumes every processors do the same work simulatinuously.

3.5 Ordinary Least Squares

Example: compute ordinary least square solutions for SPMD distributed data.

The demo command is

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(ols,'pbdDEMO',ask=F,echo=F)"
```

Ordinary least squares (OLS) is perhaps *the* fundamental tool of the statistician. The goal is to find a solution $\boldsymbol{\beta}$ such that

$$||\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}||_2^2 \tag{3.4}$$

which is minimized. In statistics, we tend to prefer to think of the problem as being of the form

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

where $\boldsymbol{y}$ is $N \times 1$ observed vector, $\boldsymbol{X}$ is $N \times p$ designed matrix which is full rank and $N >> p$, $\boldsymbol{\beta}$ is the insterested parameters and unknown to be estimated, and $\boldsymbol{\epsilon}$ is errors and to be minimized in norm.

Note that above, we do indeed mean (in fact, stress) *a* solution to the linear least squares problem. The full story is somewhat complicated. The short explanation is that for many applications a statistician will face, expression (3.4) will actually have a unique solution. But this is not always the case. Indeed, it may occur that there is an infinite family of solutions. So typically we go further and demand that a solution $\boldsymbol{\beta}$ be such that $||\boldsymbol{\beta}||_2$ is at least as small as the corresponding norm of any other solution (although even this does not guarantee uniqueness).

A properly thorough treatment of the problems involved here go beyond the scope of this document, and require the reader have in-depth familiarity with linear algebra. For our purposes, the concise explanation above will suffice.

The classical Maximum Likelihood solution is given by:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{3.5}$$

This example can be also generalized to weighted least square (WLS), and linear mixed effect models (LME).

The implementation is straight forward as

R Code

```r
mpi.ols <- function(y.spmd, X.spmd){
  if(length(y.spmd) != nrow(X.spmd)){
    stop("length(y.spmd) != nrow(X.spmd)")
  }

  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
} # End of mpi.ols().
```

While this is a fine demonstration of the power of "getting your hands dirty", this approach is only efficient for small N and small p. Worse, directly computing the product

$$\boldsymbol{X}^T\boldsymbol{X}$$

is often numerically non-stable. Instead, it is generally better (although much slower) to take an orthogonal factorization of the data matrix. Typically, the QR-decomposition is used to this end. Here $\boldsymbol{X} = QR$, where Q is orthogonal and R is upper trapezoidal. This is beneficial, because orthogonal matrices are norm-preserving, and whence

$$\begin{aligned}
||\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}||_2 &= ||QR\boldsymbol{\beta} - \boldsymbol{y}||_2 \\
&= \left|\left|Q^TQR\boldsymbol{\beta} - Q^T\boldsymbol{y}\right|\right|_2 \\
&= \left|\left|R\boldsymbol{\beta} - Q^T\boldsymbol{y}\right|\right|_2
\end{aligned}$$

The (arguably) much more well-known Singular Value Decomposition can also be used to develop yet another algebraically identical solution which is quite elegant. Here, if we take $\boldsymbol{X} = U\Sigma V^T$, then it can be shown that “the” desired solution is given by

$$\boldsymbol{\beta} = V\Sigma^+ U^T \boldsymbol{y}$$

where Σ^+ is the Moore-Penrose pseudoinverse of Σ. However, this approach is handily the most computationally intensive.

The method utilizing QR to find a minimum norm solution has been implemented in **pbdDMAT** for objects of class `ddmatrix`. For larger problems, and especially those where numerical accuracy is important, it may be more convenient to simply convert `y.spmd` and `X.spmd` into block-cyclic format as in the Part IV and to utilize **pbdBASE** and **pbdDMAT** for all matrix computation.

Part III

Reading and Managing Data

The **pbdBASE** and **pbdDMAT** packages offer a distributed matrix class, `ddmatrix`, as well as a collection of high-level methods for performing common matrix operations. For example, if you want to compute the mean of an R matrix `x`, you would call

```
mean(x)
```

That's exactly the same command you would issue if `x` is no longer an ordinary R matrix, but a distributed matrix. These methods range from simple, embarrassingly parallel operations like sums and means, to tightly coupled linear algebra operations like matrix-matrix multiply and singular value decomposition.

Unfortunately, using these higher methods comes with a different cost: getting the data into the distributed matrix class. This can be especially frustrating because we assume that the any object of class `ddmatrix` is *block cyclically distributed*. This concept is discussed at length in the **pbdBASE** vignette (Schmidt *et al.*, 2012d), and we do not intend to discuss the concept of a block cyclic data distribution at length herein. However, we will demonstrate several examples of getting data into and out of the distributed block cyclic matrix format.

In short, once the hurdle of getting the data into the "right format" is out of the way, these methods offer very simple syntax (designed to mimic R as closely as possible) with the ability to scale computations on very large distributed machines.

4

Random Distributed Matrices

For testing, or sometimes even real work, “non-data” matrices — those with entries which are constant or are sampled from a probability distribution — can be very useful. We present several ways in which one can create such matrices, based on differing sets of input parameters.

4.1 Fixed Global Dimension

Example: randomly generate distributed matrices with random normal data of specificed global dimension.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(randmat_global,'pbdDEMO',ask=F,echo=F)"
```

This demo shows 3 separate ways that one can generate a random normal matrix with specified global dimension. The first two generate the matrix in full on at least one processor and distributes the data, while the last method generates locally only what is needed. As such, the first two can be considered demonstrations with what to do when you have data read in on one processor and need to distribute it out to the remaining processors, but for the purposes of using a randomly generated distributed matrix, they are not particularly efficient strategies.

The basic idea is as follows. If I have a matrix `x` stored on processor 0 and `NULL` on the others, then I can distribute it out as an object of class `ddmatrix` via the command `as.ddmatrix()`. For example

```
if (comm.rank()==0){
  x <- matrix(rnorm(100), nrow=10, ncol=10)
} else {
  x <- NULL
}

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

will distribute the required data to the remaining processors. We note for clarity that this is not equivalent to sending the full matrix to all processors and then throwing away all but what is needed. Only the required data is communicated to the processors.

That said, having all of the data on all processors can be convenient while testing, if only for being more minimalistic in the amount of code/thinking required. To do this, one need only do the following:

```
x <- matrix(rnorm(100), nrow=10, ncol=10)

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

Now, this assumes you are using the same seed on each processor. This can be managed using the **pbdMPI** function `comm.set.seed()`, as in the demo script. For more information, see that package's documentation.

Finally, you can generate locally only what you need. The demo script does this via the **pbdDEMO** package's `Hnorm()` or "huge normal" function. There are two others provided, namely `Hconst()` and `Hunif()`. The naming convention was chosen because the latter most function name makes me laugh.

Internally, these "huge" functions rely on a much stronger working knowledge of the underlying data structure than most will be comfortable with. However, for the sake of completeness, we will briefly examine `Hnorm()`.

Hnorm()

```
Hnorm <- function(dim, bldim, mean=0, sd=1, ICTXT=0)
{
  if (length(bldim)==1L)
    bldim <- rep(bldim, 2L)

  ldim <- base.numroc(dim=dim, bldim=bldim, ICTXT=ICTXT,
     fixme=FALSE)

  if (any(ldim < 1L)){
    xmat <- matrix(0)
    ldim <- c(1, 1)
  }
  else
    xmat <- matrix(rnorm(prod(ldim), mean=mean, sd=sd),
       nrow=ldim[1L], ncol=ldim[2L])
  dx <- new("ddmatrix", Data=xmat,
            dim=dim, ldim=ldim, bldim=bldim, CTXT=ICTXT)

  return(dx)
}
```

The concise explanation is that the `base.numroc()` utility determines the size of the local storage. This is all very well documented in the **pbdBASE** documentation, but since no one even pretends to read that stuff, NUMROC is a ScaLAPACK tool, which means "NUMber of Rows Or Columns." The function `base.numroc()` is an implementation in R which calculates the number of rows *and* columns at the same time (so it is a bit of a misnomer, but preserved for historical reasons).

More precisely, it calculates the local storage requirements given a global dimension `dim`, a blocking factor `bldim`, and a BLACS context number `ICTXT`. The extra argument `fixme` determines whether or not the lowest value returned should be 1. If `fixme==FALSE` and any of the returned local dimensions are less than 1, then that processor does not actually own any of the global matrix — it has no local storage. But something must be stored, and so we default this to `matrix(0)`, the 1×1 matrix with single entry 0.

4.2 Fixed Local Dimension

Example: randomly generate distributed matrices with random normal data of specificed local dimension.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(randmat_local,'pbdDEMO',ask=F,echo=F)"
```

This is similar to the above, but with a critical difference. Instead of specifying a fixed *global* dimension and then go determine what the local storage space is, instead we specify a fixed *local* dimension and then go figure out what the global dimension should be. This can be useful for testing for weak scaling of an algorithm, where different numbers of cores are compared but with similar ram usage.

To this end, the demo script utilizes the `Hnorm.local()` function, which has the user specify a local dimension size that all the processors should use, as well as a blocking factor and BLACS context value.

Hnorm.local()

```
Hnorm.local <- function(ldim, bldim, mean=0, sd=1, ICTXT=0)
{
  if (length(bldim)==1L)
    bldim <- rep(bldim, 2L)

  blacs_ <- base.blacs(ICTXT=ICTXT)
  nprows <- blacs_$NPROW
  npcols <- blacs_$NPCOL

  dim <- c(nprows*ldim[1L], npcols*ldim[2L])

  if (any( (dim %% bldim) != 0 )){
    comm.cat("WARNING : at least one margin of 'bldim' does not
       divide the global dimension.\n", quiet=T)

    bldim[1L] <- nbd(dim[1L], bldim[1L])
    bldim[2L] <- nbd(dim[2L], bldim[2L])
    comm.cat(paste("Using bldim of ", bldim[1L], "x", bldim[2L],
       "\n\n", sep=""), quiet=T)
  }

  Data <- matrix(rnorm(prod(ldim), mean=mean, sd=sd),
     nrow=ldim[1L], ncol=ldim[2L])

  dx <- new("ddmatrix", Data=Data,
            dim=dim, ldim=ldim, bldim=bldim, CTXT=ICTXT)

  return(dx)
}
```

Now here things get somewhat tricky, because in order for this matrix to exist at all, each margin of the blocking factor must divide (as an integer) the corresponding margin of the global dimension. To better understand why this is so, the reader is suggested to read the **pbdBASE** vignette. But if you already understand or are merely willing to take it on faith, then you surely grant that this is a problem.

So here, we assume that the local dimension is chosen appropriately, but it is possible that a bad blocking factor is supplied by the user. Rather than halt with a stop error, we attempt to find the next best blocking factor possible. We do this with a simple “next best divisor” function:

nbd()

```
nbd <- function(n, d)
{
  if (n < d)
    stop("'n' may not be smaller than 'd'")

  ret <- .Fortran("NBD",
                  as.integer(n), as.integer(d),
                  PACKAGE="pbdDEMO")$D

  return( ret )
}
```

which is just a shallow wrapper on the Fortran code:

NBD

```
      SUBROUTINE NBD(N, D)
      INTEGER N, D, I, TEST

      DO 10 I = D+1, N-1, 1
        TEST = MOD(N, I)
        IF (TEST.EQ.0) THEN
          D = I
          RETURN
        END IF
   10 CONTINUE

      D = N
      RETURN
      END
```

Even those who don't know Fortran should easily be able to see what is going on here. We are given integers `N` and `D`, and we loop over the integers inbetween these two until we find one which divides `N`.

So going back to the `Hnorm.local()` function, the second `if` block contains the readjusting (as necessary) of the blocking factors. Then the local data matrix is generated and wrapped up in its class before being returned — everything else is just sugar.

5 Reading Data

As we mentioned at the beginning of the discussion on distributed matrix methods, most of the hard work in using these tools is getting the data into the right format. Once this hurdle has been overcome, the syntax will magically begin to look like native R syntax. Some insights into this difficulty can be seen in the previous section, but now we tackle the problem head on: how do you get real data into the distributed matrix format?

5.1 CSV Files

Example: read data from a csv directly into a distributed matrix.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(read_csv,'pbdDEMO',ask=F,echo=F)"
```

It is simple enough to read in a csv file serially and then distribute the data out to the other processors. This process is essentially identical to one of the random generation methods in Section 4.1. For the sake of completeness, we present a simple example here:

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

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

However, this is inefficient, especially if the user has access to a parallel file system. In this case, several processes should be used to read parts of the file, and then distribute that data out to the larger process grid. Although really, the user should not be using csv to store large amounts of data because it always requires a sort of inherent "serialness". Regardless, a demonstration of how this is done is useful. We can do so via the **pbdDEMO** package's function `read.csv.ddmatrix` on an included dataset:

Reading a CSV with Multiple Readers

```
dx <- read.csv.ddmatrix("../extra/data/x.csv",
                        sep=",", nrows=10, ncols=10,
                        header=TRUE, bldim=4,
                        num.rdrs=2, ICTXT=0)

print(dx)
```

The code powering the function itself is quite complicated, going well beyond the scope of this document. To understand it, the reader should see the advanced sections of the **pbdBASE** vignette.

5.2 SQL Databases

Example: Read data from a sql database directly into a distributed matrix.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(read_sql,'pbdDEMO',ask=F,echo=F)"
```

Just as above, we can use a SQL database to read in our data, powered by the **sqldf** package (Grothendieck, 2012). Here it is assumed that the data is stored in the database in a structure that is much the same as a csv is stored on disk. Internally, the query performed is:

```
sqldf(paste("SELECT * FROM ", table, " WHERE rowid  = 1"),
   dbname=dbname)
```

To use a more complicated query for a database with differing structure, it should be possible (no promises) to substitute this line of the `read.sql.ddmatrix()` function for the desired query. However, as before, much of the rest of the tasks performed by this function go beyond the scope of this document. However, they are described in the **pbdBASE** package vignette.

6

Redistribution Methods

One final challenge similar to, but distinct from reading in data is managing data which has already been read into the R processes.

6.1 Load Balance and Unload Balance

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(balance,’pbdDEMO’,ask=F,echo=F)"
```

In this example, suppose we have an unbalanced input `X.spmd`, then we call `balance.info` on it to save some information for later uses. Then, we call `load.balance` to obtain balanced matrix `new.X.spmd`. Then, we call `unload.balance` to obtain original matrix `org.X.spmd` which should be exactly the same as `X.spmd`. We list the key steps in the next.

R Code

```
bal.info <- balance.info(X.spmd)
new.X.spmd <- load.balance(X.spmd, bal.info)
org.X.spmd <- unload.balance(new.X.spmd, bal.info)
```

The details are depicted in the Figure 6.1 where we distributed a matrix in two ways: `X.spmd` is unbalanced and is usually filtered or selected from other matrics, `new.X.spmd` is a balanced version of `X.spmd`.

The useful function `balance.info` will returns information, says `bal.info`, how to load balance the given data matrix `X.spmd`. The return of `balance.info` is a list containing two `data.frame`'s (`send` and `recv` and two `vector`'s (`N.allspmd` and `new.N.allspmd`).

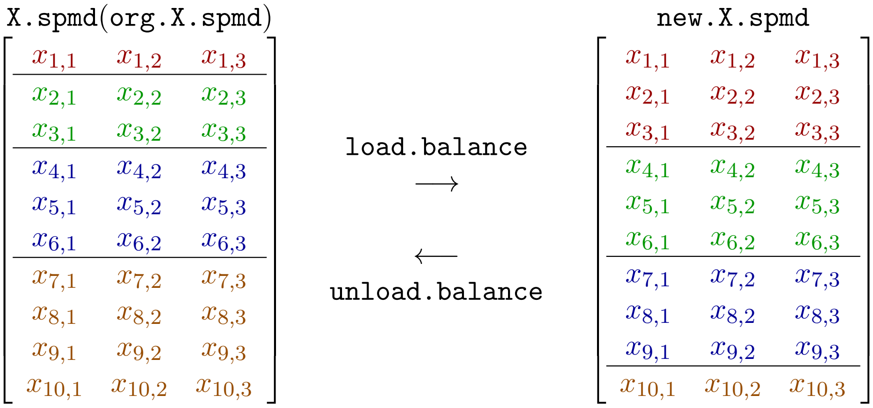

Figure 6.1: $\boldsymbol{X}$ is distributed in `X.spmd(org.X.spmd)` and `new.X.spmd`. Both are distributed by row in 4 processors where colors represents processor 0, 1, 2, and 3.

The `send` records the original rank and the belonging rank of "unbalanced" data matrix. The `load.balance` uses this table to `isend` data. If any belonging rank is not the original rank, then the corresponding entire rows will be sent to where they should belong.

The `recv` also records the original rank and the belonging rank of "balanced" data matrix. The `load.balance` uses this table to `recv` data. If any original rank is not the current rank, then the corresponding entire rows will be received from where they original rank.

The `N.allspmd` and `new.N.allspmd` both have length equals to `comm.rank(comm)` containing all numbers of rows of `X.spmd` before and after balanced. This is for double checking and avoiding 0-row matrix issue.

For `unload.balance`, the process is just a little bit tricky by reversing `bal.info` and passing it to `load.balance`.

6.2 Conver SPMD and DMAT

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(spmd_dmat,'pbdDEMO',ask=F,echo=F)"
```

As usual serial R, it is intuitively to read data in parallel for each processor. We assume the input method is in SPMD to yield data in `X.spmd`, and convert it into `ddmatrix` format in `X.dmat`, then convert again to a R object in `X` which is common on all processors, as in the next. The Figure 6.2 illustates `X.spmd` and `X.dmat`.

R Code

```
X.dmat <- spmd2dmat(X.spmd)
```

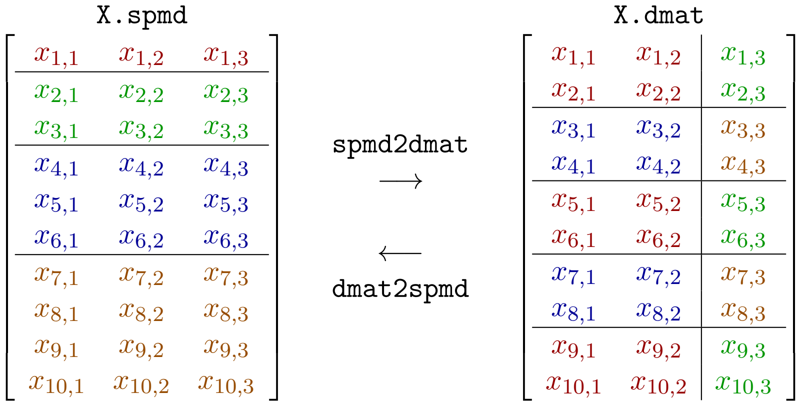

Figure 6.2: $\boldsymbol{X}$ is distributed in X.spmd and X.dmat. Both are distributed in 4 processors where colors represents processor 0, 1, 2, and 3. Note that X.dmat is in block-cyclic format of 2 × 2 grid with 2 × 2 block dimension.

```
X <- as.matrix(X.dmat)
new.X.spmd <- dmat2spmd(X.dmat)
```

Note that we also provide a reversed function dmat2spmd for spmd2dmat.

Here, the spmd2dmat function does the following:

1. Check numbers of columns of X.spmd, all processors should be the same.
2. Row balance the SPMD matrix, since number of rows of X.spmd may vary in different processors.
3. Call new on the balanced matrix to yield a ddmatrix object, says X.dmat, in block context 2 (ICTXT = 2).
4. Convert the X.dmat to other block contexts (defaultICTXT = 0) via base.reblock.

Note that the first step is done by the load.balance function (Section 6.1) which is particular useful to balance SPMD matrix and yield performance even the matrix is not in block-cyclic format.

6.3 SPMD to DMAT

Example: convert SPMD distributed data into a distributed matrix.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(spmd_dmat,'pbdDEMO',ask=F,echo=F)"
```

Suppose you have read data into the processors, by some means, into the SPMD format. Specifically, each processor owns contiguous blocks of rows (a set of rows of the data set), but that the data is not block-cyclically distributed. Then using the **pbdDEMO** package's function `spmd2dmat()`, this conversion is trivial:

```
X.dmat <- spmd2dmat(X.spmd)
```

Unfortunately, once again the ideas driving this routine are somewhat beyond the scope of this document. They are discussed at length in the **pbdBASE** vignette.

Part IV

Distributed Matrix Methods

7 Advanced Statistics Examples

The **pbdDMAT** package contains many useful methods for doing computations with distributed matrices. For comprehensive (but shallow) demonstrations of the distributed matrix methods available, see the **pbdDMAT** package's vignette and demos.

Here we showcase a few more advanced things that can be done by chaining together R and pbdR code seamlessly.

7.1 Verification of Distributed System Solving

Example: solve a system of equations and verify that the solution is correct.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(verify,'pbdDEMO',ask=F,echo=F)"
```

The **pbdDEMO** contains a set of verification routines, designed to test for validity of the numerical methods at any scale. Herein we will discuss the verification method for solving systems of linear equations, `verify.solve()`.

The process is simple. The goal is to solve the equation (in matrix notation)

$$Ax = b$$

for $n \times n$ matrix A and $n \times 1$ matrix b. However, here we start with A and x and use these to produce b. We then forget we ever knew what x was and solve the system. Finally, we remember what x really should be and compare that with our numerical solution.

More specifically, we take the matrix A to be random normal generated data and the true solution x to be a constant vector. We then calculate

$$b := Ax$$

and finally the system is solve for a now (pretend) unknown x, so that we can compare the numerically determined x to the true constant x. All processes are timed, and both success/failure and timing results are printed for the user at the completion of the routine.

Verifying Distributed System Solving

```
verify.solve <- function(nrows=1e3, mean=0, sd=1, const=1,
   bldim=8, tol=1e-7)
{
  # generating data
  time_data <- timer({
    x <- Hnorm(dim=c(nrows, nrows), bldim=bldim, mean=mean,
       sd=sd, ICTXT=0)
    truesol <- Hconst(dim=c(nrows, 1), bldim=bldim, const=const,
       ICTXT=0)
  })

  time_rhs <- timer({
    rhs <- x %*% truesol
  })

  # solving
  time_sol <- timer({
    sol <- solve(x, rhs)
  })

  # verifying
  time_verif <- timer({
    iseq <- all.equal(sol, truesol, tol=tol)
    iseq <- as.logical(allreduce(iseq, op=’min’))
  })

  comm.cat("\nIs the factorization correct?  ", quiet=T)
  if (iseq)
    comm.cat("YES!\n", quiet=T)
  else {
    comm.cat("No...\n", quiet=T)
    s <- x-newx
    diffs <- c(min(s), mean(s), max(s))
    names(diffs) <- c("min", "mean", "max")
    comm.cat("\nPrinting min/mean/max differences between
       original and result from factoring and then
       multiplying...\n", quiet=T)
    comm.print(diffs, quiet=T)
  }

  comm.cat("\n\nRun times:\n", quiet=T)

  comm.print(
         rbind(
            DataGeneration=time_data,
            RHSgeneration=time_rhs,
            Solving=time_sol,
            Verification=time_verif),
          quiet=T)

  tot <- time_data+time_rhs+time_sol+time_verif
  names(tot) <- c("", "", "")

  comm.print(rbind("Total          "=tot), quiet=T)
}
```

7.2 Principal Components Analysis

Example: take PCA and retain only a subset of the rotated data.

The demo command is

Shell Command

```
### At the shell prompt, run the demo with 4 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 4 Rscript -e "demo(pca,'pbdDEMO',ask=F,echo=F)"
```

Suppose we wish to perform a principal components analysis and retain only some subset of the columns of the rotated data. One of the ways this is often done is by using the singular values — the standard deviations of the components — as a measure of variation retained by a component. However, the first step is to get the principal components data. Luckily this is trivial. If our data is stored in the distributed matrix object `dx`, then all we need to is issue the command:

```
pca <- prcomp(x=dx, retx=TRUE, scale=TRUE)
```

Now that we have our PCA object (which has the same structure as that which comes from calling `prcomp()` on an ordinary R matrix), we need only decide how best to throw away what we do not want. We might want to retain at least as many columns as would be needed to retain 90% of the variation of the original data:

```r
prop_var <- cumsum(pca$sdev)/sum(pca$sdev)
i <- min(which(prop_var > 0.9))

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

Or we might want one fewer column than the number that would give us 90%:

```r
prop_var <- cumsum(pca$sdev)/sum(pca$sdev)
i <- max(min(which(prop_var > 0.9)) - 1, 1)

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

Bibliography

Chen WC, Ostrouchov G (2011). “HPSC – High Performance Statistical Computing for Data Intensive Research.” URL http://thirteen-01.stat.iastate.edu/snoweye/hpsc/.

Chen WC, Ostrouchov G, Schmidt D, Patel P, Yu H (2012a). “pbdMPI: Programming with Big Data – Interface to MPI.” R Package, URL `http://cran.r-project.org/package=pbdMPI`.

Chen WC, Ostrouchov G, Schmidt D, Patel P, Yu H (2012b). “A Quick Guide for the pbdMPI package.” R Vignette, URL `http://cran.r-project.org/package=pbdMPI`.

Grothendieck G (2012). *sqldf: Perform SQL Selects on R Data Frames.* R package version 0.4-6.4, URL `http://CRAN.R-project.org/package=sqldf`.

Ostrouchov G, Chen WC, Schmidt D, Patel P (2012). “Programming with Big Data in R.” URL `http://r-pbd.org/`.

R Core Team (2012). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL `http://www.r-project.org/`.

Schmidt D, Chen WC, Ostrouchov G, Patel P (2012a). “pbdBASE: Programming with Big Data – Core pbd Classes and Methods.” R Package, URL `http://cran.r-project.org/package=pbdBASE`.

Schmidt D, Chen WC, Ostrouchov G, Patel P (2012b). “pbdDEMO: Programming with Big Data – Demonstrations of pbd Packages.” R Package, URL `http://cran.r-project.org/package=pbdDEMO`.

Schmidt D, Chen WC, Ostrouchov G, Patel P (2012c). “pbdDMAT: Programming with Big Data – Distributed Matrix Algebra Computation.” R Package, URL `http://cran.r-project.org/package=pbdDMAT`.

Schmidt D, Chen WC, Ostrouchov G, Patel P (2012d). “A Quick Guide for the pbdBASE package.” R Vignette, URL `http://cran.r-project.org/package=pbdBASE`.