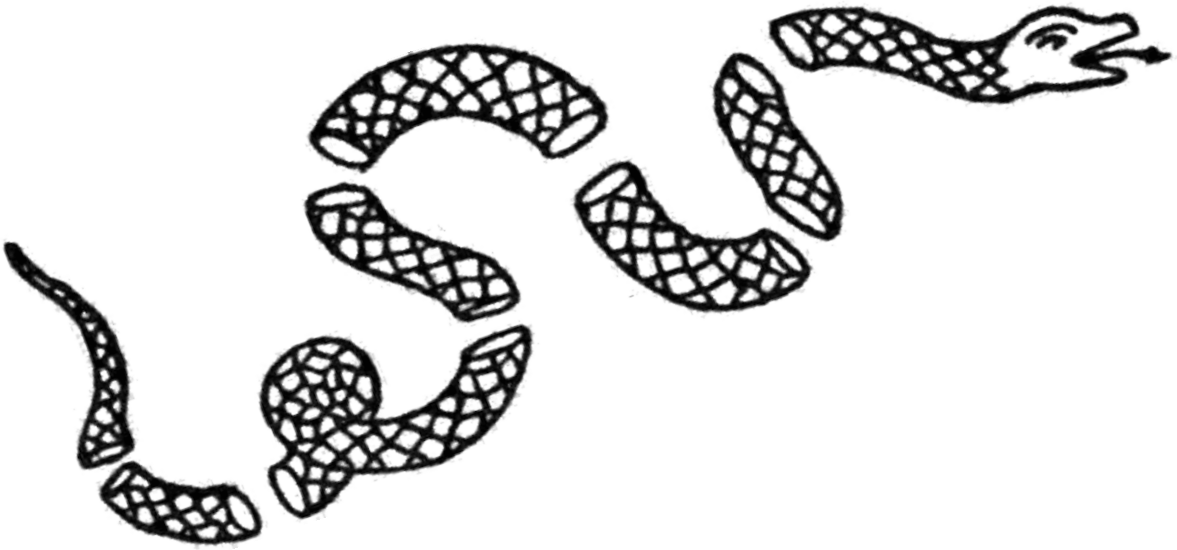
Version 1.1



Programming with Big Data in R

Speaking Serial R with a Parallel Accent

Package Examples and Demonstrations

pbdR Core Team

Speaking Serial R with a Parallel Accent

pbdR Package Examples and Demonstrations

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

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

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

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

Version 1.1

This publication was typeset using LaTeX. The illustrations were created using the **ggplot2** package (Wickham, 2009), except for Figure 1.1, which was created in Microsoft Powerpoint.

Contents

List of Figures

List of Tables

Acknowledgements

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, Uwe Ligges, and Simon Urbanek 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.*, 2013), version 0.1-1. 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

1.1 What is pbdR?

The "Programming with Big Data in R" project (Ostrouchov *et al.*, 2012) (pbd or pbdR for short) is a project that aims to elevate the statistical programming language R (R Core Team, 2012b) to leadership-class computing platforms. The main goal is empower data scientists by bringing flexibility and a big analytics toolbox to big data challenges, with an emphasis on productivity, portability, and performance. We achieve this in part by mapping high-level programming syntax to portable, high-performance, scalable, parallel libraries. In short, we make R scalable.

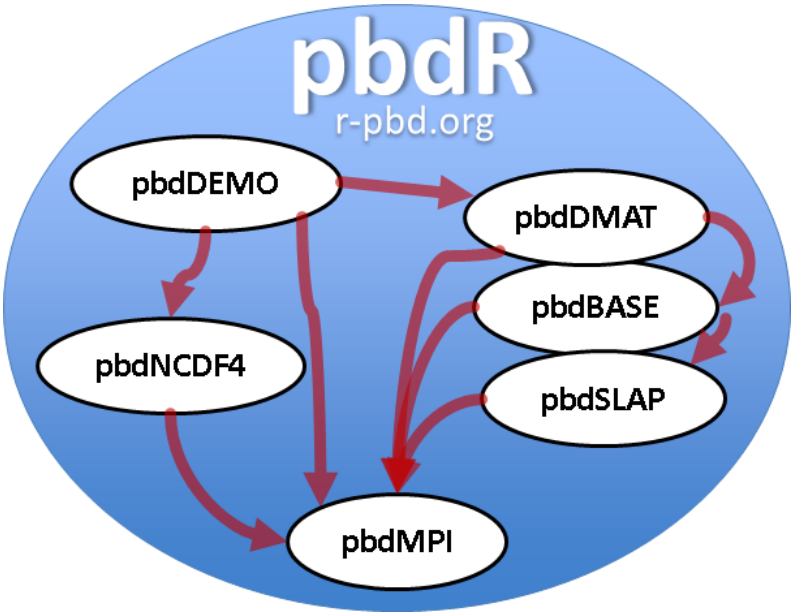


Figure 1.1: pbdR Packages

Figure 1.1 shows the current list of pbdR packages released to the CRAN (http://cran.r-project.org), and how they depend on each other. More explicitly, the current pbdR pack-

ages (Chen *et al.*, 2012b,d; Schmidt *et al.*, 2012a,b; Patel *et al.*, 2013a; Schmidt *et al.*, 2013) are:

- **pbdMPI** — an efficient interface to MPI (Gropp *et al.*, 1994) with a focus on Single Program/Multiple Data (SPMD) parallel programming style.
- **pbdSLAP** — bundles scalable dense linear algebra libraries in double precision for R, based on ScaLAPACK version 2.0.2 (Blackford *et al.*, 1997).
- **pbdNCDF4** — Interface to Parallel Unidata NetCDF4 format data files (NetCDF Group, 2008).
- **pbdBASE** — low-level ScaLAPACK codes and wrappers.
- **pbdDMAT** — distributed matrix classes and computational methods, with a focus on linear algebra and statistics.
- **pbdDEMO** — set of package demonstrations and examples, and this unifying vignette.

To try to make this landscape a bit more clear, one could divide pbdR packages into those meant for users, developers, or something in-between. Figure 1.2 shows a gradient scale representation,

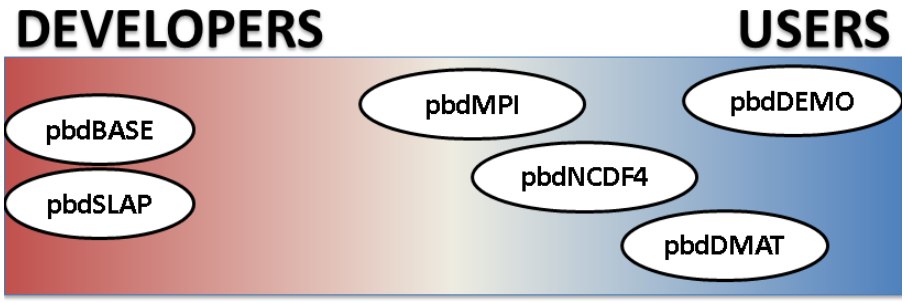


Figure 1.2: pbdR Package Use

where more red means the package is more for developers, while more blue means the package is more for users. For example, **pbdDEMO** is squarely meant for users of pbdR packages, while **pbdBASE** and **pbdSLAP** are really not meant for general use. The other packages fall somewhere in-between, having plenty of utility for both camps.

Finally, Figure 1.3 shows pbdR relationship to high-performance libraries.

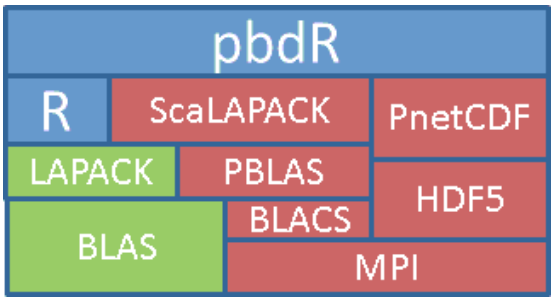


Figure 1.3: pbdR Interface to Foreign Libraries

In this vignette, we offer many examples using the above pbdR packages. Many of the examples are high-level applications and may be commonly found in basic Statistics. The purpose is to show how to reuse the preexisting functions and utilities of pbdR to create minor extensions which can quickly solve problems in an efficient way. The reader is encouraged to reuse and re-purpose these functions.

The **pbdDEMO** package consists of two main parts. The first is a collection of roughly 20+ package demos. These offer example uses of the various pbdR packages. The second is this vignette, which attempts to offer detailed explanations for the demos, as well as sometimes providing some mathematical or statistical insight. A list of all of the package demos can be found in Section 1.4.1.

1.2 Why Parallelism? Why pbdR?

It is common, in a document such as this, to justify the need for parallelism. Generally this process goes:

Blah blah blah Moore's Law, blah blah Big Data, blah blah blah Concurrency.

How about this? Parallelism is cool. Any boring nerd can use one computer, but using 10,000 at once is another story. We don't call them ***supercomputers*** for nothing.

But unfortunately, lots of people who would otherwise be thrilled to do all kinds of cool stuff with massive behemoths of computation — computers with names like **KRAKEN**[1] and **TITAN**[2] — are burdened by an unfortunate reality: it's really, really hard. Enter pbdR. Through our project, we put a shiny new set of clothes on high-powered compiled code, making massive-scale computation accessible to a wider audience of data scientists than ever before. Analytics in supercomputing shouldn't just be for the elites.

1.3 Installation

One can download **pbdDEMO** from CRAN at http://cran.r-project.org, and the installation 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 installation of any of them is unsuccessful.

[1] http://www.nics.tennessee.edu/computing-resources/kraken
[2] http://www.olcf.ornl.gov/titan/

1.4 Structure of pbdDEMO

The **pbdDEMO** package consists of several key components:

1. This vignette
2. A set of demos in the `demo/` tree
3. A set of benchmark codes in the `Benchmarks/` tree

The following subsections elaborate on the contents of the latter two.

1.4.1 List of Demos

A full list of demos contained in the **pbdDEMO** package is provided below. We may or may not describe all of the demos in this vignette.

List of Demos

```
### (Use Rscript.exe for windows systems)

                        # ---------------------- #
                        # II Direct MPI Methods #
                        # ---------------------- #

### Chapter 4
# 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)"
# Distributed Logic
mpiexec -np 4 Rscript -e "demo(comparators, package='pbdDMAT', ask=F,
    echo=F)"

                    # ------------------------------ #
                    # III Distributed Matrix Methods #
                    # ------------------------------ #

### Chapter 5
# 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)"

### Chapter 7
# Sample statistics revisited
mpiexec -np 4 Rscript -e "demo(sample_stat_dmat, package='pbdDMAT',
    ask=F, echo=F)"
# Verify solving Ax=b at scale
mpiexec -np 4 Rscript -e "demo(verify, package='pbdDMAT', ask=F,
    echo=F)"
# PCA compression
mpiexec -np 4 Rscript -e "demo(pca, package='pbdDMAT', ask=F, echo=F)"
# OLS and predictions
mpiexec -np 4 Rscript -e "demo(ols_dmat, package='pbdDMAT', ask=F,
    echo=F)"

                  # ---------------------------- #
                  # IV Reading and Managing Data #
                  # ---------------------------- #

### Chapter 8
# Reading csv
mpiexec -np 4 Rscript -e "demo(read_csv, package='pbdDMAT', ask=F,
    echo=F)"
# Reading sql
mpiexec -np 4 Rscript -e "demo(read_sql, package='pbdDMAT', ask=F,
    echo=F)"

### Chapter 9
# Reading and writing parallel NetCDF4
Rscript -e "demo(trefht, package="pbdDEMO", ask = F, echo = F)"
mpiexec -np 4 Rscript -e "demo(nc4_serial, package='pbdDEMO', ask=F,
    echo=F)"
mpiexec -np 4 Rscript -e "demo(nc4_parallel, package='pbdDEMO', ask=F,
    echo=F)"
mpiexec -np 4 Rscript -e "demo(nc4_dmat, package='pbdDEMO', ask=F,
    echo=F)"
mpiexec -np 4 Rscript -e "demo(nc4_spmdc, package='pbdDEMO', ask=F,
    echo=F)"

### Chapter 10
# Loand/unload balance
mpiexec -np 4 Rscript -e "demo(balance, package='pbdDMAT', ask=F,
    echo=F)"
# SPMD to DMAT
mpiexec -np 4 Rscript -e "demo(spmd_dmat, package='pbdDMAT', ask=F,
    echo=F)"
# Distributed matrix redistributions
mpiexec -np 4 Rscript -e "demo(reblock, package='pbdDMAT', ask=F,
    echo=F)"

                  # ---------------------------- #
                  # V Applications               #
```

```
                # ---------------------------- #

### Chapter 11
# Parallel Model-Based Clustering
Rscript -e "demo(iris_overlap, package='pbdDEMO', ask=F, echo=F)"
Rscript -e "demo(iris_serial, package='pbdDEMO', ask=F, echo=F)"
Rscript -e "demo(iris_spmd, package='pbdDEMO', ask=F, echo=F)"
Rscript -e "demo(iris_dmat, package='pbdDEMO', ask=F, echo=F)"
```

1.4.2 List of Benchmarks

At the time of writing, there are benchmarks for computing covariance, linear models, and principal components. The benchmarks come in two variants. The first is an ordinary set of benchmark codes, which generate data of specified dimension(s) and time the indicated computation. This operation is replicated for a user-specified number of times (default 10), and then the timing results are printed to the terminal.

From the `Benchmarks/` subtree, the user can run the first set of benchmarks with, for example, 4 processors by issuing any of the commands:

```
### (Use Rscript.exe for windows systems)
mpiexec -np 4 Rscript cov.r
mpiexec -np 4 Rscript lmfit.r
mpiexec -np 4 Rscript pca.r
```

The second set of benchmarks are those that try to find the "balancing" point where, for the indicted computation with user specified number of cores, the computation is performed faster using pbdR than using serial R. In general, throwing a bunch of cores at a problem may not be the best course of action, because parallel algorithms (almost always) have inherent overhead over their serial counterparts that can make their use ill-advised for small problems. But for sufficiently big (which is usually not very big at all) problems, that overhead should quickly be dwarfed by the increased scalability.

From the `Benchmarks/` subtree, the user can run the second set of benchmarks with, for example, 4 processors by issuing any of the commands:

```
### (Use Rscript.exe for windows systems)
mpiexec -np 4 Rscript balance_cov.r
mpiexec -np 4 Rscript balance_lmfit.r
mpiexec -np 4 Rscript balance_pca.r
```

Now we must note that there are other costs than just statistical computation. There is of course the cost of disk IO (when dealing with real data). However, a parallel file system should help with this, and for large datasets should actually be faster anyway. The main cost not measured here is the cost of starting all of the R processes and loading packages. Assuming R is not compiled statically (and it almost certainly is not), then this cost is non-trivial and somewhat unique to very large scale computing. For instance, it took us well over an hour to start 12,000

R sessions and load the required packages on the supercomputer KRAKEN[3]. This problem is not unique to R, however. It affects any project that has a great deal of dynamic library loading to do. This includes Python, although their community has made some impressive strides in dealing with this problem.

1.5 Exercises

1-1 Read the MPI wikipedia page https://en.wikipedia.org/wiki/Message_Passing_Interface including it's history, overview, functionality, and concepts sections.

1-2 Read the **pbdMPI** vignette and install either OpenMPI (http://www.open-mpi.org/) or MPICH2 (http://www.mcs.anl.gov/research/projects/mpich2/), and test if the installation is correct (see http://www.r-pbd.org/install.html for more details).

1-3 After completing Exercise 1-2, install all pbdR packages and run each package's demo codes.

[3]See https://en.wikipedia.org/wiki/Kraken_(supercomputer)

2 Background

2.1 Parallelism

What is parallelism? At its core (pun intended), parallelism is all about trying to throw more resources at a problem, usually to get the problem to complete faster than it would with the more minimal resources. Sometimes we wish to utilize more resources as a means of being able to make a computationally (literally or practically) intractable problem into one which will complete in a reasonable amount of time. Somewhat more precisely, parallelism is the leveraging of parallel processing. It is a general programming model whereby we execute different computations simultaneously. This stands in contrast to *serial* programming, where you have a stream of commands, executed one at a time.

Serial programming has been the dominant model from the invention of the computer to present, although this is quickly changing. The reasons why this is changing are numerous and boring; the fact is, if it is true now that a researcher must know some level of programming to do his/her job, then it is certainly true that in the near future that he/she will have to be able to do some parallel programming. Anyone who would deny this is, frankly, more likely trying to vocally assuage personal fears more so than accurately forecasting based on empirical evidence. For many, parallel programming isn't *coming*; it's *here*.

As a general rule, parallelism should only come after you have exhausted serial optimization. Even the most complicated parallel programs are made up of serial pieces, so inefficient serial codes produce inefficient parallel codes. Also, generally speaking, one can often eke out much better performance by implementing a very efficient serial algorithm rather than using a handful of cores (like on a modern multicore laptop) using an inefficient parallel algorithm. However, once that serial-optimization well runs dry, if you want to continue seeing performance gains, then you must implement your code in parallel.

Next, we will discuss some of the major parallel programming models. This discussion will be fairly abstract and superficial; however, the overwhelming bulk of this text is comprised of examples which will appeal to data scientists, so for more substantive examples, especially for those more familiar with parallel programming, you may wish to jump to Section 4.

Data Parallelism

There are many ways to write parallel programs. Often these will depend on the physical hardware you have available to you (multicore laptop, several GPU's, a distributed supercomputer, ...). The pbdR project is principally concerned with *data parallelism*. We will expand on the specifics in Section 2.3 and provide numerous examples throughout this guide. However, in general, data parallelism is a parallel programming model whereby the programmer splits up a data set and applies operations on the sub-pieces to solve one larger problem.

Figure 2.1 offers a visualization of a very simple data parallelism problem. Say we have an array

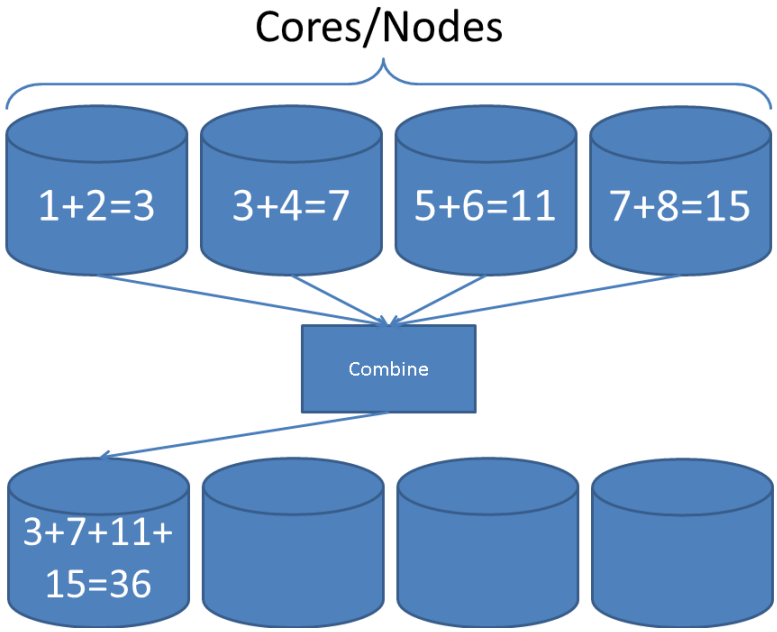


Figure 2.1: Task Parallelism Example

consisting of the values 1 through 8, and we have 4 cores (processing units) at our disposal, and we want to add up all of the elements of this array. We might distribute the data as in the diagram (the first two elements of the array on the first core, the next two elements on the second core, and so on). We then perform a local summation operation; this local operation is serial, but because we have divided up the overall task of summation across the multiple processors, for a very large array we would expect to see performance gains.

A very loose pseudo code for this procedure might look like:

Pseudocode

```
1: mydata = map(data)
2: total_local = sum(mydata)
3: total = reduce(total_local)
4: if this_processor == processor_1 then
5:     print(total)
6: end if
```

Then each of the four cores could execute this code simultaneously, with some cooperation between the processors for step 1 (in determining who owns what) and for the reduction in step 3. This is an example of using a higher-level parallel programming paradigm called "Single Program/Multiple Data" or SPMD. . We will elucidate more as to exactly what this means in the sections to follow.

Task Parallelism

Data parallelism is one parallel programming model. By contrast, another important parallel programming model is *task parallelism*, which much of the R community is already fairly adept at, especially when using the manager/worker paradigm (more on this later). Common packages for this kind of work include **snow** (Tierney *et al.*, 2012), **parallel** (R Core Team, 2012a), and **Rmpi** (Yu, 2012)[1].

Task parallelism involves, as the name implies, distributing different execution tasks across processors. Task parallelism is often *embarrassingly parallel* — meaning the parallelism is so easy to exploit that it is embarrassing. This kind of situation occurs when you have complete independence, or a *loosely coupled* problem (as opposed to something *tightly coupled*, like computing the Singular Value Decomposition (SVD) of a distributed data matrix, for example).

As a simple example of task parallelism, say you have one dataset and four processing cores, and you want to fit all four different linear regression models for that dataset, and then choose the model with lowest AIC (Akaike, 1974) (we are not arguing that this is necessarily a good idea; this is just an example). Fitting one model does not have any dependence on fitting another, so you might want to just do the obvious thing and have each core fit a separate model, compute the AIC value locally, then compare all computed AIC values, lowest is the winner. Figure 2.2 offers a simple visualization of this procedure.

A very loose pseudo code for this problem might look like:

Pseudocode

```
1: load_data()
2: if this_processor == processor_1 then
3:     distribute_tasks()
4: else
5:     receive_tasks()
6: end if
7: model_aic = aic(fit(mymodel))
8: best_aic = min(allgather(model_aic))
9: if model_aic == best_aic then
10:    print(mymodel)
11: end if
```

Then each of the four cores could execute this code simultaneously, with some cooperation between the processors for the distribution of tasks (which model to fit) in step 1 and for the

[1]For more examples, see "CRAN Task View: High-Performance and Parallel Computing with R" at http://cran.r-project.org/web/views/HighPerformanceComputing.html.

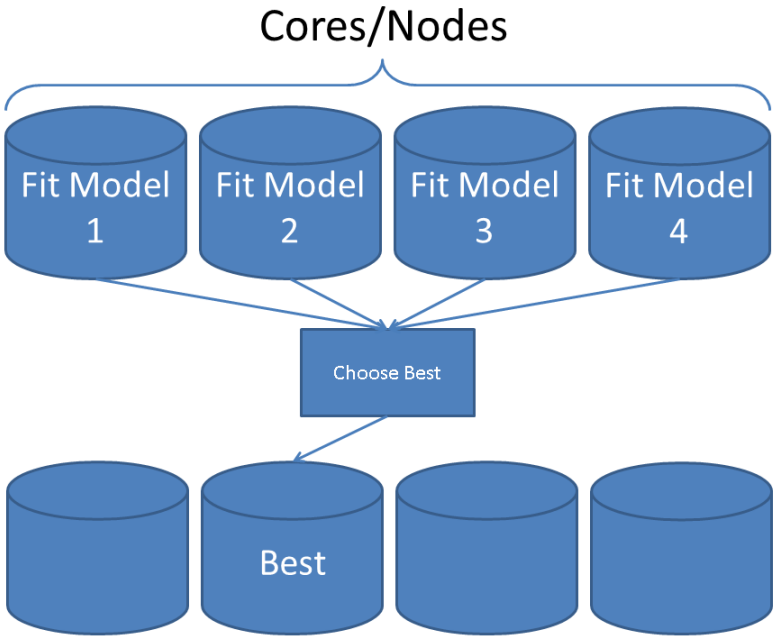


Figure 2.2: Task Parallelism Example

gather operation in step 8.

The line between data parallelism and task parallelism sometimes blurs, especially in simple examples such as those presented here; given our loose definitions of terms, is our first example really data parallelism? Or is it task parallelism? It is best not to spend too much time worrying about what to call it and instead focus on how to do it. These are simple examples and should not be taken too far out of context. All that said, the proverbial rabbit hole of parallel programming goes quite deep, and often it is not a matter of one programming model or another, but leveraging several at once to solve a complicated problem.

2.2 SPMD Programming with R

Throughout this document, we will be using the 'Single Program/Multiple Data", or SPMD, paradigm for distributed computing. Writing programs in the SPMD style is a very natural way of doing computations in parallel, but can be somewhat difficult to properly describe. As the name implies, only one program is written, but the different processors involved in the computation all execute the code independently on different portions of the data. The process is arguably the most natural extension of running serial codes in batch. This model lends itself especially well to data parallelism problems.

Unfortunately, executing jobs in batch is a somewhat unknown way of doing business to the typical R user. While details and examples about this process will be provided in the chapters to follow, the reader is encouraged to read the **pbdMPI** package's vignette (Chen *et al.*, 2012c) first. Ideally, readers should run the demos of the **pbdMPI** package, going through the code step by step.

This paradigm is just one model of parallel programming, and in reality, is a sort of "meta model", encompassing many other parallel programming models. The R community is already familiar with the manager/worker[2] programming model. This programming model is particularly well-suited to task parallelism, where generally one processor will distribute the task load to the other processors.

The two are not mutually exclusive, however. It is easy enough to engage in task parallelism from an SPMD-written program. To do this, you essentially create a "task-parallelism block", where one processor

Pseudocode

```
1: if this_processor == manager then
2:     distribute_tasks()
3: else
4:     receive_tasks()
5: end if
```

See Exercise 2-2 for more details.

One other model of note is the MapReduce model. A well-known implementation of this is Apache's Hadoop, which is really more of a poor man's distributed file system with MapReduce bolted on top. The R community has a strange affection for MapReduce, even among people who have never used it. MapReduce, and for instance Hadoop, most certainly has its place, but one should be aware that MapReduce is not very well-suited for tightly coupled problems; this difficulty goes beyond the fact that tightly coupled problems are harder to parallelize than their embarrassingly parallel counterparts, and is, in part, inherent to MapReduce itself. For the remainder of this document, we will not discuss MapReduce further.

2.3 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 than 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.*, 2012b) packages.

[2] Sometimes referred to as "master/slave" or "master/worker"

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 analysis 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` to differentiate them from the more general `.spmd` object. 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).

2.4 Exercises

2-1 Read the SPMD wiki page at http://en.wikipedia.org/wiki/SPMD and it's related information.

2-2 **pbdMPI** provides a function `get.jid()` to divide `N` jobs into n processors nearly equally which is best for homogeneous computing environment to do task parallelism. The FAQs section of **pbdMPI**'s vignette has an example, try it as next.

R Script

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

id <- get.jid(N)

### Using a loop
for(i in id){
  # put independent task i script here
}
finalize()
```

2-3 Multi-threading and forking are also popular methods of parallelism for shared memory systems, such as in a personal laptop. The function `mclapply()`[3] in **parallel** originated from the **multicore** (Urbanek, 2011) package, and is for simple parallelism on shared memory machines by using the `fork` mechanism. Compare this with OpenMP (OpenMP ARB, 1997).

[3] This method is not available on Windows, because Windows has no system-level fork command.

Part II

Direct MPI Methods

3

MPI for the R User

Cicero once said that "If you have a garden and a library, you have everything you need." So in that spirit, for the next two chapters we will use the MPI library to get our hands dirty and root around in the dirt of low-level MPI programming.

3.1 MPI Basics

In a sense, Cicero (in the above tortured metaphor) was quite right. MPI is all that we *need* in the same way that I might only *need* bread and cheese, but really what I *want* is a pizza. MPI is somewhat low-level and can be quite fiddly, but mastering it adds a very powerful tool to the repertoire of the parallel R programmer, and is essential for anyone who wants to do large scale development of parallel codes.

"MPI" stands for "Message Passing Interface". How it really works goes *well* beyond the scope of this document. But at a basic level, the idea is that the user is running a code on different compute nodes that (usually) can not directly modify objects in each others' memory. In order to have all of the nodes working together on a common problem, data and computation directives are passed around over the network (often over a specialized link called infiniband).

At its core, MPI is a standard interface for managing communications (data and instructions) between different nodes or computers. There are several major implementations of this standard, and the one you should use may depend on the machine you are using. But this is a compiling issue, so user programs are unaffected beyond this minor hurdle. Some of the most well-known implementations are OpenMPI, MPICH2, and Cray MPT.

At the core of using MPI is the *communicator*. At a technical level, a communicator is a pretty complicated data structure, but these deep technical details are not necessary for proceeding. We will instead think of it somewhat like the post office. When we wish to send a letter (communication) to someone else (another processor), we merely drop the letter off at a post office mailbox (communicator) and trust that the post office (MPI) will handle things accordingly (sort of).

The general process for directly — or indirectly — utilizing MPI in SPMD programs goes something like this:

1. Initialize communicator(s).
2. Have each process read in its portion of the data.
3. Perform computations.
4. Communicate results.
5. Shut down the communicator(s).

Some of the above steps may be swept away under a layer of abstraction for the user, but the need may arise where directly interfacing with MPI is not only beneficial, but necessary.

More details and numerous examples using MPI with R are available in the sections to follow, as well as in the **pbdMPI** vignette.

3.2 pbdMPI vs Rmpi

There is another package on the CRAN which the R programmer may use to interface with MPI, namely **Rmpi** (Yu, 2012). There are several issues one must consider when choosing which package to use if one were to only use one of them.

1. (+) **pbdMPI** is easier to install than **Rmpi**
2. (+) **pbdMPI** is easier to use than **Rmpi**
3. (+) **pbdMPI** can often outperform **Rmpi**
4. (+) **pbdMPI** integrates with the rest of pbdR
5. (−) **Rmpi** can be used with **foreach** (Analytics, 2012) via **doMPI** (Weston, 2010)
6. (−) **Rmpi** can be used in the manager/worker paradigm

We do not believe that the above can be reduced to a zero-sum game with unambiguous winner and loser. Ultimately the needs of the user (or developer) are paramount. We believe that pbdR makes a very good case for itself, especially in the SPMD style, but it can not satisfy everyone. However, for the remainder of this section, we will present the case for several of the, as yet, unsubstantiated pluses above.

In the case of ease of use, **Rmpi** uses bindings very close to the level as they are used in C's MPI API. Specifically, whenever performing, for example, a reduction operation such as "allreduce", you must specify the type of your data. For example, using **Rmpi**'s API

```
1 mpi.allreduce(x, type = 1)
```

would perform the sum allreduce if the object `x` consists of integer data, while

```
mpi.allreduce(x, type = 2)
```

would be used if x consists of doubles. However, with **pbdMPI**

```
allreduce(x)
```

is used for both by making use of R's S4 system of object oriented programming. This is not mere code golfing[1] that we are engaging in. The concept of what "type" your data is in R is fairly foreign to most R users, and misusing the `type` argument in **Rmpi** is a very easy way to crash your program. Even if you are more comfortable with statically typed languages and have no problem with this concept, consider the following example:

Types in R

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

There are good reasons for R Core to have made this choice; that is not the point. The point is that because objects in R are dynamically typed, having to know the type of your data when utilizing **Rmpi** is a needless burden. Instead, **pbdMPI** takes the approach of adding a small abstraction layer on top (which we intend to show does not negatively impact performance) so that the user need not worry about such fiddly details.

In terms of performance, **pbdMPI** can greatly outperform **Rmpi**. We present here the results of a benchmark we performed comparing the "allgather" operation between the two packages (Schmidt *et al.*, 2012e). The benchmark consisted of calling the respective "allgather" function from each package on a randomly generated $10,000 \times 10,000$ distributed matrix with entries coming from the standard normal distribution, using different numbers of processors. Table 3.1 shows the results for this test, and in each case, **pbdMPI** is the clear victor.

Table 3.1: Benchmark Comparing **Rmpi** and **pbdMPI**. Run time in seconds is listed for each operation. The speedup is relative to **Rmpi**.

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

[1] See https://en.wikipedia.org/wiki/Code_golf

Whichever package you choose, whichever your favorite, for the remainder of this document we will be using (either implicitly or explicitly) **pbdMPI**.

3.3 The SPMD Data Structure

This is the boring stuff everyone hates, but like your medicine, it's ultimately better for you to just take it and get it out of the way: data structures. In particular, we will be discussing a distributed data structure that, for lack of a better name (and I assure you are tried), we will call the SPMD data structure. This data structure is more paradigm or philosophy than a rigid data structure like an array or list. Consider it a set of "best practices", or if nothing else, a starting place if you have no idea how to proceed.

The SPMD data structure is designed to fit the types of problems which are arguably most common to data science, namely tall and skinny matrices. It will work best with these (from a computational efficiency perspective) problems, although that is not required. In fact, very little at all is required of this data structure. At its core, the data structure is a distributed matrix data structure, with the following rules:

1. `SPMD` is *distributed*. No one processor owns all of the matrix.
2. `SPMD` is *non-overlapping*. Any row owned by one processor is owned by no other processors.
3. `SPMD` is *row-contiguous*. If a processor owns one element of a row, it owns the entire row.
4. `SPMD` is globally *row-major*[2], locally *column-major*[3].
5. 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. That is, global row-adjacency is preserved in local storage via the communicator.
6. `SPMD` is (relatively) easy to understand, but can lead to bottlenecks if you have many more columns than rows.

Of this list, perhaps the most difficult to understand is number 5. This is a precise, albeit cumbersome explanation for a simple idea. If two processors are adjacent and each owns data, then their local sub-matrices are adjacent row-wise as well. For example, rows n and $n+1$ of a matrix are adjacent; possible configurations for the distributed ownership are processors q owns row n and $q+1$ owns row $n+1$; processor q owns row n, processor $q+1$ owns *nothing*, and processor $q+2$ owns row $n+1$.

For some, no matter how much we try or what we write, the wall of text simply will not suffice. So here are a few visual examples. Suppose we have the global data matrix x, given as:

[2] In the sense of the data decomposition. More specifically, the global matrix is chopped up into local sub-matrices in a row-major way.

[3] The local sub-objects are R matrices, which are stored in column-major fashion.

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

with processor array[4] (indexing always starts at 0 not 1)

$$\text{Processors} = \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array}$$

Then we might split up and distribute the data onto processors like so:

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

With local storage view:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & x_{17} & x_{18} & x_{19} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} & x_{27} & x_{28} & x_{29} \end{bmatrix}_{2\times 9}$$
$$\begin{bmatrix} x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & x_{37} & x_{38} & x_{39} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} & x_{47} & x_{48} & x_{49} \end{bmatrix}_{2\times 9}$$
$$\begin{bmatrix} x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} & x_{57} & x_{58} & x_{59} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} & x_{67} & x_{68} & x_{69} \end{bmatrix}_{2\times 9}$$
$$\begin{bmatrix} x_{71} & x_{72} & x_{73} & x_{74} & x_{75} & x_{76} & x_{77} & x_{78} & x_{79} \end{bmatrix}_{1\times 9}$$
$$\begin{bmatrix} x_{81} & x_{82} & x_{83} & x_{84} & x_{85} & x_{86} & x_{87} & x_{88} & x_{89} \end{bmatrix}_{1\times 9}$$
$$\begin{bmatrix} x_{91} & x_{92} & x_{93} & x_{94} & x_{95} & x_{96} & x_{97} & x_{98} & x_{99} \end{bmatrix}_{1\times 9}$$

This is a *load balanced* approach, where we try to give each processor roughly the same amount of stuff. Of course, that is not part of the rules of the SPMD structure, so we could just as well

[4] Palette selected to be distinguishable by the color blind, taken from http://jfly.iam.u-tokyo.ac.jp/color/#pallet

distribute the data like so:

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

With local storage view:

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

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

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

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

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

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

Generally, we would recommend using a load balanced approach over this bizarre distribution, although some problems may call for very strange data distributions. For example, it is possible and common to have an empty matrix after some subsetting or selectation.

With our first of two cumbersome data structures out of the way, we can proceed to much more interesting content: actually using MPI.

3.4 Common MPI Operations

Fully explaining the process of MPI programming is a daunting task. Thankfully, we can punt and merely highlight some key MPI operations and how one should use them with **pbdMPI**.

3.4.1 Basic Communicator Wrangling

First things first, we must examine basic communicator issues, like construction, destruction, and each processor's position within a communicator.

- **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**: No processor can proceed until *all* processors can proceed.
  `barrier()` — "computation wall" that only all processors together can tear down.

One quick word before proceeding. If a processor queries `comm.size()`, this will return the total number of processors in the communicators. However, communicator indexing is like indexing in the programming language C. That is, the first element is numbered 0 rather than 1. So when the first processor queries `comm.rank()`, it will return 0, and when the last processor queries `comm.rank()`, it will return `comm.size() - 1`.

We are finally ready to write our first MPI program:

Simple pbdMPI Example 1

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

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

finalize()
```

Unfortunately, it is not very exciting, but you have to crawl before you can drag race. Remember that all of our programs are written in SPMD style. So this *one* single program is written, and each processor will execute the same program, but with different results, whence the name "Single Program/Multiple Data".

So what does it do? First we initialize the MPI communicator with the call to `init()`. Next, we have each processor query its rank via `comm.rank()`. Since indexing of MPI communicators starts at 0, we add 1 because that is what we felt like doing. Finally we call R's `print()` function to print the result. This printing is not particularly clever, and each processor will be clamoring to dump its result to the output file/terminal. We will discuss more sophisticated means of printing later. Finally, we shut down the MPI communicator with `finalize()`.

If you were to save this program in the file `mpiex1.r` and you wished to run it with 2 processors, you would issue the command:

Shell Command

```
### (Use Rscript.exe for windows system)
mpiexec -np 2 Rscript mpiex1.r
```

To use more processors, you modify the `-np` argument ("number processors"). So if you want to use 4, you pass `-np 4`.

The above program technically, though not in spirit, bucks the trend of officially opening with a "Hello World" program. So as not to incur the wrath of the programming gods, we offer a simple such example by slightly modifying the above program:

Simple pbdMPI Example 1.5

```
1  library(pbdMPI, quiet = TRUE)
2  init()
3
4  myRank <- comm.rank()
5
6  if (myRank == 0){
7    print("Hello, world.")
8  }
9
10 finalize()
```

One word of general warning we offer now is that we are taking very simple approaches here for the sake of understanding, heavily relying on function argument defaults. However, there are all kinds of crazy, needlessly complicated things you can do with these functions. See the **pbdMPI** reference manual for full details about how one may utilize these (and other) **pbdMPI** functions.

3.4.2 Reduce, Broadcast, and Gather

Once managing a communicator is under control, you presumably want to do things with all of your processors. The typical way you will have the processors interact is given below:

- **Reduction**: Say each processor has a number `x.spmd`. Add all of them up, find the largest, find the smallest, ...
  `reduce(x.spmd, op='sum')` — only one processor gets result (default is 0)
  `allreduce(x.spmd, op='sum')` — every processor gets result
- **Gather**: Say each processor has a number. Create a new object on some processor(s) 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)`

Here perhaps explanations are inferior to examples; so without wasting any more time, we proceed to the next example:

Simple pbdMPI Example 2

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

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

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

gt <- allgather(n.spmd)
print(gt)

finalize()
```

So what does it do? First each processor samples a number from 1 to 10; it is probably true that each processor will be using a different seed for this sampling, though you should not rely on this alone to ensure good parallel seeds. More on this particular problem in Section 3.4.3 below.

Next, we perform an `allreduce()` on `n.spmd`. Conceivably, the processors could have different values for `n.spmd`. So the value of `n` is local to each processor. So perhaps we want to add up all these numbers (with as many numbers as there are processors) and store them in the global value `sm` (for "sum"). Each processor will agree as to the value of `sm`, even if they disagree about the value of `n.spmd`.

Finally, we do the same but with an `allgather()` operation.

Try experimenting with this by running the program several times. You should get different results each time. To make sure we have not been lying to you about what is happening, you can even print the values of `n.spmd` before the reduce and gather operations.

3.4.3 Printing and RNG Seeds

In addition to the above common MPI operations, which will make up the bulk of the MPI programmer's toolbox, we offer a few extra utility functions:

- **Print**: printing with control over which processor prints.
  `comm.print(x, ...)`
  `comm.cat(x, ...)`
- **Random Seeds**:
  `comm.set.seed(seed, diff=FALSE)`: every processor uses the seed `seed`
  `comm.set.seed(diff=TRUE)`: every processor uses an independent seed (via **rlecuyer**)

The `comm.print()` and `comm.cat()` functions are especially handy, because they are much more sophisticated than their R counterparts when using multiple processes. These functions which processes do the printing, and if you choose to have all processes print their result, then the printing occurs in an orderly fashion, with processor 0 getting the first line, processor 1 getting the second, and so on.

For example, revisiting our "Hello, world" example, we can somewhat simplify the code with a slight modification:

Simple pbdMPI Example 3

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

myRank <- comm.rank()

comm.print("Hello, world.")

finalize()
```

If we want to see what each processor has to say, we can pass the optional argument `all,rank=TRUE` to `comm.print()`. By default, each process will print its rank, then what you told it to print. You can suppress the printing of communicator rank via the optional argument `quiet=TRUE` to `comm.print()`.

These functions are quite handy...

HOWEVER

these functions are potentially dangerous, and so some degree of care should be exercised. Indeed, it is possible to lock up all of the active R sessions by incorrectly using them. Worse, achieving this behavior is fairly easy to do. The way this occurs is by issuing a `comm.print()` on an expression which requires communication. For example, suppose we have a distributed object with local piece `x.spmd` and a function `myFunction()` which requires communication between the processors. Then calling

A Cautionary Tale of Printing in Parallel (1 of 3)

```
print(myFunction(x.spmd))
```

is just fine, but will not have the nice orderly, behaved printing style of `comm.print()`. However, if we issue

A Cautionary Tale of Printing in Parallel (2 of 3)

```
comm.print(myFunction(x.spmd))
```

then we have just locked up all of the R processes. Indeed, behind the scenes, a call somewhat akin to

```
for (rank in 0:comm.size()){
  if (comm.rank() == rank){
    # do things
  }
  barrier()
}
```

has been ordered. The problem arises in the "do things" part. Since (in our hypothetical example) the function `myFunction()` requires communication between the processors, it will simply wait forever for the others to respond until the job is killed. This is because the other processors skipped over the "do things" part and are waiting at the barrier. So lonely little processor 0 has been stood up, unable to communicate with the remaining processors.

To avoid this problem, make it a personal habit to only print on *results*, not *computations*. We can quickly rectify the above example by doing the following:

A Cautionary Tale of Printing in Parallel (3 of 3)

```
myResult <- myFunction(x.spmd)
comm.print(myResult)
```

In short, printing stored objects is safe. Printing a yet-to-be-evaluated expression is not safe.

3.4.4 Apply, Lapply, and Sapply

But the **pbdMPI** sugar extends to more than just printing. We also have a family of "*ply" functions, in the same vein as R's `apply()`, `lapply()`, and `sapply()`:

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

Here is a simple example utilizing `pbdLapply()`:

Example 4

```
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()
```

So what does it do? Why don't you tell us? We're busy people, after all, and we're not going to be around forever. Try guessing what it will do, then run the program to see if you are correct. As you evaluate this and every parallel code, ask yourself which pieces involve communication and which pieces are local computations.

3.5 Timing MPI Tasks

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:

- the minimum elapsed time measured across all processes,
- the average elapsed time measured across all processes, and
- the maximum elapsed time across all processes.

The code for this function is listed below:

Timer Function

```
1  demo.timer <- function(timed)
2  {
3    ltime <- system.time(timed)[3]
4
5    mintime <- allreduce(ltime, op='min')
6    maxtime <- allreduce(ltime, op='max')
7
8    meantime <- allreduce(ltime, op='sum') / comm.size()
9
10   return( c(min=mintime, mean=meantime, max=maxtime) )
11 }
```

3.6 Exercises

3-1 Write a script that will have each processor randomly take a sample of size 1 of `TRUE` and `FALSE`. Have each processor print its result.

3-2 Modify the script in Exercise 3-1 above to determine if any processors sampled `TRUE`. Do the same to determine if all processors sampled `TRUE`. In each case, print the result. Compare to the functions `comm.all()` and `comm.any()`. Hint: use `allreduce()`.

3-3 In **pbdMPI**, there is a parallel sorting function called `comm.sort()` which is similar to the usual `sort()` of R. Implement parallel equivalents to the usual `order()` and `rank()` of R.

3-4 Time the performance of Exercise 3-3. Identify the need of MPI communications for different sorting or ordering algorithms.

3-5 There are "parallel copycat" versions of to R's *ply functions in several R packages, such as `mclapply()` (a parallel version of `lapply()`) in the **parallel** package. Try to compare the difference and performance of those *ply-like functions.

4 Basic Statistics Examples

This section introduces a few simple examples and explains a little about computing with distributed data directly over MPI. 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`.

4.1 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

$$\pi \approx 4\frac{L}{N} \tag{4.1}$$

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

$$x_i^2 + y_i^2 \leq 1 \tag{4.2}$$

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 *directly* approximating an area through this sampling scheme, because to do so with a finite-point sampling scheme would be madness requiring a transfinite process. Indeed, let S_N be the collection of elements satisfying inequality (4.2). Then note that for each $N \in \mathbb{N}$ that the area of S_N is precisely 0. Whence,

$$\lim_{N \to \infty} Area(S_N) = 0$$

This bears repeating. Finite sampling of an uncountable space requires uncountably many such sampling operations to "fill" the infinite space. For a proper treatment of set theoretic constructions, including infinite cardinals, see (Kunen, 1980).

One could argue that we are evaluating a ratio of integrals with each using the counting measure, which satisfies technical correctness but is far from clear. Now indeed, certain facts of area are vital here, but some care should be taken in the discussion as to what exactly justifies our claim in (4.1).

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

$$\bar{X}_N \xrightarrow{a.s.} \frac{\pi}{4} \quad \text{as } N \to \infty \tag{4.3}$$

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 the author a grumpy old man, the misleading picture you desire can be found in Figure 4.1. 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:

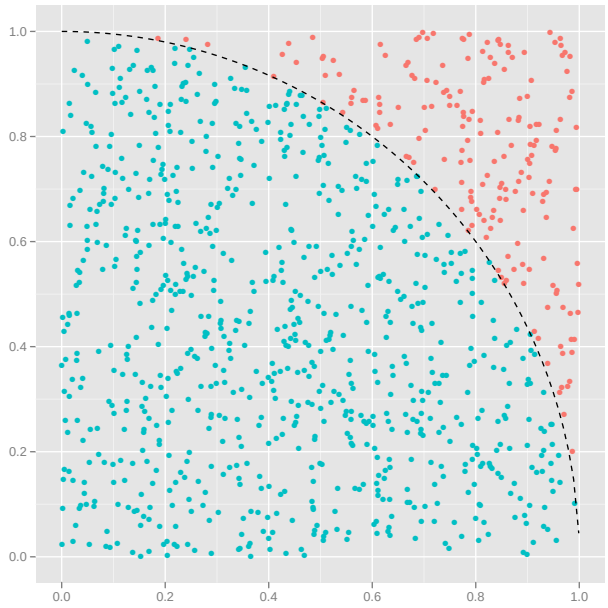


Figure 4.1: Approximating π by Monte Carlo methods

R Code

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

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 the "radii" 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 (4.1). Since SPMD, `ret` is common on all processors, and so is `PI`.

4.2 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:

$$\begin{aligned} \bar{x} &= \frac{1}{N}\sum_{n=1}^{N} x_n \\ &= \sum_{n=1}^{N} \frac{x_n}{N} \end{aligned} \tag{4.4}$$

and

$$\begin{aligned} s_x &= \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \bar{x})^2 \\ &= \frac{1}{N-1}\sum_{n=1}^{N} x_n^2 - \frac{2\bar{x}}{N-1}\sum_{n=1}^{N} x_n + \frac{1}{N-1}\sum_{n=1}^{N} \bar{x}^2 \\ &= \sum_{n=1}^{N}\left(\frac{x_n^2}{N-1}\right) - \frac{N\bar{x}^2}{N-1} \end{aligned} \tag{4.5}$$

where expressions (4.4) and (4.5) are one-pass algorithms, which are potentially faster than the first expressions, especially for large N. However, this method of computing the variance in one pass can suffer from round-off errors, and so in general is not numerically stable. We provide this here for demonstration purposes only. Additionally, only the first and second moments are implemented, while the extension of one-pass algorithms to higher order moments is also possible.

The demo generates random data on 4 processors, then utilizes the `mpi.stat()` function:

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.*, 2012b) can be utilized in this examples to aggregate local information across all processors.

4.3 Binning

Example: Find binning counts 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(binning,'pbdDEMO',ask=F,echo=F)"
```

Binning is a classical problem in statistics which helps to quickly summarize the data structure by setting some "breaks" between the minimum and maximum values. This is a particularly useful tool for constructing histograms, as well as categorical data analysis.

The demo generates random data on 4 processors, then utilizes the `mpi.bin()` function:

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().
```

This simple implementation utilizes R's own `table()` function to obtain local counts, then calls `allreduce()` to obtain global counts on all processors.

4.4 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)"
```

Another fundamental tool in the statistician's toolbox is finding quantiles. Quantiles are points taken from the cumulative distribution function. Formally, a *q-quantile* (or *q-tile*) with $q \in [0, 1]$

of a random variable X is any value θ_q such that[1]

$$P(X \leq \theta_q) \geq q \qquad \text{and}$$
$$P(X \geq \theta_q) \leq 1 - q$$

Note that by this definition, a quantile neither need exist or be unique. Indeed, for the former, consider the standard normal distribution with $q = 1$, and for the latter consider the probability 0 values of a uniform distribution. Perhaps to narrow the scope of these problems, another common definition is

$$\theta_q = \inf\{x \mid P(X \leq x) \geq q\}$$

In this example, we will be estimating quantiles from a sample. Doing so requires sub-dividing the data into q (almost) evenly sized subsets, giving rise to the language k'th q-tile, for integers $0 < k < \frac{1}{q}$.

Before proceeding, we wish to make very clear the distinction between a theoretical quantile and quantile estimation, as many web pages confuse these two topics. A quantile estimation from a sample requires ordering and can take many forms; in fact, there are nine possible such forms in R's own `quantile()` function (see `help(quantile)` in R). The definitions of Kendall and Cramer may be the source of all the confusion (Benson, 1949). Kendall's definition, conflating the term "quantile" with the act of quantile estimation, seems to have entered most dictionaries (and Wikipedia), whereas mathematical statistics favors the more general and simple definition of Cramer.

This example can be extended to construct Q-Q plots, compute cumulative density function estimates and nonparametric statistics, as well as solve maximum likelihood estimators. This is perhaps 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 at a large scale. The demo generates random data on 4 processors, then utilizes the `mpi.quantile()`:

R Code

```
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().
```

[1] This definition is due to the mathematical statistician Herman Rubin: http://mathforum.org/kb/message.jspa?messageID=406278

Here, a numerical function is solved by using `uniroot()` to find out the appropriate value where the cumulative probability is less than or equal to the specified quantile. Specifically, it finds the zero, or root, of the monotone `f.quantile()` function. This simple example shows that with just a little effort, direct MPI methods are greatly applicable on large scale data analysis and likelihood computing.

Note that in the way that the `uniroot()` call is used above, we are legitimately operating in parallel and on distributed data. Other optimization functions such as `optim()` and `nlm()` can be utilized in the same way.

4.5 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

$$||\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}||_2^2 \tag{4.6}$$

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

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{4.7}$$

where $\boldsymbol{y}$ is $N \times 1$ observed vector, $\boldsymbol{X}$ is $N \times p$ (possibly designed) matrix which is often assumed to have full rank (more on that later), and $N >> p$, $\boldsymbol{\beta}$ is the unknown parameter 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. For many applications a statistician will face, expression (4.6) will actually have a unique solution. But this is not always the case, and trouble often arises when the model matrix is rank-deficient. Indeed, in this case 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 may 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.

In the full rank case, we can provide an analytical, "closed-form" solution to the problem. In this case, the classical is given by:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{4.8}$$

This example can be also generalized to weighted least squares (WLS), and linear mixed effect models. See http://en.wikipedia.org/wiki/Least_squares and http://en.wikipedia.org/wiki/Mixed_model for more details.

The implementation is straight forward:

R Code

```
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
```

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. This is, in large part, because the operation is not "fully parallel", in that the solution is serial and replicated on all processors. Worse, directly computing

$$\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}$$

has numerical stability issues. Instead, it is generally better (although much slower) to take an orthogonal factorization of the data matrix. See Appendix A for details.

Finally, all of the above assumes that the model matrix $\boldsymbol{X}$ is full rank. However, we have implemented an efficient method of solving linear least squares problems in **pbdDMAT**'s `lm.fit()` method for distributed matrices. This method uses a fully parallel rank-revealing QR Decomposition to find the least squares solution. So for larger problems, and especially those where numerical accuracy is important or rank-degeneracy is a possibility, it is much better to simply convert `y.spmd` and `X.spmd` into the block-cyclic format as in Part III and utilize **pbdBASE** and **pbdDMAT** for all matrix computations.

4.6 Distributed Logic

Example: Manage comparisons across all MPI processes.

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(comparators,'pbdDEMO',ask=F,echo=F)"
```

This final MPI example is not statistical in nature, but is very useful all the same, and so we include it here. The case frequently arises where the MPI programmer will need to do logical

comparisons across all processes. The idea is to extend the very handy `all()` and `any()` base R functions to operate similarly on distributed logicals.

You could do this directly. Say you want to see if any processes have `TRUE` stored in the variable `localLogical`. This amounts to something on the order of:

R Code

```
globalLogical <- as.logical(allreduce(localLogical, op='max')
```

Or you can use the function `comm.any()` from **pbdMPI**:

R Code

```
globalLogical <- comm.any(localLogical)
```

which essentially does the same thing, but is more concise. Likewise, there is a `comm.all()` function, which in the equivalent "long-form" above would use `op='min'`.

The demo for these functions consists of two parts. For the first, we do a simple demonstration of how these functions behave:

R Code

```
rank <- comm.rank()

comm.cat("\ntest value:\n", quiet=T)
test <- (rank > 0)
comm.print(test, all.rank=T, quiet=T)

comm.cat("\ncomm.all:\n", quiet=T)
test.all <- comm.all(test)
comm.print(test.all, all.rank=T, quiet=T)

comm.cat("\ncomm.any:\n", quiet=T)
test.any <- comm.any(test)
comm.print(test.any, all.rank=T, quiet=T)
```

which should have the output:

```
test value:
[1] FALSE
[1] TRUE
[1] TRUE
[1] TRUE

comm.all:
[1] FALSE
[1] FALSE
[1] FALSE
[1] FALSE
```

```
comm.any:
[1] TRUE
[1] TRUE
[1] TRUE
[1] TRUE
```

The demo also has another use case which could be very useful to a developer. You may be interested in trying something on only one processor and then shutting down all MPI processes if problems are encountered. To do this in SPMD style, you can create a variable on all processes to track whether a problem has been encountered. Then after critical code sections, use `comm.any()` to update and act appropriately. A very simple example is provided below.

R Code

```
need2stop <- FALSE

if (rank==0){
  need2stop <- TRUE
}

need2stop <- comm.any(need2stop)

if (need2stop)
  stop("Problem :[")
```

4.7 Exercises

4-1 What are the assumptions used in order to invoke the Strong Law of Large Numbers (SLLN) in statement (4.3)?

4-2 What is the Weak Law of Large Numbers (WLLN)? Prove that the SLLN implies the WLLN. Provide a counter example that the WLLN does not imply the SLLN.

4-3 In statement (4.3), we showed that $\bar{X}_N$ converges to $\frac{\pi}{4}$ almost surely, a very strong form of convergence. Show that additionally, $\bar{X}_N$ converges to $\frac{\pi}{4}$ in probability by the WLLN, and that the sequence converges in distribution. (This can be as simple or as complicated as you like, depending on how many big theorems you wish to invoke).

4-4 Let $g : [0, 1] \to \mathbb{R}$ be any continuous function, and let $\bar{X}_N$ be as in statement (4.3). Show that $g(\bar{X}_N)$ converges to $g\left(\frac{\pi}{4}\right)$ almost surely (Hint: use the property of continuity with respect to limits of sequences and the definition of almost sure convergence).

4-5 What are assumptions for Statement (4.7)?

4-6 Prove that $\hat{\boldsymbol{\beta}}$ of statement (4.8) is an unbiased estimator of $\boldsymbol{\beta}$, i.e., show that $E[\hat{\boldsymbol{\beta}}] = \boldsymbol{\beta}$.

4-7 Prove $\boldsymbol{X}^\top \boldsymbol{X}$ is non-negative definite if $\boldsymbol{X}$ has full column rank p (and whence in this case,

the inverse exists).

4-8 Iteratively Reweighted Least Squares (IRLS) is an important method for finding solutions to generalized linear models (GLM)[2]. A common application of GLM's is logistic regression[3].

Implement a (not necessarily numerically stable) logistic regression function using IRLS for SPMD data. For simplicity, you may wish to assume that the weighted matrix $\boldsymbol{X}^T\mathbf{W}\boldsymbol{X}$ is full rank at each iteration.

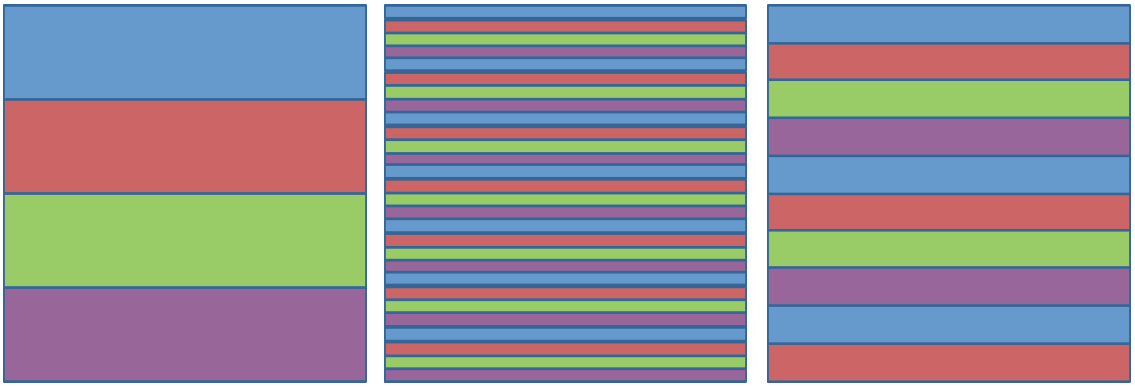
[2]See http://en.wikipedia.org/wiki/Generalized_linear_model for details
[3]See http://stat.psu.edu/~jiali/course/stat597e/notes2/logit.pdf

Part III

Distributed Matrix Methods

5 The Distributed Matrix Data Structure

Before continuing, we must spend some time describing a new distributed data structure. In reality, this data structure is the merging of two different kinds of distributed data structures, namely *block distributions* and *cyclic distributions*. Eventually we will get to *block cyclic distributions*, but this structure is complicated enough that it is wise to examine each component separately first.



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

Figure 5.1: Matrix Distribution Schemes

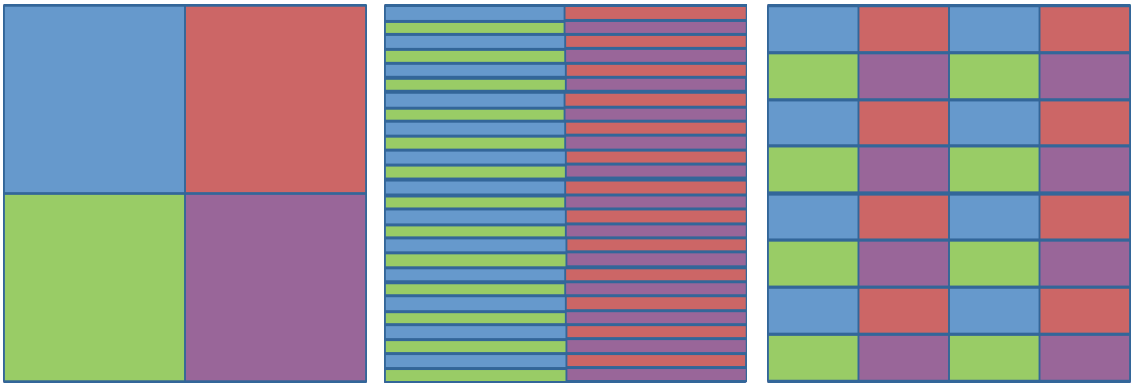
Figure 5.1 shows examples of the three different distribution schemes for 4 processors. The block scheme is simple enough; imagine chopping the matrix into nearly equal blocks and distributing those blocks to different processors. This can be viewed as a special case of the `SPMD` data structure of Section 3.3.

For the cyclic distribution scheme, one can imagine taking each row (or column) of a matrix and sending the first to one processor, the second to the next, and so on until all processors are exhausted; if the data is not exhausted, then one merely cycles back through the processors, continuing in this fashion until all of the matrix has been distributed.

Finally, the block-cyclic decomposition is the obvious blending of these two schemes, so that each of the former becomes a special case of this new type. Here, we can imagine chopping the

matrix up into blocks, but the blocks are not (necessarily) so large that they use up the entire matrix. Once we use up all of the blocks, we use the cyclic data distribution scheme to cycle back through our processors, only using (potentially) blocks of more than one row at a time. From this light, a block-cyclic distribution where the block size is large enough to get all of the data in one cycle is also a block distribution, and a block-cyclic distribution where the blocks are taking just one row at a time is also a cyclic distribution.

The obvious analogue to Figure 5.1 for distributing by column is also possible, but there is a much more important — and complicated — generalization of this scheme. Above, we were thinking of the aggregate of processors as essentially being in a vector, or lying on a one-dimensional line. However, we can extend this to two-dimensional grids of processors as well. Figure 5.2



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

Figure 5.2: Matrix Distribution Schemes Onto a 2-Dimensional Grid

shows how the extension to a 2-dimensional grid of processors, still with just 4 processors, only here, we are assuming that they form a 2×2 grid. This data structure is a generalization of the 1-dimensional block-cyclic distribution, and so it is a generalization of 1-dimensional block and 1-dimensional cyclic distributions as well.

The data structure can get quite complicated, especially when there are many processors involved. Table 5.1 shows the different possible grid shapes for six processors. In general, if we

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

Table 5.1: Processor Grid Shapes with 6 Processors

have n processors, then there are $\sigma_0(n)$ total possible grid shapes, where

$$\sigma_m(n) = \sum_{d \mid n} d^m$$

and $d \in \mathbb{N}$ (a positive integer). Thus the grid shapes are given by:

$$\left(d, \frac{n}{d}\right)$$

for each $d \mid n$ with $d \in \mathbb{N}$.

This added complication is not for pure masochism; it has some real advantages. For one, this 2-dimensional block-cyclic (henceforth simply referred to as "block-cyclic") decomposition is the data structure employed by the state of the art dense linear algebra library ScaLAPACK, and if one wishes to use this library, then the use must occur on its terms. However, there are some real performance benefits to this data structure. For many linear algebra operations (which includes many statistical operations, in whole or in part), this data structure offers an interesting balance between communication cost and parallelism. For very large problems, many are surprised to find that communication between processors will often dwarf the computation overhead. This will generally become apparent at the 10,000+ processor count except for the most embarrassingly parallel problems, and the cost of communication gets *much* worse the more cores are added after that. The rate at which this scales badly will depend a great deal on the hardware, but there is no machine in existence at the time of writing for which the above vague warning will not hold true.

Returning to the data structure, notice that since we have control over the processor grid shape and the blocking factor (or blocking dimension — the number of rows/columns in the blocks for the block-decomposition), we can very directly tune the amount of parallelism, and therefore the amount of communication. Make the blocks too small (say 1×1, or single element blocks) and there will be a great deal of parallelism, in the sense that most processors will stay busy most of the time; but the processors will have to talk to each other to get *anything* done. This makes the communication cost skyrocket. On the other hand, we could make the blocking factor so large in each dimension that it encompasses the entire matrix. That is, the matrix would be stored in its entirety on a single processor. In doing so, we entirely eliminate the communication, but we also elimination the parallelism.

The fact of the matter is, hard problems require data movement and communication. We should strive to minimize these burdens, but not so myopically that we throw out the parallelism as well. Balancing these parameters then becomes important, and a not entirely trivial optimization problem. The **pbdDMAT** package includes defaults for each that should be "ok" if you have no intuition whatsoever. However, these defaults may not be well-suited to a specific problem, and knowing ahead of time how best to distribute the data is often more art than science.

For the remainder of this chapter, we will be examining these shapes in more depth to get a better feel for the data structure. To do so, let us return to our old friend from Section 3.3:

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

However, we note that the **pbdDMAT** package offers numerous high-level tools for managing these structures, so that the management of distributed details can be as implicit or explicit as the user desires.

5.1 Block Data Distributions

Let us start with the 1-dimensional block data distribution. So here, we will assume that our processor grid looks like:

$$\text{Processors} = \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}$$

To block-distribute our matrix onto this 1-dimensional grid by rows, then we would have no option but to do the following:

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

Notice that here, processor 3 receives none of the matrix. This is so because if the block size (here, 3×9) were any smaller, then we would not be able to distribute all of the data without cycling. Similarly, if we were to distribute by column then we would have:

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

for exactly the same reason.

If we used a 2-dimensional grid of processors, say a 2×2 grid:

$$\text{Processors} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

then our data would be distributed as

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

5.2 Cyclic Data Distributions

Proceeding as in the previous section, we would cyclically distribute this matrix by row onto the 1-dimensional processor grid as:

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

and by column:

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

Finally, the distribution onto the 2-dimensional grid would look like:

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

5.3 Block-Cyclic Data Distributions

By this time, the reader should feel fairly comfortable with the basic idea and the distribution scheme. So we will jump straight to full generality. To make things more interesting (really, to show the full generality of the distribution), let us now suppose that we have 6 processors in a 2×3 grid:

$$\text{Processors} = \left[\begin{array}{ccc} 0 & 1 & 2 \\ 3 & 4 & 5 \end{array}\right] = \left[\begin{array}{ccc} (0,0) & (0,1) & (0,2) \\ (1,0) & (1,1) & (1,2) \end{array}\right]$$

with the usual MPI processor rank on the left, and the corresponding BLACS processor grid position on the right. This new naming convention is just for convenience of describing a processor by its position in the grid and carries no additional semantic meaning. We will preserve our 2×2 dimensional blocking factor.

Recall that to distribute this data across our 6 processors in the form of a 2×3 process grid in 2×2 blocks, we go in a "round robin" fashion, assigning 2×2 submatrices of the original matrix to the appropriate processor, starting with processor $(0, 0)$. Then, if possible, we move on to the next 2×2 block of x and give it to processor $(0, 1)$. We continue in this fashion with $(0, 2)$ if necessary, and if there is yet more of x in that row still without ownership, we cycle back to processor $(0, 0)$ and start over, continuing in this fashion until there is nothing left to distribute in that row.

After all the data in the first two rows of x has been chopped into 2-column blocks and given to the appropriate process in process-column 1, we then move onto the next 2 rows, proceeding in the same way but now using the second process row from our process grid. For the next 2 rows, we cycle back to process row 1. And so on and so forth.

Then distributed across processors, the data will look like:

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

with local storage:

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

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

You *could* use some more natural data distributions than the above, such as the block data structure. However, this may have a substantial impact on performance, depending on the kinds of operations you wish to do. For things that make extensive use of linear algebra — particularly matrix factorizations — you are probably much better off using the above kind of block-cyclic data distribution. Sometimes there is a benefit to using a 1-dimensional grid of processors while still using the full block-cyclic structure. These different processor grid shapes are referred to as *contexts*. They are actually specialized MPI communicators. By default, the recommended (easy) way of managing these contexts with **pbdDMAT** is to call

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

The call to `init.grid()` will initialize three such contexts, named 0, 1, and 2. Context 0 is a communicator with processors as close to square as possible, like above. This can be confusing if you ever need to directly manipulate this data structure, but **pbdDMAT** contains *numerous* helper methods to make this process painless, often akin to manipulating an ordinary, non-distributed R data structure. Context 1 puts the processors in a 1-dimensional grid consisting of 1 row. Continuing with our example, the processors form the grid:

$$\text{Processors} = \left[\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array}\right] = \left[\begin{array}{cccccc} (0,0) & (0,1) & (0,2) & (0,3) & (0,4) & (0,5) \end{array}\right]$$

and if we preserve the 2×2 blocking factor, then the data would be distributed like so:

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

Locally, the data is stored as follows:

$$\left[\begin{array}{cc} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \\ x_{51} & x_{52} \\ x_{61} & x_{62} \\ x_{71} & x_{72} \\ x_{81} & x_{82} \\ x_{91} & x_{92} \end{array}\right]_{9\times 2} \left[\begin{array}{cc} x_{13} & x_{14} \\ x_{23} & x_{24} \\ x_{33} & x_{34} \\ x_{43} & x_{44} \\ x_{53} & x_{54} \\ x_{63} & x_{64} \\ x_{73} & x_{74} \\ x_{83} & x_{84} \\ x_{93} & x_{94} \end{array}\right]_{9\times 2} \left[\begin{array}{cc} x_{15} & x_{16} \\ x_{25} & x_{26} \\ x_{35} & x_{36} \\ x_{45} & x_{46} \\ x_{55} & x_{56} \\ x_{65} & x_{66} \\ x_{75} & x_{76} \\ x_{85} & x_{86} \\ x_{95} & x_{96} \end{array}\right]_{9\times 2} \left[\begin{array}{cc} x_{17} & x_{18} \\ x_{27} & x_{28} \\ x_{37} & x_{38} \\ x_{47} & x_{48} \\ x_{57} & x_{58} \\ x_{67} & x_{68} \\ x_{77} & x_{78} \\ x_{87} & x_{88} \\ x_{97} & x_{98} \end{array}\right]_{9\times 2} \left[\begin{array}{c} x_{19} \\ x_{29} \\ x_{39} \\ x_{49} \\ x_{59} \\ x_{69} \\ x_{79} \\ x_{89} \\ x_{99} \end{array}\right]_{9\times 1} \left[\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \end{array}\right]_{0\times 1}$$

Here, the first dimension of the blocking factor is irrelevant. All processors own either some part of *all* rows, or they own nothing at all. So the above would be the exact same data distribution if we had a blocking factor of 100×2 or 2×2. However, the decomposition is still block-cyclic; here we use up everything before needing to cycle, based on our choice of blocking factor. If we instead chose a 1×1 blocking, then the data would be distributed like so:

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

Finally, there is context 2. This is deceivingly similar to the SPMD data structure, but the two are, in general, not comparable. This context puts the processors in a 1-dimensional grid consisting of one column (note the transpose):

$$\text{Processors} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \end{bmatrix}^T = \begin{bmatrix} (0,0) & (1,0) & (2,0) & (3,0) & (4,0) & (5,0) \end{bmatrix}^T$$

So here, the data would be decomposed as:

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

with local storage view:

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

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

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

$$\left[\begin{array}{ccccccccc} x_{71} & x_{72} & x_{73} & x_{74} & x_{75} & x_{76} & x_{77} & x_{78} & x_{79} \\ x_{81} & x_{82} & x_{83} & x_{84} & x_{85} & x_{86} & x_{87} & x_{88} & x_{89} \end{array}\right]_{9\times 2}$$

$$\left[\begin{array}{ccccccccc} x_{91} & x_{92} & x_{93} & x_{94} & x_{95} & x_{96} & x_{97} & x_{98} & x_{99} \end{array}\right]_{9\times 1}$$

$$\left[\begin{array}{c} \end{array}\right]_{1\times 0}$$

5.4 Summary

This 2-dimensional block-cyclic data structure — the `DMAT` data structure — is fairly complicated, but can pay great dividends if some appreciation and understand is given to it. To briefly summarize this 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 blocking factor used.
4. Processor $0 = (0, 0)$ will always own at least as much data as every other processor.
5. `DMAT` is locally column-major and globally, it depends...
6. `DMAT` is confusing, but very robust and useful for matrix algebra (and thus most non-trivial statistics).

The only items in common between SPMD and DMAT are items 1 and 2. A full characterization can be given as follows. Let X be a distributed matrix with n (global) rows and p (global) columns. Suppose we distribute this matrix onto a set of *nprocs* processors in context 2 using a blocking factor $b = (b_1, b_2)$. Then DMAT is a special case of SPMD *if and only if* we have $b_1 > \frac{n}{nprocs}$. Otherwise, there is no relationship between these two structures (and converting between them can be difficult).

In the chapters to follow, we offer numerous examples utilizing this data structure. The dedicated reader can find more information about these contexts and utilizing the DMAT data structure, see the **pbdBASE** (Schmidt *et al.*, 2012c) and **pbdDMAT** (Schmidt *et al.*, 2012d) vignettes. Additionally, you can experiment more with different kinds of block-cyclic data distributions on 2-dimensional processor grids using a very useful website at `http://acts.nersc.gov/scalapack/hands-on/datadist.html`.

5.5 Exercises

5-1 Read two papers given at `http://acts.nersc.gov/scalapack/hands-on/datadist.html`. "The Design of Linear Algebra Libraries for High Performance Computers", by J. Dongarra and D. Walker, and "Parallel Numerical Linear Algebra", by J. Demmel, M. Heath, and H. van der Vorst.

6 Constructing Distributed Matrices

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

```
1 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, these higher methods come with a different cost: getting the data into the right format, namely the distributed matrix data structure DMAT, discussed at length in the previous chapter. That said, 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. But to get to the fun stuff, the process of exactly how to decompose data into a block-cyclic distribution must be addressed. We begin dealing with this issue in the simplest way possible.

6.1 Fixed Global Dimension

In these examples, we will examine the case where you know ahead of time what the global number of rows and columns are.

6.1.1 Constructing Simple Distributed Matrices

It is possible to construct fairly simple distributed matrices much in the same way that one can construct simple matrices in R. We can do this using the functions `ddmatrix()` and `as.ddmatrix()`.

The former essentially behaves identically to R's own `matrix()` function. This function takes a global input vector/matrix `data=`, as well as the global number of rows `nrow=` and the global number of columns `ncol=`. Additionally, the user may specify the blocking factor `bldim=` and the BLACS context `CTXT`, and the return is a distributed matrix. For instance, we can specify

ddmatrix()

```
dx <- ddmatrix(data=0, nrow=10, ncol=10)
```

to get a distributed matrix with *global* dimension 10×10 consisting of zeros. We can also do cute things like

ddmatrix()

```
dx <- ddmatrix(data=1:3, nrow=5, ncol=5)
```

which will create the distributed analogue of

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    2    1    3
[2,]    2    1    3    2    1
[3,]    3    2    1    3    2
[4,]    1    3    2    1    3
[5,]    2    1    3    2    1
```

How exactly that "distributed analogue" will look (locally) depends on the processor grid shape (whence too, the number of processors) as well as the blocking factor. This operation performs no communication.

While this can be useful, it is far from the only way to construct distributed matrices. One can also convert a global (non-distributed) matrix into a distributed matrix. There are some caveats; this matrix must either be owned in total by all processors (which is very useful in testing, but should not be used at scale), or the matrix is owned in total by one processor, with all others owning `NULL` for that object.

For example, we can create identical return to the above via

as.ddmatrix()

```
x <- matrix(data=1:3, nrow=5, ncol=5)
dx <- as.ddmatrix(x)
```

or

as.ddmatrix()

```
if (comm.rank()==0){
  x <- matrix(data=1:3, nrow=5, ncol=5)
} else {
  x <- NULL
}
```

```
6
7 dx <- as.ddmatrix(x)
```

Each of these operations performs communication.

Other, more general combinations are possible through other means, but they are much more cumbersome.

6.1.2 Diagonal Distributed Matrices

*Example: construct **diagonal** distributed matrices of specified 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_diag_global,'pbdDEMO',ask=F,echo=F)"
```

In R, the `diag()` function serves two purposes; namely, it is both a reduction operation and a reverse-reduction operation, depending on the input. More specifically, if given a matrix, it produces a vector containing the diagonal entries of that matrix; but if given a vector, it constructs a diagonal matrix whose diagonal is that vector. And so for example, the zero and identity matrices of any dimension can quickly be constructed via:

Diagonal Matrices in R

```
1 diag(x=0, nrow=10, ncol=10) # zero matrix
2 diag(x=1, nrow=10, ncol=10) # identity matrix
```

Both of the above functionalities of `diag()` are available for distributed matrices; however we will only focus on the latter.

When you wish to construct a diagonal distributed matrix, you can easily do so by using the additional `type=` argument to our `diag()` method. By default, `type="matrix"`, though the user may specify `type="ddmatrix"`. If so, then as one might expect, the optional `bldim=` and `ICTXT=` arguments are available. So with just a little bit of tweaking, the above example becomes:

Diagonal Matrices in pbdR

```
1 diag(x=0, nrow=10, ncol=10, type="ddmatrix") # zero
     (distributed) matrix
2 diag(x=1, nrow=10, ncol=10, type="ddmatrix") # identity
     (distributed) matrix
```

In fact, the `type=` argument employs partial matching, so if we really want to be lazy, then we could simply do the following:

Diagonal Matrices in pbdR

```
diag(x=0, nrow=10, ncol=10, type="d") # zero (distributed) matrix
diag(x=1, nrow=10, ncol=10, type="d") # identity (distributed)
   matrix
```

Beyond the above brief explanation, the demo for this functionality is mostly self-contained, although we do employ the `redistribute()` function to fully show off local data storage. This function is explained in detail in Chapter 11.

6.1.3 Random Matrices

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 distribute(s) 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 building a randomly generated distributed matrix, they are not particularly efficient strategies.

As described in the previous section, if we have a matrix `x` stored on processor 0 and `NULL` on the others, then we 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)
```

Here, each processor generates the full, global matrix, then throws away what is not needed. Again, this is not efficient, but the code is concise, so it is extremely useful in testing. 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 **pbdDMAT** package's `ddmatrix()` function. This is "new" behavior for this syntax (if you view `ddmatrix()` as an extension of `matrix()`). Ordinarily you would merely execute something like

Creating a random normal matrix in serial R

```
x <- rnorm(n*p)
x <- matrix(x, nrow=n, ncol=p) # this creates a copy

y <- rnorm(n*p)
dim(y) <- c(n, p) # this does not
```

However, things are slightly more complicated with `ddmatrix` objects, and the user may not easily know ahead of time what the size of the local piece is just from knowing the global dimension. Because this requires a much stronger working knowledge of the underlying data structure than most will be comfortable with, we provide this simple functionality as an extension. However, we note that the disciplined reader is more than capable of figuring out how it functions by examining the source code and checking with the reference manual. 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 "`NUM`ber of `R`ows `O`r `C`olumns." 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). 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.

6.2 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 weak scaling of an algorithm, where different numbers of cores are used with the same local problem size.

To this end, the demo script utilizes the `ddmatrix.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. 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 **pbdDMAT** 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 by the user, 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. To do this, we must find the smallest integer above the specified blocking factor that will divide the number of local rows or columns.

6.3 Exercises

6-1 Random number generation (RNG) is used in this Section such as `rnorm()`. In pbdR, we use an R package **rlecuyer** (Sevcikova and Rossini, 2012) to set different streams of seed in parallel. Try to find and use other RNG methods or implementations in R.

7

Basic Examples

There is a deep part of the author that does not want to begin with these examples. There is a real danger for the cursory observer to see these and hastily conclude that our work, or R as a whole, is merely a "Matlab Clone." Nothing could be further from reality.

Matlab is an amazing product. It costs quite a lot of money; it had better damn well be. However, for statistics, machine learning, data mining — data science — we believe that R is "better." Is R faster? Emphatically, no. But we argue that R wins in other ways.

It is true that everything R can do, so too can Matlab; of course, the converse is also true — that everything Matlab can do, R can do as well. Each is a turing complete language. But being turing complete is not sufficient; LaTeX is turing complete, and yet we do not perform scientific computation in it (although of course it is unparalleled in typesetting). But we could.

The fact that we do not is an extension of the fact that math journals do not publish articles written in C or Fortran. Those programming languages are the wrong mediums of abstraction for expressing highly complicated ideas to domain experts. Only a madman would attempt to express deep mathematical abstraction in these languages for publication (implementation being an entirely separate issue). Likewise, we do not perform our statistical analyses in LaTeX (don't be a pedant; we are not talking about sweave and you know it). People overwhelmingly choose R for the analysis of data because it is the closest brain $\rightarrow$ computer translation available for such problems.

Of course, this goes both ways. If your life is matrix algebra, then R is a much worse fit for you than is Matlab. Much of statistics is applied matrix algebra, but not all matrix algebra is statistics.

So we reluctantly press on with several basic examples utilizing distributed matrices. For meatier examples, see Chapter 8.

7.1 Reductions and Transformations

7.1.1 Reductions

In Section 6.1.2, we discussed the way that the `diag()` method may be utilized as a reduction operator. We have numerous other reductions available, such as `sum()`, `prod()`, `min()`, and `max()`. These operate exactly as their serial counterparts:

Reductions

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

dx <- ddmatrix(data=0:1, nrow=10, ncol=10)

sm <- sum(dx)
comm.print(sm)

pd <- prod(dx)
comm.print(pd)

mn <- min(dx)
comm.print(mn)

mx <- max(dx)
comm.print(mx)

finalize()
```

We also offer some "super reductions". It is possible to change a distributed matrix into a non-distributed matrix or vector using the methods `as.matrix()` or `as.vector()`. For example:

Super Reductions

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

dx <- ddmatrix(data=0:1, nrow=10, ncol=10)
print(dx)

x <- as.matrix(dx)
comm.print(x)

finalize()
```

These can be very useful in testing, but should be used sparingly at scale.

7.1.2 Transformations

We also offer numerous in-place transformations, such as the various `log()` functions, `abs()`, `sqrt()`, `ceiling()`, `floor()`, and `round()`. For example:

Transformations

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

comm.set.seed(diff = TRUE)

dx <- ddmatrix(data=-3:3, nrow=10, ncol=10)

dx <- ceiling(sqrt(abs(dx)))

x <- as.matrix(dx)
comm.print(x)

finalize()
```

7.2 Matrix Arithmetic

We also offer a complete set of methods for distributed matrix arithmetic. With identical syntax to R, we can do some reasonably complicated things, such as:

Transformations

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

dx <- ddmatrix(data=-3:3, nrow=10, ncol=10)
vec <- 1:2

dy <- (dx - vec) %*% dx

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

finalize()
```

For a full list of methods, see the **pbdDMAT** documentation.

One item worth noting is that, as with regular R, if the user wishes to compute $X^T X$ or $X X^T$, then it is usually much faster to use the methods `crossprod()` and `tcrossprod()`, respectively. However, for this operation, things are somewhat more complicated in the distributed sphere

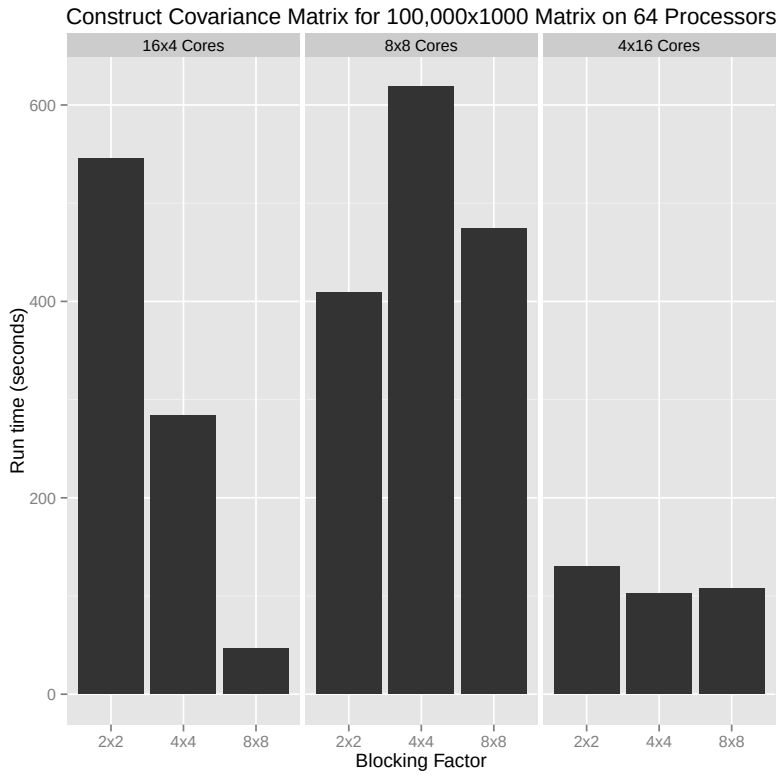


Figure 7.1: Covariance Benchmark Showing Effect of Parameter Calibration

than in serial. Figure 7.1 shows the results of a benchmark of the `cov()` method for computing variance-covariance matrices (which is just a small amount of extra work on top of `crossprod()`). Here, each run consists of 25 replicates of calling `cov()` (which calls `crossprod()`) and then reporting the average run time. The changes in parameters are subtle, but the effects are enormous. Sometimes is may be (much) more beneficial to use `t(x) %*% x`. Others it may not. Proper calibration of these parameters to achieve optimal performance for a given task is still somewhat of an open question to the HPC community.

7.3 Matrix Factorizations

In addition to all of the above, we also provide several of the more important matrix factorizations for distributed matrices. Namely, the singular value decomposition `svd()`/`La.svd()`, QR factorization `qr()`, Cholesky factorization `chol()`, and LU factorization `lu()`. So for example:

Matrix Factorizations

```
1 library(pbdDEMO, quiet = TRUE)
2 init.grid()
3
4 comm.set.seed(diff = TRUE)
5
```

```
dx <- ddmatrix("rnorm", nrow=10, ncol=10, bldim=2)

out <- chol(crossprod(dx))
print(out)

finalize()
```

7.4 Exercises

7-1 Sub-setting, selection and filtering are basic matrix operation featured in R. The next may look silly, but it is useful for data processing. Suppose $\boldsymbol{X}$ is in `ddmatrix` with dimension 97×42, say `dX <- ddmatrix(rnorm(97 * 42), nrow=37)`, do the following:

- ```
  dY <- dX[c(1, 41, 5, 4, 3),]
  dY <- dX[, c(10:3, 5, 5)]
  dY <- dX[3:51, 5:3]
  ```
- ```
  dY <- dX[dX[, 31] > 10,]
  dY <- dX[dX[, 41] > dx[, 40],]
  dY <- dX[, dX[41,] > dx[40,]]
  dY <- dX[dX[, 41] > dx[, 40], c(1, 3, 5)]
  ```
- ```
  dX[c(1, 41, 5, 4, 3),] <- 10
  dX[, c(10:3, 5, 5)] <- 9
  dX[3:51, 5:3] <- 8
  ```
- ```
  dX[dX[, 31] > 0,] <- 7
  dX[dX[, 41] > dx[, 40],] <- 6
  dX[, dX[41,] > dx[40,]] <- 5
  dX[dX[, 41] > dx[, 40], c(1, 3, 5)] <- 4
  ```
- ```
  dX[c(1, 40, 5, 4, 3),] <- dX[c(1, 41, 5, 4, 3) + 1,]
  dX[, c(10:3, 5, 5)] <- dX[, c(10:3, 5, 5) + 1]
  dX[c(10:3, 5, 5),] <- dX[c(10:3, 5, 5) + 1,]
  dX[3:51, 5:3] <- dX[(3:51) + 1, (5:3) + 1]
  ```
- ```
  dX[dX[, 31] > 0,] <- dX[dX[, 31] > 0, c(42, 1:41)]
  dX[dX[, 41] > dx[, 40],] <- dX[dX[, 41] > dx[, 40], c(41:42, 1:40)]
  dX[, dX[41,] > dx[40,]] <- dX[c(96:97, 1:95), dX[, 41] > dx[, 40]]
  dX[dX[, 41] > dx[, 40], c(1, 3, 5)] <- dX[dX[, 41] > dx[, 40], c(1, 3, 5)
  + 1]
  ```

If any of above does not work, please report the bugs.

7-2 Suppose `dX` is as Exercise 7-1, do the following:

- ```
  dY <- dX[-c(1, 41, 5, 4, 3),]
  dY <- dX[, -c(10:3, 5, 5)]
  ```
  ```

```
dY <- dX[-(3:51), -(5:3)]
```

- `dY <- dX[dX[, 41] > dx[, 40], -c(1, 3, 5)]`
- ```
  dX[-c(1, 41, 5, 4, 3),] <- 10
  dX[, -c(10:3, 5, 5)] <- 9
  dX[-(3:51), -(5:3)] <- 8
  ```
- `dX[dX[, 41] > dx[, 40], -c(1, 3, 5)] <- 4`
- ```
  dX[-c(1, 40, 5, 4, 3),] <- dX[-(c(1, 41, 5, 4, 3) + 1),]
  dX[, -c(10:3, 5, 5)] <- dX[, -(c(10:3, 5, 5) + 1)]
  dX[-c(10:3, 5, 5),] <- dX[-(c(10:3, 5, 5) + 1),]
  dX[-(3:51), -(5:3)] <- dX[-((3:51) + 1), -((5:3) + 1)]
  ```
- `dX[dX[, 41] > dx[, 40], -c(1, 3, 5)] <- dX[dX[, 41] > dx[, 40], -(c(1, 3, 5) + 1)]`

7-3 Verify the validity of Exercises 7-1 and 7-2 using ordinary R operations (cast the matrix as global first using `X <- as.matrix(dX)`).

7-4 Implement SPMD row-major matrix format in 2 processors for Exercises 7-1 and 7-2.

8

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.

8.1 Sample Mean and Variance Revisited

Example: Get summary statistics from 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(sample_stat_dmat,'pbdDEMO',ask=F,echo=F)"
```

Returning to the sample statistics problem from Section 4.2, we can solve these same problems — and then some — using distributed matrices. For the remainder, suppose we have a distributed matrix `dx`.

Computing a mean is simple enough. We need only call

Summary Statistics

```
1 mean(dx)
```

We also have access to the other summary statistics methods for matrices, however, such as `rowSums()`, `rowMeans()`, etc. Furthermore, we can calculate variances for distributed matrices. Constructing the variance-covariance matrix is as simple as calling

Summary Statistics

```
cov(dx)
```

Or we could generate standard deviations column-wise, using the method R suggests for ordinary matrices using `apply()`

Summary Statistics

```
apply(dx, MARGIN=2, FUN=sd)
```

or we could simply call

Summary Statistics

```
sd(dx)
```

In R, calling `sd()` on a matrix issues a warning, telling the user to instead use `apply()`. Either of these approaches works with a distributed matrix (with the code as above), but for us, using `sd()` is preferred. This is because, as outlined in Section 11.2, our `apply()` method carries an implicit data redistribution with it, while the `sd()` method is fast, ad-hoc code which requires no redistribution of the data.

8.2 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. This effectively amounts to calling:

Verifying Distributed System Solving

```
# generating data
timer({
  x <- ddmatrix("rnorm", nrow=nrows, ncol=ncols)
  truesol <- ddmatrix(0.0, nrow=nrows, ncol=1)
})

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

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

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

with some added window dressing.

8.3 Compression with 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:

```
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%:

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

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

8.4 Predictions with Linear Regression

Example: Fit a linear regression model and use it to make a prediction on new 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(ols_dmat,'pbdDEMO',ask=F,echo=F)"
```

Suppose we have some predictor variables stored in the distributed $n \times p$ matrix `dx` and a response variable stored in the $n \times 1$ distributed matrix `dy`, and we wish to use the ordinary least squares model from (4.7) to make a prediction about some new data, say `dy.new`. Then this really amounts to just a few simple commands, namely:

```
mdl <- lm.fit(dx, dy)

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

comm.print(submatrix(pred), quiet=T)
```

8.5 Exercises

8-1 Based on Section 8.2, extend the code to find $\boldsymbol{X}$ which solves $\boldsymbol{AX} = \boldsymbol{B}$ where $\boldsymbol{A}$, $\boldsymbol{X}$ and $\boldsymbol{B}$ are matrices with appropriated dimensions and $\boldsymbol{A}$ and $\boldsymbol{B}$ are known.

8-2 The `prcomp()` method introduced in Section 8.3 also returns rotations for all components. Computationally verify with several examples that these rotations are orthogonal, i.e., that their crossproduct is the identity matrix.

8-3 Based on Section 8.4, find a point-wise 95% confidence interval for the observed data $\hat{\boldsymbol{y}}|\boldsymbol{X}$ and a 95% predictive interval for the prediction for a new data $\hat{y}_{new}|\boldsymbol{x}_{new}$.

Part IV

Reading and Managing Data

9 Reading CSV and SQL Files

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?

9.1 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 6.1.3. 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 **pbdDMAT** vignette.

9.2 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.

9.3 Exercises

9-1 In Section 9.1, we have seen an CSV reading example, however, this is not an efficient way for large CSV files by calling `read.csv()`. The R functions `con <- file(...)` can open a connection to the CSV files and `readLines(con, n = 100000)` can access a chunk of data (100,000 lines) from disk more efficiently. Implement a simple function as `read.csv()` and compare performance.

9-2 As Exercise 9-1, implement a simple function by utilizing `writeLines()` for writing large CSV file and compare performance to the `write.csv()`.

9-3 Other R packages can deal with fast reading for CSV format or so in serial. Try **ff** (Adler *et al.*, 2013) and **bigmemory** (Kane and Emerson, 2010).

10 Parallel NetCDF4 Files

10.1 Introduction

Network Common Data Form version 4 (NetCDF4) is a self-describing, machine-independent data format primarily used for very large scale array-oriented scientific data. The NetCDF4 library is available from the Unidata Program at `http://www.unidata.ucar.edu/software/netcdf`. NetCDF4 is built on top of HDF5 data model for extremely large and complex data collections. More specifically, NetCDF4 is a subset of HDF5 but with enhanced usability features. The HDF5 library is available from the HDF Group `http://www.htfgroup.org/HDF5/`.

Both libraries provide high-performance functionality to create, access, read, write, and modify NetCDF4 files. The R package **ncdf4** (Pierce, 2012) provides an R-level interface for NetCDF4 libraries. A short summary of its major functions is given in the Table 10.1

Both NetCDF4 and HDF5 provide the capability for parallel I/O, allowing multiple processors to collectively access the same file. To enable this mechanism, HDF5 and NetCDF4 are required to be compiled and linked against an MPI library. In addition to offering access to collective I/O supported by parallel HDF5 and NetCDF4 libraries, the R package **pbdNCDF4** (Patel *et al.*, 2013a) is a parallel extension of **ncdf4** and provides functions for collectively accessing the same NetCDF4 file by multiple processors at the same time.

Users are encouraged to read the vignette (Patel *et al.*, 2013b) of **pbdNCDF4** which includes information for compiling HDF5 and NetCDF4 in parallel, and demonstration of parallel-enabled functions. Table 10.1 also lists the the major functions of **pbdNCDF4**.

The **pbdDEMO** has an example dataset `TREFHT` from a Community Atmosphere Model (CAM) version 5 simulation output. CAM is a series of global atmosphere models originally developed at the National Center for Atmospheric Research (NCAR) and currently guided by Atmosphere Model Working Group (AMWG) of the Community Earth System Model (CESM) project. CAM version 5 (CAM5) is the latest standalone model modified substantially with a range of enhancements and improvement in the representation of physical processes since version 4 (Eaton, 2011; Vertenstein *et al.*, 2011).

Table 10.1: Functions from **pbdNCDF4** and **ncdf4** for accessing NetCDF4 files

Package	Function	Purpose
pbdNCDF4	`nc_create_par`	Create a NetCDF4 file in parallel
	`nc_open_par`	Open a NetCDF4 file in parallel
	`nc_var_par_access`	Specify parallel variable
ncdf4	`nc_create`	Create a NetCDF4 file
	`nc_open`	Open a NetCDF4 file
pbdNCDF4 & **ncdf4**	`ncdim_def`	Define data dimension
	`ncvar_def`	Define a variable
	`ncvar_put`	Write data to a NetCDF4 file
	`ncvar_get`	Read data from a NetCDF4 file
	`nc_close`	Close a NetCDF4 file

The data `TREFHT` as shown in the Figure 10.1 is taken from monthly averaged temperature at reference height of January 2004. This dataset is about three megabytes and is a tiny part of ultra-large simulations conducted by Mr. Prabhat and Michael Wehner of Lawrence Berkeley National Laboratory. The simulations run from 1987 to 2005 over 1152 longitudes (lon), 768 latitudes (lat), and 30 altitudes (lev). The total amount of simulation outputs is over 200 Terabytes, which are summarized and averaged including monthly-averaged, daily-averaged, and three-hours-averaged data. More datasets are available on ESGF (`http://www.earthsystemgrid.org/`) through the C20C project (on the NERSC portal).

A user with **pbdDEMO** installed can load the `TREFHT` dataset in the usual way, namely `data(TREFHT)` after loading the **pbdDEMO** package. Here, `TREFHT` is a list consisting of several elements. First, `TREFHT$def` contains all definitions regarding to this variable in class `ncvar4` including locations, dimensions, units, variable size, data storage, missing values, etc.

Next, `TREFHT$def$size` gives the data dimensions which are $(lon, lat, time) = (1152, 768, 1)$. Since this data is monthly averaged of Jan. 2004, it is stored as an one-time step output which is an averaged slice among 20 years.

Finally, `TREFHT$data` contains the values of each location and is a matrix with dimension 1152×768. Note that the column (lon) is in x-axis direction and the row (lat) is in y-axis direction.

Example: Temperature at reference height (TREFHT).

In an R session (interactive mode), run the demo by executing

R Code

```
demo(trefht, 'pbdDEMO', ask = F, echo = F)
```

This will show a plot as the Figure 10.1 providing a visualization about this variable and how temperatures are vary across locations, particularly decreasing in latitudes. Moreover, the South hemisphere is hoter than the North hemisphere since the seasonal effect.

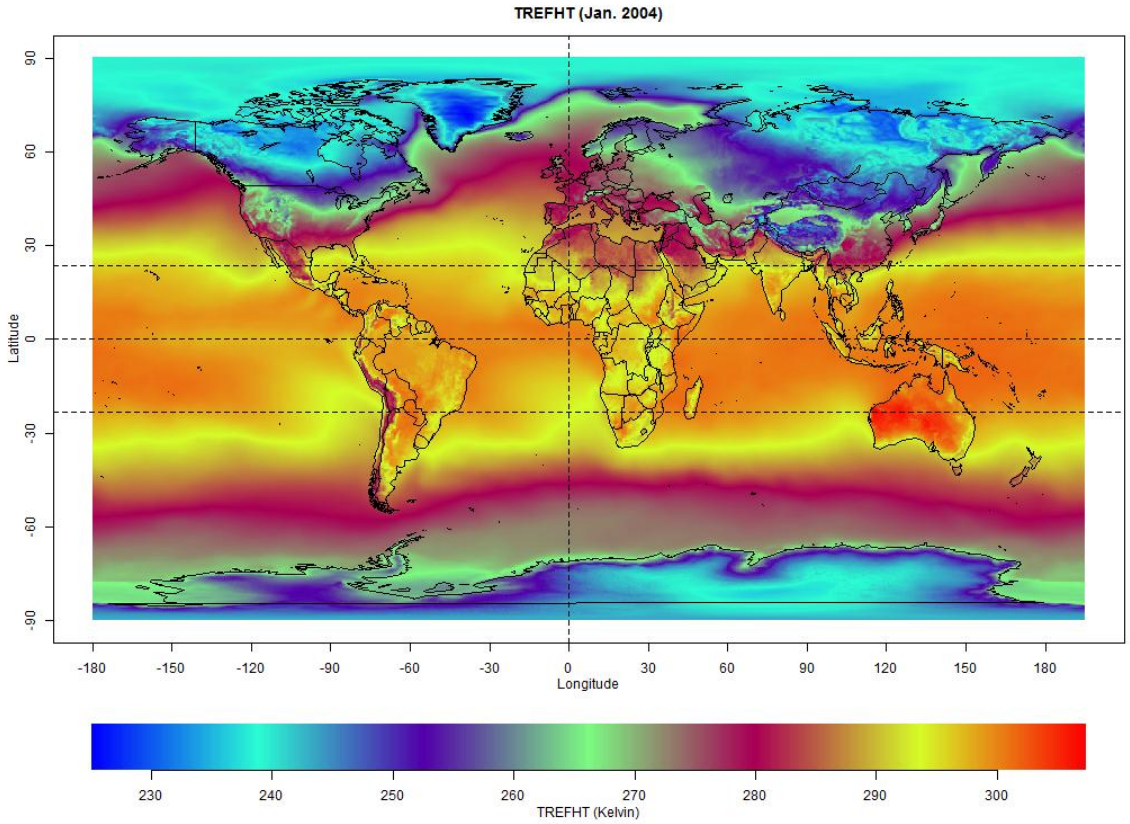


Figure 10.1: Monthly averaged temperature at reference height (TREFHT) in Kelvin (K) for the January 2004. Water freezes at 273.15K and boils at 373.15K.

10.2 Parallel Write and Read

Example: Dump a ddmatrix to a NetCDF4 file and load them from disk.

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(nc4_dmat,'pbdDEMO',ask=F,echo=F)"
```

Main part of the demo is given in the next:

nc4_dmat

```
1 ### divide data into ddmatrix
2 x <- TREFHT$data
3 dx <- as.ddmatrix(x)
4
5 # define dimension and variable
```

```
6  lon <- ncdim_def("lon", "degree_east", vals =
       TREFHT$def$dim[[1]]$vals)
7  lat <- ncdim_def("lat", "degree_north", vals =
       TREFHT$def$dim[[2]]$vals)
8  var.def <- ncvar_def("TREFHT", "K", list(lon = lon, lat = lat),
       NULL)
9
10 ### parallel write
11 file.name <- "nc4_dmat.nc"
12 nc <- nc_create_par(file.name, var.def)
13 ncvar_put_dmat(nc, "TREFHT", dx)
14 nc_close(nc)
15 if(comm.rank() == 0){
16   ncdump(file.name)
17 }
18
19 ### parallel read (everyone owns a portion)
20 nc <- nc_open_par(file.name)
21 if(comm.rank() == 0){
22   print(nc)
23 }
24 new.dx <- ncvar_get_dmat(nc, "TREFHT", bldim = bldim(dx), ICTXT
       = ctxt(dx))
25 nc_close(nc)
```

Line 2 and 3 convert `TREFHT$data` into a `ddmatrix` distributed across 4 processors. Line 6 and 7 define the dimensions `lon` and `lat` for longitudes and latitudes, and line 8 defines `var.def` as the dumping variable for "TREFHT" according to the dimensions. Line 12, 13, and 14 create a parallel NetCDF4 file `nc4_dmat.nc`, write the data into the variable on the disk, and close the file. Line 20, 24, and 25 open the file again and read the data from the variable from the data and convert them to a `ddmatrix`.

Note that `ncvar_put_dmat()` and `ncvar_get_dmat()` are implemented for 2D variables only. Please use **pbdNCDF4**/**ncdf4** primitive functions `ncvar_put()` and `ncvar_get()` via arguments `start` and `count` for more complicated cases. For example, we may write the `TREFHT` into a slice of a hypercube according to it's time step (Jan. 2004).

10.3 Exercises

10-1 The demo code `demo/nc4_serial.r` of **pbdDEMO** has a serial version of writing and reading `TREHFT` as using **ncdf4** on a single NetCDF4 file `nc4_serial.nc`. It is in the sense of single processor programming and has low cost if file is not too big. It is tedious but possible for multiple processors to write a single file with carefully manual barrier and synchronization. Modify `demo/nc4_serial.r` for writing with multiple processors.

10-2 It is also possible to read whole chunk of data from a single processor and distribute data later manually. Modify the demo code `demo/nc4_parallel.r` to accomplish this goal and make performance comparisons.

10-3 Implement functions or add arguments to `ncvar_put_dmat()` and `ncvar_get_dmat()` to enable writing and reading high dimension data, for example, $(lon, lat, time)$ is 2D in time (3D cube) or $(lon, lat, lev, time)$ is 3D in time (4D hypercube). Dump `TREFHT` to a slice of 3D cube and load them back to a `ddmatrix`.

10-4 In the Sections 11.3 and 11.4, we introduce simple matrix distributed formats `spmdr` and `spmdc` similar to the BLACS contexts `ICTXT` 2 and 1 with very large block size. The demo code `demo/nc4_spmdc.r` implements similar functionality as for `ddmatrix`, but for `spmdc` format only. Modify the demo code for `spmdr` format. Hint: See the Exercise 11-4.

11

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. Throughout this chapter, we will be making reference to several particulars to the block-cyclic data type used for objects of class `ddmatrix`. In particular, a working knowledge of the block-cyclic data structure and their relationship with BLACS contexts is most useful for the content to follow. As such, the reader is *strongly* encouraged to be familiar with the content of the **pbdDMAT** vignette before proceeding.

11.1 Distributed Matrix Redistributions

Example: Convert between different distributed matrix distributions.

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(reblock,'pbdDEMO',ask=F,echo=F)"
```

The distributed matrix class `ddmatrix` has two components which can be specified, and modified, by the user to drastically affect the composition of the distributed matrix. In particular, these are the object's block-cyclic blocking factor `bldim`, and the BLACS communicator number `CTXT` which sets the 2-dimensional processor grid.

Thankfully, redistributing is a fairly simple process; though we would emphasize that **this is not free of cost**. Reshaping data, especially at scale, can be much more expensive in total than even computation time. That said, sometimes data must move. It is better to get the job done slowly than to "take your ball and go home" with no results. But we caution that if redistribution can be avoided, then it should, at all costs.

The demo relies on a utility from the **pbdBASE** package, namely `redistribute()`. As the name implies, this method is for "reshaping" a block-cyclically distributed matrix of one kind to

another. Specifically, this takes an object of class `ddmatrix` as both an input and an output; i.e., and to emphasize the title of the chapter, this is not a method of *distribution* but *redistribution*.

For example, if I have a distributed matrix `dx` and I wish to reshape the distributed matrix so that it now has blocking dimension `newbldim` and is distributed across BLACS context `newCTXT`, then I need merely call:

```
dy <- redistribute(dx, bldim=newbldim, ICTXT=newCTXT)
```

Assuming the data is block cyclic of *any* kind, including degenerate cases, we can convert it to a block cyclic format of any other kind we wish via this `redistribute()` function. The only requirement is that the two different distributions have at least 1 processor in common, and so using the default BLACS contexts (0, 1, and 2) is always acceptable.

11.2 Implicit Redistributions

There are several useful functions which apply to distributed matrices, but require a data redistribution as in Section 11, whether the user realizes it or not. These functions are listed in

Function	Example	Package	Effect
`'['`	`dx[, -1]`	**pbdBASE**	Row/Column extraction and subsetting
`na.exclude()`	`na.exclude(dx)`	**pbdBASE**	Drop rows with `NA`'s
`apply()`	`apply(dx, 2, sd)`	**pbdDMAT**	Applies function to margin

Table 11.1: Distributed Matrix Methods with Implicit Data Redistributions

Table 11.1. By default, these functions will re-distribute back to the original data distribution after having performed the initial (necessary) redistribution and performed the requested operations. That is, by default, the problem of managing different data distributions is hidden from the user and entirely implicit. However, there are advantages to becoming familiar with managing these data distributions, because each of these functions has the option to have redistribution directly managed. Now, a data redistribution must occur to use these functions, but understanding which and why can help minimize the number of redistributions performed.

Many of the full details, such as *why* the redistributions need occur in the first place, are outlined in the **pbdDMAT** vignette, but we provide a simple example here. Suppose we have a distributed matrix `dx` distributed on the default grid (i.e., BLACS context 0) and we wish to drop the first column and then use the `apply()` function to extract the p-values, column-wise, of the result of running the Shapiro-Wilk normality test independently on the columns. No one is claiming that this is a wise thing to do, but it is useful for the purpose of demonstration.

To achieve this, we could execute the following:

Implicit Redistributions

```
dx <- dx[-1, ]
```

```
2
3 result <- apply(dx, MARGIN=2, FUN=function(col)
     shapiro.test(col)$p, reduce=TRUE)
```

In reality, underneath this is actually performing the following sequence of operations:

Implicit Redistributions

```
1 dx <- redistribute(dx, ICTXT=2)
2 dx <- dx[, -1]
3 dx <- redistribute(dx, ICTXT=0)
4
5 dx <- redistribute(dx, ICTXT=2)
6 result <- apply(dx, MARGIN=2, FUN=function(col)
     shapiro.test(col)$p, reduce=TRUE)
```

Or suppose we wanted instead to drop the first column; then this is equivalent to

Implicit Redistributions

```
1 dx <- redistribute(dx, ICTXT=1)
2 dx <- dx[, -1]
3 dx <- redistribute(dx, ICTXT=0)
4
5 dx <- redistribute(dx, ICTXT=2)
6 result <- apply(dx, MARGIN=2, FUN=function(col)
     shapiro.test(col)$p, reduce=TRUE)
```

The problem should be obvious. However, thoroughly understanding the problem, we can easily manage the data redistributions using the `ICTXT=` option in these function. So for example, we can minimize the redistributions to only the minimal necessary amount with the following:

Implicit Redistributions

```
1 dx <- dx[, -1, ICTXT=2]
2
3 result <- apply(dx, MARGIN=2, FUN=function(col)
     shapiro.test(col)$p, reduce=TRUE)
```

This is equivalent to explicitly calling:

Implicit Redistributions

```
1 dx <- redistribute(dx, ICTXT=2)
2 dx <- dx[, -1, ICTXT=2]
3
4 result <- apply(dx, MARGIN=2, FUN=function(col)
     shapiro.test(col)$p, reduce=TRUE)
```

This is clearly preferred. For more details, see the relevant function documentation.

11.3 Load Balance and Unload Balance

Example: Load balancing (and unbalancing) distributed 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(balance,'pbdDEMO',ask=F,echo=F)"
```

Suppose we have an unbalanced, distributed input matrix `X.spmd`. We can call `balance.info()` on this object to store some information about how to balance the data load across all processors. This can be useful for tracking data movement, as well as for "unbalancing" later, if we so choose. Next, we call `load.balance()` to obtain a load-balanced object `new.X.spmd`. We can also now undo this entire process and get back to `X.spmd` by calling `unload.balance()` on `new.X.spmd`.

All together, the code looks something like:

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 of this exchange are depicted in the example in Figure 11.1. Here, `X.spmd` is unbalanced, and `new.X.spmd` is a balanced version of `X.spmd`.

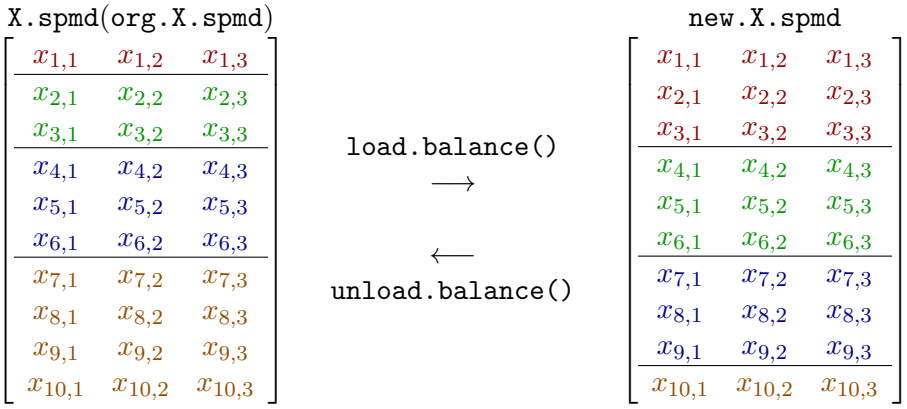


Figure 11.1: Load Balancing/Unbalancing Data: $\boldsymbol{X}$ is distributed in `X.spmd`(`org.X.spmd`) and `new.X.spmd`. Both are distributed row-wise in 4 processors. The colors represent processors 0, 1, 2, and 3, respectively.

The function `balance.info()` is extremely useful, because it will return the information used to load balance the given data `X.spmd`. The return of `balance.info()` is a list consisting of two data frames, `send()` and `recv()`, as well as two vectors, `N.allspmd` and `new.N.allspmd`.

Here, `send` records the original processor rank and the destination processor rank of the unbalanced data (that which is to be transmitted by that processor). The `load.balance()` function

uses this table to move the data via **pbdMPI**'s `isend()` function. If any "destination rank" is not the "original rank", then the corresponding data row will be moved to the appropriate processor. On the other hand, `recv` records the original processor rank and the destination rank of balanced data (that which is received by that processor).

The `N.allspmd` and `new.N.allspmd` objects both have length equal to the communicator containing all numbers of rows of `X.spmd` before and after the balancing, respectively. This is for double checking and avoiding a 0-row matrix issue.

For `unload.balance`, the process amounts to reversing `bal.info` and passing it to `load.balance`.

Finally, note that the "balanced" data is chosen to be balanced in a very particular way; it is arguably not "balanced", since 3 processors own 3 rows while 1 owns 1 row, and it is perhaps more balanced to have 2 processors own 3 rows and 2 own 2. However, we make this choice for the reason that our "balanced" data will always be a certain kind of degenerate block-cyclic structure. We will discuss this at length in the following section.

11.4 Convert Between SPMD and DMAT

Example: Convert between SPMD and DMAT formats.

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)"
```

The final redistribution challenge we will present is taking an object in SPMD format and putting it in the DMAT format. More precisely, we assume the input object `X.spmd` is in SPMD and convert the object into an object of class `ddmatrix` which we will call `X.dmat`.

The Figure 11.2 illustrates an example `X.spmd` and `X.dmat` conversion. For full details about the block-cyclic data format used for class `ddmatrix`, see the **pbdDMAT** vignette.

To perform such a redistribution, one simply needs to call:

R Code

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

or

R Code

```
X.spmd <- dmat2spmd(X.dmat)
```

Here, the `spmd2dmat` function does the following:

1. Check number of columns of `X.spmd`. All processors should be the same.

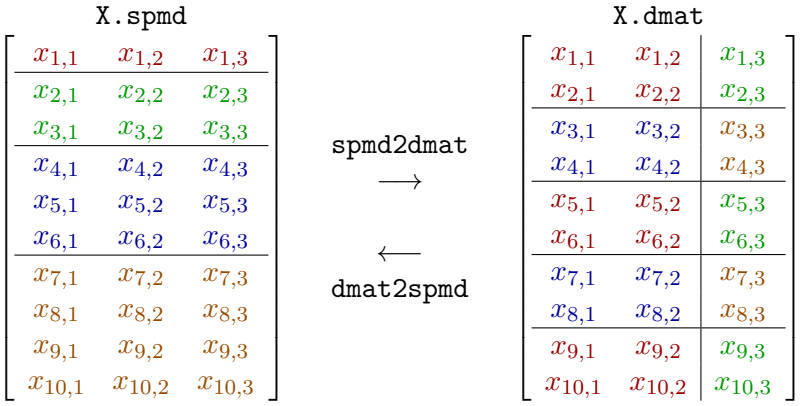


Figure 11.2: Converting Between SPMD and DMAT: $\boldsymbol{X}$ is distributed in `X.spmd` and `X.dmat`. Both are distributed in 4 processors where colors represents processor 0, 1, 2, and 3. Note that `X.dmat` is in block-cyclic format of 2×2 grid with 2×2 block dimension.

2. Row balance the SPMD matrix as necessary via `load.balance()` as in Section 11.3.
3. Call construct a new ddmatrix object (via the `new()` constructor) on the balanced matrix, say `X.dmat`, in BLACS context 2 (`ICTXT = 2`).
4. Redistribute `X.dmat` to another BLACS context as needed (default `ICTXT = 0`) via the `base.reblock()` function as in Section 11.1.

Note that the `load.balance()` function, as used above, is legitimately necessary here. Indeed, this function takes a collection of distributed data and converts it into a degenerate block cyclic distribution; namely, this places the data in block "1-cycle" format, distributed across an $n \times 1$ processor grid. In the context of Figure 11.2 (where the aforementioned process is implicit), this is akin to first moving the data into a distributed matrix format with `bldim=c(3,3)` and `CTXT=2`. Finally, we can take this degenerate block-cyclic distribution and again to Figure 11.2 as our motivating example, we convert the data balanced data so that it has `bldim=c(2,2)` and `CTXT=0`.

11.5 Exercises

11-1 In the Sections 11.3 and 11.4, we have seen the load balance of SPMD matrix and the conversion between SPMD and DMAT where SPMD matrices `X.spmd` are presumed in row-major as shown in the Figures 11.1 and 11.2. Create new functions `spmdr2spmdc()` and `spmdc2spmdr()` converting between row-major and column-major by utilizing functions `spmd2dmat()` and `dmat2spmd()` and changing their option `spmd.major`.

11-2 The demo code `demo/spmd_dmat.r` of **pbdDEMO** has a SPMD row-major matrix `X.spmd`. Utilize the functions developed in the Exercise 11-1. Convert `X.spmd` to a column-major matrix `new.X.spmdc` by calling `spmdr2spmdc()`, then convert `new.X.spmdc` back to a row-major matrix `new.X.spmdr` by calling `spmdc2spmdr()`. Check if `new.X.spmdr` were the same as `X.spmd`.

11-3 In **pbdDEMO**, there are some internal functions `demo.spmdr2dmat()`, `demo.spmdc2dmat()`, `demo.dmat2spmdr()`, and `demo.dmat2spmdc()` which have similar implementations as the functions `spmdr2spmdc()` and `spmdc2spmdr()` of the Exercise 11-1. Utilize these functions as templates. Create a function `spmd2spmd()` with an argument `new.major` (1, 2) for designated row- or column-majors. Return warnings or errors if the input matrix is not convertible.

11-4 The demo code `demo/nc4_spmdc.r` of **pbdDEMO** is an example utilizing SPMD column-major matrix `X.spmdc` and dumps the matrix into a NetCDF4 file. Adjust the code. Create a SPMD row-major matrix `X.spmdr` and dump the matrix to a new NetCDF4 file `nc4_spmdr.nc` by utilizing the function `ncvar_put_spmd()` with option `spmd.major = 1`. Verify that all `TREFHT` values of both `nc4_spmdc.nc` and `nc4_spmdr.nc` are identical. Hint: The local matrix of a SPMD row- or column-major matrix is still row-major as the default of R.

11-5 The `load.balance()` and `unload.balance()` have a potential bug when data size is small and can not fit into the desired block size of a block-cyclic matrix. For instance, four processes in a SPMD row-major format with a matrix 5×1. The two functions will (un-) balance the data in 2×1 in process 0, and 1×1 in others. If the desired block size is 2, then the data should be 2×1 in processes 0 and 1, 1×1 in process 2, and no element for processor 3. Does any way to fix these two functions?

Part V

Applications

12

Model-Based Clustering

12.1 Introduction

Model-based clustering is an unsupervised learning technique and mainly based on finite mixture models to fit the data, cluster the data, and draw inference from the data (Fraley and Raftery, 2002; Melnykov and Maitra, 2010). The major application of model-based clustering focuses on Gaussian mixture models. For example, $\boldsymbol{X}_n$ is a random p-dimensional observation from the Gaussian mixture model with K components, which has density

$$f(\boldsymbol{X}_n; \boldsymbol{\Theta}) = \sum_{k=1}^{K} \eta_k \phi_p(\boldsymbol{X}_n; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{12.1}$$

where $\phi_p(\cdot;\cdot,\cdot)$ is a p-dimensional Gaussian density,

$$\boldsymbol{\Theta} = \{\eta_1, \eta_2, \ldots, \eta_{K-1}, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_K, \boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2, \ldots, \boldsymbol{\Sigma}_K\},$$

is the parameter space, η_k's are mixing proportion, $\boldsymbol{\mu}_k$'s are the centers of the components, and $\boldsymbol{\Sigma}_k$'s are the dispersion of the components.

Suppose a data set $\boldsymbol{X} = \{\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_N\}$ has N observations. Then the log likelihood is

$$\log L(\boldsymbol{\Theta}; \boldsymbol{X}) = \sum_{n=1}^{N} \log f(\boldsymbol{X}_n; \boldsymbol{\Theta}) \tag{12.2}$$

where f is as in Equation (12.1). Solving the problem of maximizing this log-likelihood is usually done by the expectation-maximization (EM) algorithm (Dempster *et al.*, 1977). Assuming the EM algorithm converges, we let $\hat{\boldsymbol{\Theta}}$ be the maximum likelihood estimator of Equation (12.2). Then the maximum posterior probability

$$\underset{k}{\operatorname{argmax}} \frac{\hat{\eta}_k \phi_p(\boldsymbol{X}_n; \hat{\boldsymbol{\mu}}_k, \hat{\boldsymbol{\Sigma}}_k)}{f(\boldsymbol{X}_n; \hat{\boldsymbol{\Theta}})}$$

for $n = 1, 2, \ldots, N$ indicates the membership of the observations of the data set $\boldsymbol{X}$.

The **mclust** (Fraley *et al.*, 1999) and **EMCluster** (Chen *et al.*, 2012a) packages are the two main R packages implementing the EM algorithm for the model-based clustering. The **mclust** package has several selections on different kinds of models one may fit, while **EMCluster** implements the most complicated model (dispersions are all unstructured) in a more efficient way, using several initializations, and semi-supervised learning. However, both assume small N and tiny p, and only run in serial with sufficient memory.

Note that the k-means algorithm (Forgy, 1965) equivalently assumes $\eta_1 = \eta_2 = \cdots = \eta_K \equiv 1/K$ and $\mathbf{\Sigma}_1 = \mathbf{\Sigma}_2 = \cdots = \mathbf{\Sigma}_K \equiv \boldsymbol{I}$ in Equation (12.1), where $\boldsymbol{I}$ is the identity matrix. As such, the k-means algorithm is a restricted Gaussian mixture model, such that it can be implemented with a simplified version of the EM algorithm. However, due to its strict assumptions, the cluster results are almost always unrealistic, leaving the data scientist unable to draw meaningful inference from the data, and sometimes have unreasonably high classification errors.

12.1.1 Parallel Model-Based Clustering

The **pmclust** (Chen and Ostrouchov, 2012) package is an R package for parallel model-based clustering based on Gaussian mixture models with unstructured dispersions. The package uses data parallelism to solve one very large clustering problem, rather than the embarrassingly parallel problem of fitting many independent models to dataset(s). This approach is especially useful for large, distributed platforms, where the data will be distributed across nodes. And of course it is worth nothing that the package does not merely perform a local clustering operation on the local data pieces; some "`gather`" and "`reduce`" operations are necessary at some stages of the parallel EM algorithm.

An expectation-gathering-maximization (EGM) algorithm (Chen *et al.*, 2013) is established for minimizing communication and data movement between nodes. There are four variants of EGM-like algorithms implemented in **pmclust** including EM, AECM (Meng and van Dyk, 1997) , APECM (Chen and Maitra, 2011) , and APECMa (Chen *et al.*, 2013) . The variants are trying to achieve better convergence rates and less computing time than the original EM algorithm. For completeness' sake, a simple k-means algorithm is also implemented in **pmclust**.

The **pmclust** package is the first pbdR application, and the first R package in SPMD to analyze distributed data in Gigabyte scale. It was originally designed for analyzing Climate simulation outputs (CAM5), as discussed in Section 10.1, and is a product for the project "Visual Data Exploration and Analysis of Ultra-large Climate Data" supported by U.S. DOE Office of Science.

The **pmclust** package initially depended on **Rmpi**, but designed in SPMD approach rather than in the manager/worker paradigm even before pbdR existed. Later, it migrated to use **pbdMPI** (Chen *et al.*, 2012b) because of performance issues with **Rmpi** on larger machines. So, by default, the package assumes data are stored in SPMD row-major matrix format.

Currently, the package also utilizes **pbdSLAP** (Chen *et al.*, 2012d), **pbdBASE** (Schmidt *et al.*, 2012a), and **pbdDMAT** (Schmidt *et al.*, 2012b) to implement a subset of the above algorithms for data in the `ddmatrix` format. Table 12.1 lists the current implementations.

Table 12.1: Parallel Mode-Based Clustering Algorithms in **pmclust**

Algorithm	SPMD	`ddmatrix`
EM	yes	yes
AECM	yes	no
APECM	yes	no
APECMa	yes	no
k-means	yes	yes

Based on **pmclust** version 0.1-4

12.2 An Example Using the *Iris* Dataset

The `iris` (Fisher, 1936) dataset is a famous dataset available in R consisting of 50 Iris flowers from each of three species of Iris, namely *Iris setosa*, *Iris versicolor*, and *Iris virginica*. The dataset is tiny, even by today's standards, with only 150 rows and five columns. The column variables consist of the four features sepal length, sepal width, petal length, and petal width, as well as the class of species. We take the first four columns of `iris` to form the matrix $\boldsymbol{X}$, where each row can be classified in three groups by the true id (the fifth column of `iris`) for supervised learning, or clustered in three groups by algorithms for unsupervised learning. Note that the dimension of $\boldsymbol{X}$ is $N = 150$ by $p = 4$.

Figure 12.1 shows the pair-wised scatter plot for all features denoted on the diagonal, and classes are indicated by colors. Each panel plots two features on x and y axes. It is clear that `Petal.Length` can split three species in two groups. However, one of the group is mixed with two species and can not be distinguished by any one of these four features.

From the supervised learning point view, the empirical estimation for $\boldsymbol{\Theta}$ from data will be the best description for the data, assuming the "true model" is a Gaussian mixture. The (serial) demo code `iris_overlap` in **pbdDEMO** quickly suggests the overlap level of three Iris species. It can be obtained by executing:

R Code

```
R> demo(iris_overlap, 'pbdDEMO', ask = F, echo = F)
```

which utilizes the `overlap` function of **MixSim** (Melnykov *et al.*, 2012). The output is:

R Output

```
R> (ret <- overlap(ETA, MU, S))
$OmegaMap
              [,1]          [,2]        [,3]
[1,] 1.000000e+00 7.201413e-08 0.00000000
[2,] 1.158418e-07 1.000000e+00 0.02302315
[3,] 0.000000e+00 2.629446e-02 1.00000000

$BarOmega
[1] 0.01643926

$MaxOmega
```

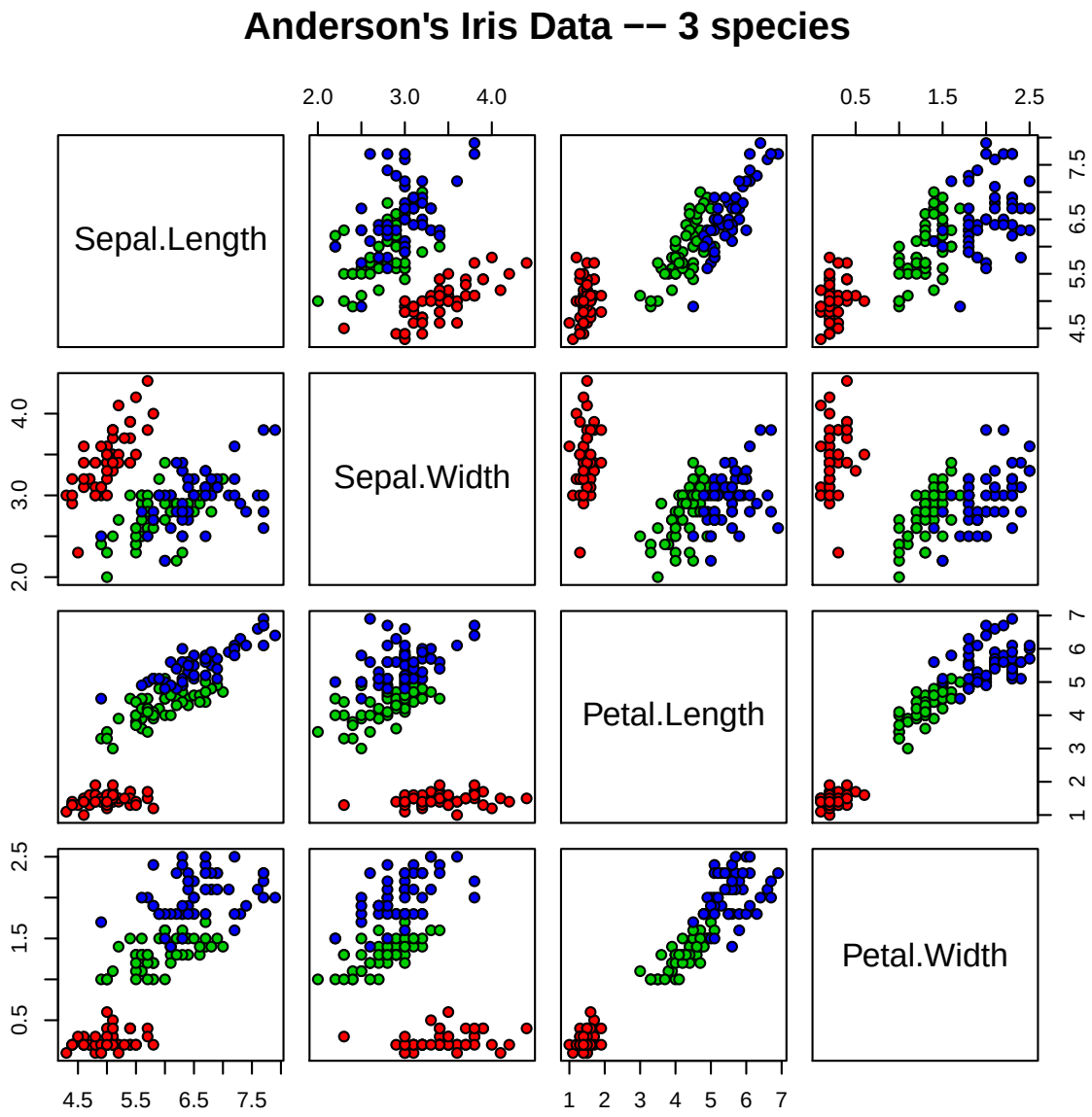


Figure 12.1: Iris pair-wised scatter plot. *Iris setosa* is in red, *Iris versicolor* is in green, and *Iris virginica* is in blue.

```
[1] 0.0493176

$rcMax
[1] 2 3

R> (levels(iris[, 5]))
[1] "setosa"     "versicolor" "virginica"
```

The `OmegaMap` matrix is a map of pair-wise overlap of three species where rows/columns 1, 2, and 3 are *Iris setosa*, *Iris versicolor*, and *Iris virginica*, respectively. The outputs also indicate

that the averaged pair-wised overlap (`BarOmega`) is about 1.6%, and the maximum pair-wised overlap (`MaxOmega`) is about 4.9% among these three Iris species. Also, the maximum occurs at 2 (*Iris versicolor*) and 3 (*Iris virginica*) indicating these two species are partly inseparable given these four features.

From the unsupervised learning point view, such as model-based clustering, we must pretend that we are blind to the true class ids, or said another way, we must treat the fifth column of $\boldsymbol{X}$ as unobserved. We can then use the four features to form the model and cluster the data, then go back and compare our unsupervised learning results to the true values.

Note that *Iris versicolor* and *Iris virginica* are partly inseparable, so misclassification can happen at the overlap region. We validate the results by comparing the clustering ids to the true class ids using adjusted Rand index (ARI) (Hubert and Arabie, 1985) . The ARI takes values between -1 and 1, where 1 is a perfect match. The function `RRand()` in **MixSim** also provides the ARI.

The analysis in the unsupervised learning approach proceeds as follows:

1. decompose $\boldsymbol{X}$ on its principal components,
2. project $\boldsymbol{X}$ onto the first two principal components (those with largest variability),
3. fit a k-means model and a model-based clustering model, and finally
4. visualize $\boldsymbol{X}$ on the plane formed by these new axes, labeling the entries of $\boldsymbol{X}$ on this plot with the true ids, and the estimated ids from the clustering algorithms.

This will be the general procedure whether in serial or parallel. For example's sake, we will extend these steps to offer SPMD code and `ddmatrix` code to show the similarity of codes.

This example demonstrates that the **pmclust** package can perform analysis correctly, but is not meant to be a demonstration of its scalability prowess. The `iris` dataset is, by any reasonable definition, tiny. Small datasets are generally not worth the trouble of developing parallelized analysis codes for, especially since all the extra overhead costs inherent to parallelism might dominate any theoretical performance gains. Again, the merit of the example is to show off the syntax and accuracy on a single machine; however, **pmclust** scales up nicely to very large dataset running on supercomputers.

12.2.1 *Iris* in Serial Code and Sample Outputs

The demo code for the serial Iris example can be found with the package demos, and executed via:

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(iris_serial, 'pbdDEMO', ask=F, echo=F)"
```

The code is fairly self-explanatory, and well-commented besides, so we will leave it as an exercise to the reader to read through it carefully.

Running this demo should produce an output that looks something like the following:

```
 Sepal.Length   Sepal.Width  Petal.Length   Petal.Width
-4.480675e-16  2.035409e-16 -2.844947e-17 -3.714621e-17
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000  -0.1175698    0.8717538   0.8179411
Sepal.Width    -0.1175698   1.0000000   -0.4284401  -0.3661259
Petal.Length    0.8717538  -0.4284401    1.0000000   0.9628654
Petal.Width     0.8179411  -0.3661259    0.9628654   1.0000000
K-means clustering with 3 clusters of sizes 50, 47, 53

Cluster means:
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1  -1.01119138  0.85041372   -1.3006301  -1.2507035
2   1.13217737  0.08812645    0.9928284   1.0141287
3  -0.05005221 -0.88042696    0.3465767   0.2805873

Clustering vector:
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     1 1 1 1
 [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 3 3 3 2 3 3 3 3 3 3 3 2 3 3 3 3
    2 3 3 3
 [75] 3 2 2 2 3 3 3 3 3 3 3 2 2 3 3 3 3 3 3 3 3 3 3 3 3 2 3 2 2 2 2 3
    2 2 2 2
[112] 2 2 3 3 2 2 2 2 3 2 3 2 3 2 2 3 2 2 2 2 2 2 3 3 2 2 2 3 2 2 2 3 2
   2 2 3 2
[149] 2 3

Within cluster sum of squares by cluster:
[1] 47.35062 47.45019 44.08754
 (between_SS / total_SS =  76.7 %)

Available components:

[1] "cluster"      "centers"      "totss"        "withinss"
   "tot.withinss"
[6] "betweenss"    "size"
Loading required package: MASS
Method: em.EMRnd.EM
 n = 150, p = 4, nclass = 3, flag = 0, logL = -288.5244.
nc:
[1] 50 55 45
pi:
[1] 0.3333 0.3673 0.2994
null device
          1
```

Finally, Figure 12.2 shows the visualization created by this script.

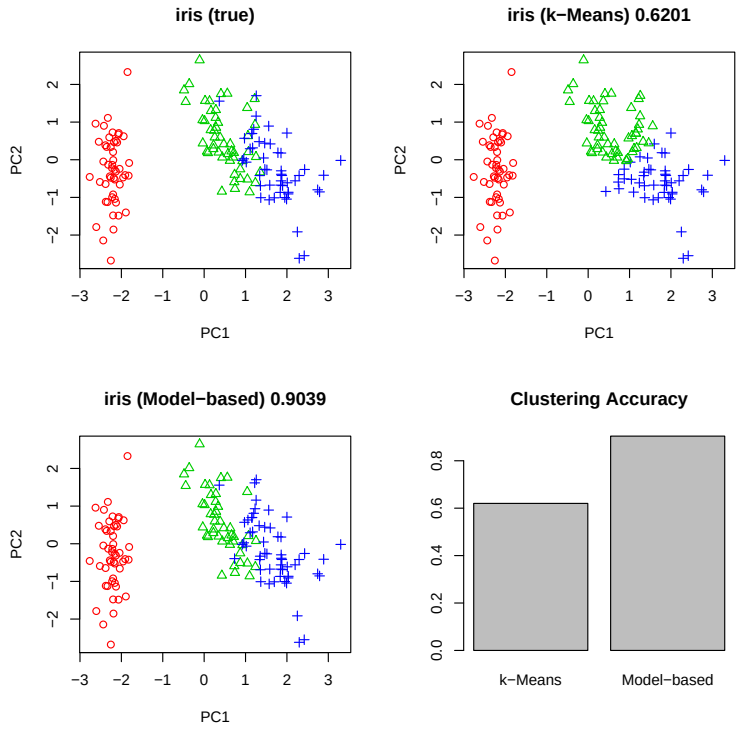


Figure 12.2: Iris Clustering Plots — Serial

12.2.2 *Iris* in SPMD Code

The demo code for the SPMD Iris example can be found with the package demos, and executed via:

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(iris_spmd, 'pbdDEMO', ask=F, echo=F)"
```

Sample Outputs

Running this script should produce an output that looks something like the following:

```
COMM.RANK = 0
[1] 2.547376e-14 8.076873e-15 4.440892e-14
COMM.RANK = 0
[1] 0.6201352 0.6311581 0.6928082
null device
          1
```

Finally, figure 12.4 shows the visualization created by this script.

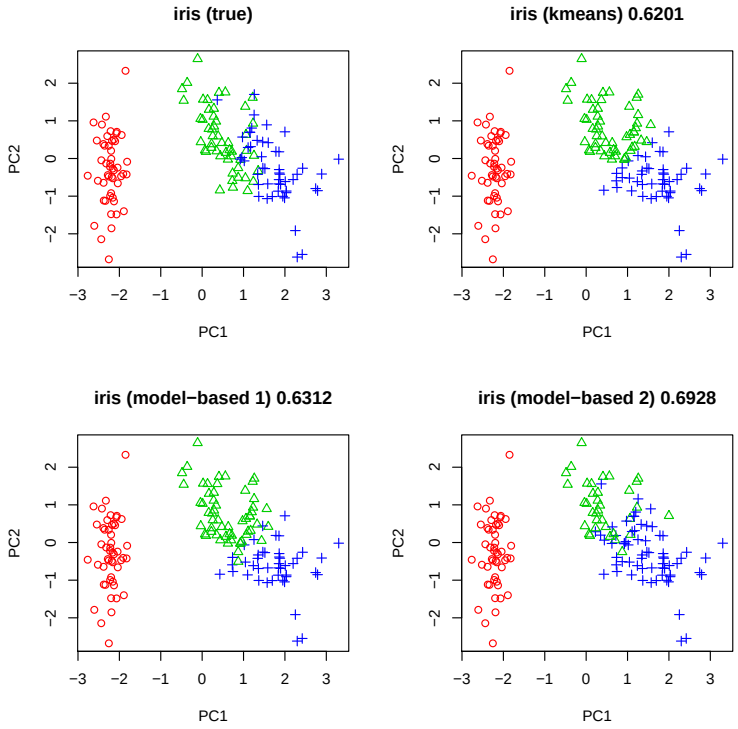


Figure 12.3: Iris Clustering Plots — SPMD

12.2.3 *Iris* in `ddmatrix` Code

The demo code for the DMAT Iris example can be found with the package demos, and executed via:

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(iris_dmat, 'pbdDEMO', ask=F, echo=F)"
```

Sample Outputs

Running this script should produce an output that looks something like the following:

```
Using 2x2 for the default grid size

COMM.RANK = 0
```

```
               [,1]          [,2]          [,3]          [,4]
[1,] -4.440892e-16 1.990595e-16 -2.428613e-17 2.498002e-16
COMM.RANK = 0
           [,1]       [,2]       [,3]       [,4]
[1,]  1.0000000 -0.1175698  0.8717538  0.8179411
[2,] -0.1175698  1.0000000 -0.4284401 -0.3661259
[3,]  0.8717538 -0.4284401  1.0000000  0.9628654
[4,]  0.8179411 -0.3661259  0.9628654  1.0000000
COMM.RANK = 0
[1] 0.645147

null device
          1
```

Finally, figure 12.4 shows the visualization created by this script.

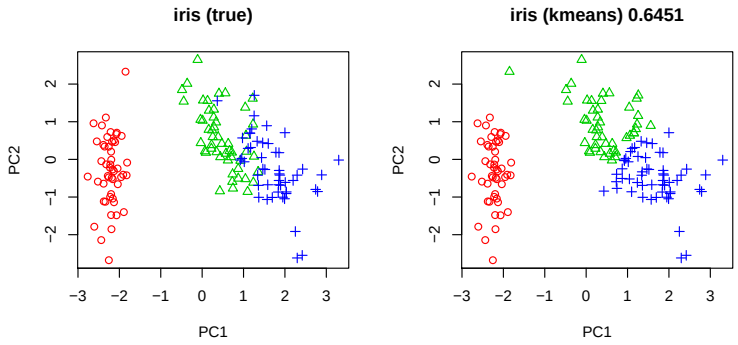


Figure 12.4: Iris Clustering Plots — SPMD

12.3 Exercises

12-1 As Figures 12.2 and 12.4, none of clustering method is able to obtain the true. However, there are ways that may improve the final clustering and close to the true, including

1) by reducing the convergent criteria,

2) by increasing the number of initialization steps,

3) by aggregating several initialization strategies, and

4) by given some prior information about classification.

Using `iris` as a data, and trying different ways to see if final clustering results are improved. See the next Exercises for details.

12-2 In serial, utilizing **MixSim** to generate parameters with different levels of overlaps, based on the parameters to generated data from Gaussian mixture models, and repeat Exercise 12-1 on the generated data to show how overlaps can affect algorithm performances by comparing ARIs.

12-3 In serial, utilizing **EMCluster** on the generated data from Exercise 12-2 to show and test how initialization strategies can affect algorithm performances by comparing ARIs.

12-4 In serial, **EMCluster** also implements semi-supervised model-based clustering , select some high density points from the generated data and labeling them as prior know information, then test how these information can affect algorithm performances by comparing ARIs.

Part VI

Appendix

A

Numerical Linear Algebra and Linear Least Squares Problems

For the remainder, assume that all matrices are real-valued.

Let us revisit the problem of solving linear least squares problems, introduced in Section 4.5. Recall that we wish to find a solution $\boldsymbol{\beta}$ such that

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

In the case that $\boldsymbol{X}$ is full rank (which is often assumed, whether reasonable or not), this has analytical solution

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

However, even with this nice closed form, implementing this efficiently on a computer is not entirely straightforward. Herein we discuss several of the issues in implementing the linear least squares solution efficiently. For simplicity, we will assume that $\boldsymbol{X}$ is full rank, although this is not necessary — although rank degeneracy does complicate things. For more details on the rank degeneracy problem, and linear least squares problems in general, see the classic *Matrix Computations* (Golub and Van Loan, 1996).

A.1 Forming the Normal Equations

If we wish to implement this numerically, then rather than directly computing the inverse of $\boldsymbol{X}^T\boldsymbol{X}$ directly, we would instead compute the Cholesky factorization

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

where L is lower triangular. Then turning to the so-called "normal equations"

$$\left(\boldsymbol{X}^T\boldsymbol{X}\right)\boldsymbol{\beta} = \boldsymbol{X}^T\boldsymbol{y} \tag{A.2}$$

by simple substitution and grouping, we have

$$L\left(L^T\boldsymbol{\beta}\right) = \boldsymbol{X}^T\boldsymbol{y}$$

Now, since L is triangular, these two triangular systems (one forward and one backward substitution found by careful grouping of terms above) can be solved in a numerically stable way (Higham, 2002). However, forming the Cholesky factorization itself suffers from the effects of roundoff error in having to form the product $\boldsymbol{X}^T\boldsymbol{X}$. We elaborate on this to a degree in the following section.

A.2 Using the QR Factorization

Directly computing the normal equations is ill advised, because it is often impossible to do so with adequate numerical precision. To fully appreciate this problem, we must entertain a brief discussion about condition numbers.

By definition, if a matrix has finite condition number, then it must have been invertible. However, for numerical methods, a condition number which is "big enough" is essentially infinite (loosely speaking). And observe that forming the product $\boldsymbol{X}^T\boldsymbol{X}$ squares the condition number of $\boldsymbol{X}$:

$$\begin{aligned}
\kappa\left(\boldsymbol{X}^T\boldsymbol{X}\right) &= \left|\left|\boldsymbol{X}^T\boldsymbol{X}\right|\right| \left|\left|\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\right|\right| \\
&= \left|\left|\boldsymbol{X}^T\boldsymbol{X}\right|\right| \left|\left|\boldsymbol{X}^{-1}\left(\boldsymbol{X}^T\right)^{-1}\right|\right| \\
&= \left|\left|\boldsymbol{X}^T\right|\right| ||\boldsymbol{X}|| \left|\left|\boldsymbol{X}^{-1}\right|\right| \left|\left|\boldsymbol{X}^{-T}\right|\right| \\
&= ||\boldsymbol{X}|| \, ||\boldsymbol{X}|| \left|\left|\boldsymbol{X}^{-1}\right|\right| \left|\left|\boldsymbol{X}^{-1}\right|\right| \\
&= ||\boldsymbol{X}||^2 \left|\left|\boldsymbol{X}^{-1}\right|\right|^2 \\
&= \kappa(\boldsymbol{X})^2
\end{aligned}$$

So if $\kappa\left(\boldsymbol{X}\right)$ is "large", then forming this product can lead to large numerical errors when attempting to numerically invert or factor a matrix, or solve a system of equations.

To avoid this problem, the orthogonal QR Decomposition is typically used. Here we take

$$\boldsymbol{X} = QR$$

where Q is orthogonal and R is upper trapezoidal (n the overdetermined case, R is triangular). This is beneficial, because orthogonal matrices are norm-preserving, i.e. Q is an isometry, and whence

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

This amounts to solving the triangular system

$$R\boldsymbol{\beta} = Q^T\boldsymbol{y}$$

As noted in Section A.1, solving this system can be done in a numerically stable fashion (and the high performance libraries employed by both R and pbdR use stable implementations). The key difference here is that the QR factorization is of $\boldsymbol{X}$, not $\boldsymbol{X}^T\boldsymbol{X}$, and so we need only worry about the conditioning of $\boldsymbol{X}$ (as opposed to its squared condition number).

While this method is much less prone to the numerical issues discussed above, it is much slower computationally. Additionally, we note that unlike the method in forming the normal equations, this method can be extended to the rank degenerate case.

A.3 Using the Singular Value Decomposition

There is another, arguably much more well-known matrix factorization which we can use to develop yet another analytically equivalent solution to the least squares problem, namely the Singular Value Decomposition (SVD). Using this factorization leads to a very elegant solution, as is so often the case with the SVD.

Note that in (A.1), the quantity

$$(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T$$

is the Moore-Penrose inverse of $\boldsymbol{X}$. So if we take

$$\boldsymbol{X} = U\Sigma V^T$$

to be the SVD of $\boldsymbol{X}$, then we have

$$\begin{aligned}
\boldsymbol{X}^+ &= \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T \\
&= \left(\left(U\Sigma V^T\right)^T\left(U\Sigma V^T\right)\right)^{-1}U\Sigma V^T \\
&= \left(V\Sigma^T\Sigma V^T\right)^{-1}V\Sigma^T U^T \\
&= V\left(\left(\Sigma^T\Sigma\right)^{-1}\Sigma^T\right)U^T \\
&= V\Sigma^+U^T
\end{aligned}$$

Whence,

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

Conceptually, this is arguably the most elegant method of solving the linear least squares problem. Additionally, as with the QR method above, with slight modification the above argument can extend to the rank degenerate case; however, we suspect that the SVD is much more well known to mathematicians and statisticians than is the QR decomposition. This abstraction comes at a great cost, though, as this approach is handily the most computationally intensive of the three presented here.

Linear Regression and Rank Degeneracy in R

In the case that $\boldsymbol{X}$ is rank deficient, then $\boldsymbol{X}$ (and whence $\boldsymbol{X}^T\boldsymbol{X}$) is not invertible, so the problem can not be solved by the method of Section A.1. Both R and pbdR use a QR factorization as in Section A.2, although the two systems use a slightly different approach. While most of the linear algebra in R is handled by LAPACK (Anderson *et al.*, 1999), arguably the most important numerical function in all of R, namely `lm.fit()` used by `lm()` to fit linear regression models, uses LINPACK (Dongarra *et al.*, 1979). By comparison to LAPACK, LINPACK is much less sophisticated. However, pbdR uses level 3 PBLAS and ScaLAPACK (the distributed equivalent of using level 3 BLAS and LAPACK) to fit linear regression models.

The LINPACK routines used by R are `DQRLS`, which calls a modified `DQRDC2` to compute a rank-revealing QR factorization with a "limited pivoting strategy" (more on this later). Finally, `DQRSL` is called to apply the output of the QR factorization to compute the least squares soluations. By contrast, pbdR uses a modified `PDGELS` routine, which uses a version of `PDGEQPF` modified to use R's "limited pivoting strategy", and then calls `PDORMQR` to fit the least squares solution.

Neither approach assumes that the model matrix is full rank. Instead, the methods are *rank-revealing*, in that they attempt to discover the numerical rank while computing the orthogonal factorization. However, both R and (for the sake of consistency) pbdR use a "limited pivoting strategy" (with language, we believe, due to Ross Ihaka) in determining numerical rank. Generally when computing a QR with pivoting, for the sake of numerical stability one chooses the column with largest partial norm while forming the Householder reflections. However, in doing so it is possible to permute the columns in such a way that a desired statistical interpretation (such as in an ANOVA) is destroyed. The solution employed by R is to merely iterate over the columns and choose the current column as the pivot each time. When a column is detected to have "small" partial norm, it is pushed to the back. The author of these modification writes:

> a limited column pivoting strategy based on the 2-norms of the reduced columns moves columns with near-zero norm to the right-hand edge of the x matrix. this strategy means that sequential one degree-of-freedom effects can be computed in a natural way.
>
> i am very nervous about modifying linpack code in this way. if you are a compu-

> tational linear algebra guru and you really understand how to solve this problem please feel free to suggest improvements to this code.

So in this way, if a model matrix is full rank, then the estimates coming from R should be considered at least as trustworthy as probably every other statistical software package of note. If it is not, then this method presents a possible numerical stability issue; although to what degree, if any at all, this is actually a problem, the authors at present have no real knowledge. If numerical precision is absolutely paramount, consider using the SVD to solve the least squares problem; though do be aware that this is hands down the slowest possible approach.

We again note that the limited pivoting strategy of R is employed by pbdR in the `lm.fit()` method for class `ddmatrix`.

Part VII

Miscellany

References

Adler D, Gläser C, Nenadic O, Oehlschlägel J, Zucchini W (2013). "ff: memory-efficient storage of large data on disk and fast access functions." R Package version 2.2-11, URL http://CRAN.R-project.org/package=ff.

Akaike H (1974). "A new look at the statistical model identification." *IEEE Transaction on Automatic Control*, **19**, 716–723.

Analytics R (2012). *foreach: Foreach looping construct for R*. R Package version 1.4.0, URL http://CRAN.R-project.org/package=foreach.

Anderson E, Bai Z, Bischof C, Blackford LS, Demmel J, Dongarra JJ, Du Croz J, Hammarling S, Greenbaum A, McKenney A, Sorensen D (1999). *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. ISBN 0-89871-447-8.

Benson F (1949). "A Note on the Estimation of Mean and Standard Deviation from Quantiles." *Journal of the Royal Statistical Society. Series B (Methodological)*, **11**(1), pp. 91–100.

Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1997). *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA. ISBN 0-89871-397-8 (paperback). URL http://netlib.org/scalapack/slug/scalapack_slug.html/.

Chen WC, Maitra R (2011). "Model-Based Clustering of Regression Time Series Data via APECM — an AECM Algorithm Sung to an Even Faster Beat." *Statistical Analysis and Data Mining*, **4**, 567–578.

Chen WC, Maitra R, Melnykov V (2012a). "EMCluster: EM Algorithm for Model-Based Clustering of Finite Mixture Gaussian Distribution." R Package, URL http://cran.r-project.org/package=EMCluster.

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 (2012). "pmclust: Parallel Model-Based Clustering." R Package, URL http://cran.r-project.org/package=pmclust.

Chen WC, Ostrouchov G, Pugmire D, Prabhat M, Wehner M (2013). "A Parallel EM Algorithm for Model-Based Clustering Applied to the Exploration of Large Spatio-Temporal Data." *Technometrics*. (accepted).

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

Chen WC, Schmidt D, Ostrouchov G, Patel P (2012d). "pbdSLAP: Programming with Big Data – Scalable Linear Algebra Packages." R Package, URL http://cran.r-project.org/package=pbdSLAP.

Dempster A, Laird N, Rubin D (1977). "Maximum Likelihood for Incomplete Data via the EM Algorithm (with discussion)." *Jounal of the Royal Statistical Society, Series B*, **39**, 1–38.

Dongarra JJ, Moler CB, Bunch JR, Stewart GW (1979). *LINPACK User's Guide*. SIAM.

Eaton B (2011). *User's Guide to the Community Atmosphere Model CAM-5.1*. NCAR. URL http://www.cesm.ucar.edu/models/cesm1.0/cam/.

Fisher R (1936). "The use of multiple measurements in taxonomic problems." *Annals of Eugenics*, **2**, 179–188.

Forgy E (1965). "Cluster analysis of multivariate data: efficiency vs. interpretability of classifications." *Biometrics*, **21**, 768–780.

Fraley C, Raftery A (2002). "Model-Based Clustering, Discriminant Analysis, and Density Estimation." *Journal of the American Statistical Association*, **97**, 611–631.

Fraley C, Raftery A, Scrucca L (1999). "mclust: Normal Mixture Modeling for Model-Based Clustering, Classification, and Density Estimation." R Package, URL http://cran.r-project.org/package=mclust.

Golub GH, Van Loan CF (1996). *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)(3rd Edition)*. 3rd edition. The Johns Hopkins University Press.

Gropp W, Lusk E, Skjellum A (1994). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA, USA: MIT Press Scientific And Engineering Computation Series.

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.

Higham NJ (2002). *Accuracy and Stability of Numerical Algorithms*. 2nd edition. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. ISBN 0898715210.

Hubert L, Arabie P (1985). "Comparing partitions." *Journal of Classification*, **2**, 193–218.

Kane M, Emerson J (2010). "The Bigmemory Project." URL http://www.bigmemory.org.

Kunen K (1980). *Set Theory: An Introduction to Independence Proofs*. North-Holland.

Melnykov V, Chen WC, Maitra R (2012). "MixSim: An R Package for Simulating Data to Study Performance of Clustering Algorithms." *Journal of Statistical Software*, **51**(12), 1–25. URL http://www.jstatsoft.org/v51/i12/.

Melnykov V, Maitra R (2010). "Finite Mixture Models and Model-Based Clustering." *Statistics Surveys*, **4**, 80–116.

Meng X, van Dyk D (1997). "The EM Algorithm — an Old Folk-song Sung to a Fast New Tune (with discussion)." *Journal of the Royal Statistical Society B*, **59**, 511–567.

NetCDF Group (2008). "Network Common Data Form." Software package, URL http://www.unidata.ucar.edu/software/netcdf/.

OpenMP ARB (1997). "OpenMP." URL http://www.openmp.org/.

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

Patel P, Ostrouchov G, Chen WC, Schmidt D, Pierce D (2013a). "pbdNCDF4: Programming with Big Data – Interface to Parallel Unidata NetCDF4 Format Data Files." R Package, URL http://cran.r-project.org/package=pbdNCDF4.

Patel P, Ostrouchov G, Chen WC, Schmidt D, Pierce D (2013b). "A Quick Guide for the pbdNCDF4 package." R Vignette, URL http://cran.r-project.org/package=pbdNCDF4.

Pierce D (2012). "ncdf4: Interface to Unidata netCDF (version 4 or earlier) format data files." R Package, URL http://CRAN.R-project.org/package=ncdf4.

R Core Team (2012a). "parallel: Support for Parallel Computation in R." R Package.

R Core Team (2012b). *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). "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 (2012c). *A Quick Guide for the pbdBASE package*. R Vignette, URL http://cran.r-project.org/package=pbdBASE.

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

Schmidt D, Chen WC, Ostrouchov G, Patel P (2013). "pbdDEMO: Programming with Big

Data – Demonstrations of pbd Packages." R Package, URL http://cran.r-project.org/package=pbdDEMO.

Schmidt D, Ostrouchov G, Chen WC, Patel P (2012e). "Tight Coupling of R and Distributed Linear Algebra for High-Level Programming with Big Data." In P Kellenberger (ed.), *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE Computer Society.

Sevcikova H, Rossini T (2012). *rlecuyer: R interface to RNG with multiple streams*. R Package version 0.6-1, URL http://CRAN.R-project.org/package=rlecuyer.

Tierney L, Rossini AJ, Li N, Sevcikova H (2012). "snow: Simple Network of Workstations." R Package (v:0.3-9), URL http://cran.r-project.org/package=snow.

Urbanek S (2011). *multicore: Parallel processing of R code on machines with multiple cores or CPUs*. R Package version 0.1-7, URL http://CRAN.R-project.org/package=multicore.

Vertenstein M, Craig T, Middleton A, Feddema D, Fischer C (2011). *CESM1.0.4 User's Guide*. NCAR. URL http://www.cesm.ucar.edu/models/cesm1.0/cesm/.

Weston S (2010). *doMPI: Foreach parallel adaptor for the Rmpi package*. R Package version 0.1-5, URL http://CRAN.R-project.org/package=doMPI.

Wickham H (2009). *ggplot2: elegant graphics for data analysis*. Springer New York. ISBN 978-0-387-98140-6. URL http://had.co.nz/ggplot2/book.

Yu H (2012). *Rmpi: Interface (Wrapper) to MPI (Message-Passing Interface)*. R Package version 0.6-1, URL http://CRAN.R-project.org/package=Rmpi.

Index