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

July 18, 2013

This publication was typeset using LaTeX. Illustrations were created using the **ggplot2** package (Wickham, 2009), native R functions, and 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.

Disclaimer

Warning: The findings and conclusions in this article have not been formally disseminated by the U.S. Department of Energy and should not be construed to represent any determination or policy of University, Agency and National Laboratory.

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

There is nothing new under the sun. It has all been done before.

—**Sherlock Holmes**

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 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 packages (Chen *et al.*, 2012a,c; Schmidt *et al.*, 2012a,b; Patel *et al.*, 2013; 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.

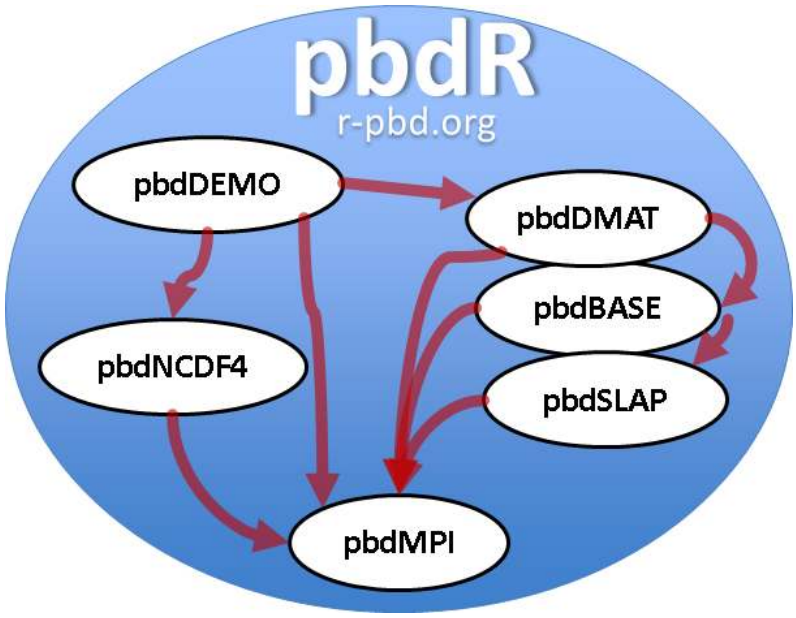

Figure 1.1: pbdR Packages

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,

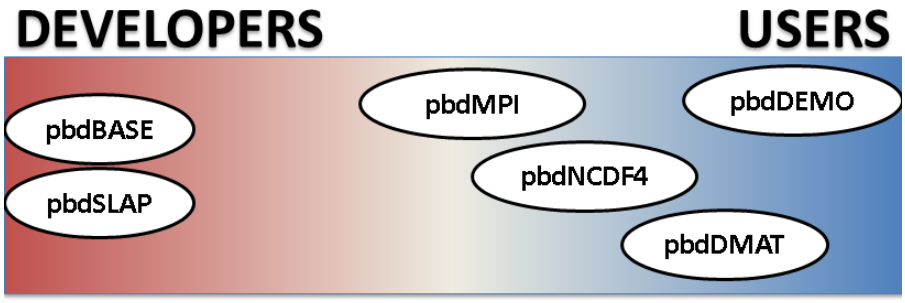

Figure 1.2: pbdR Package Use

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.

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+

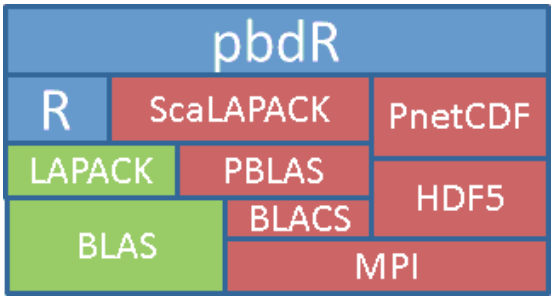

Figure 1.3: pbdR Interface to Foreign Libraries

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 6
# 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 8
# 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 9
# 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 10
# 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_gbdc, package='pbdDEMO', ask=F,
    echo=F)"

### Chapter 11
# Loand/unload balance
mpiexec -np 4 Rscript -e "demo(balance, package='pbdDMAT', ask=F,
    echo=F)"
# GBD to DMAT
mpiexec -np 4 Rscript -e "demo(gbd_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 13
# 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_gbd, package='pbdDEMO', ask=F, echo=F)"
Rscript -e "demo(iris_dmat, package='pbdDEMO', ask=F, echo=F)"

### Chapter 14
mpiexec -np 4 Rscript -e "demo(task_pull, package='pbdMPI', ask=F,
    echo=F)"
mpiexec -np 4 Rscript -e "demo(phyclust_bootstrap, 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

My name is Sherlock Holmes. It is my business to know what other people don't know.
—**Sherlock Holmes**

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 **??**.

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

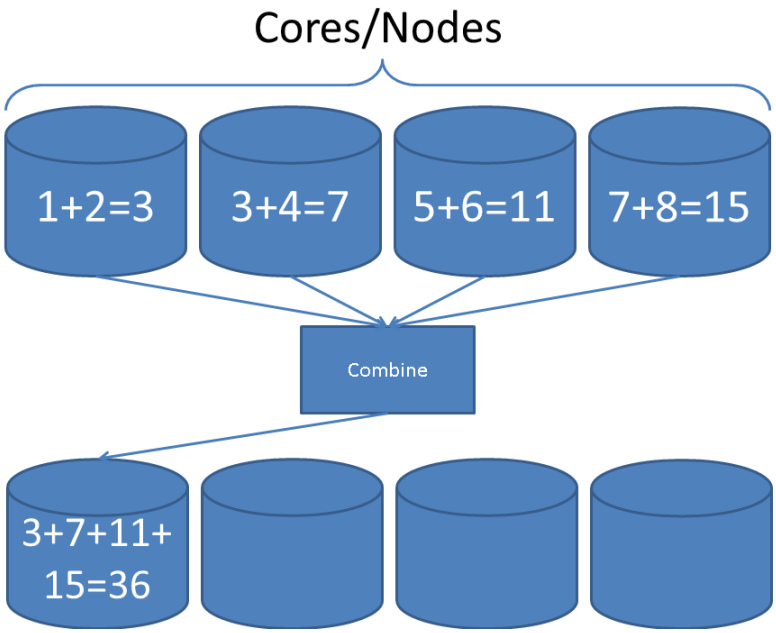

Figure 2.1: Task Parallelism Example

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

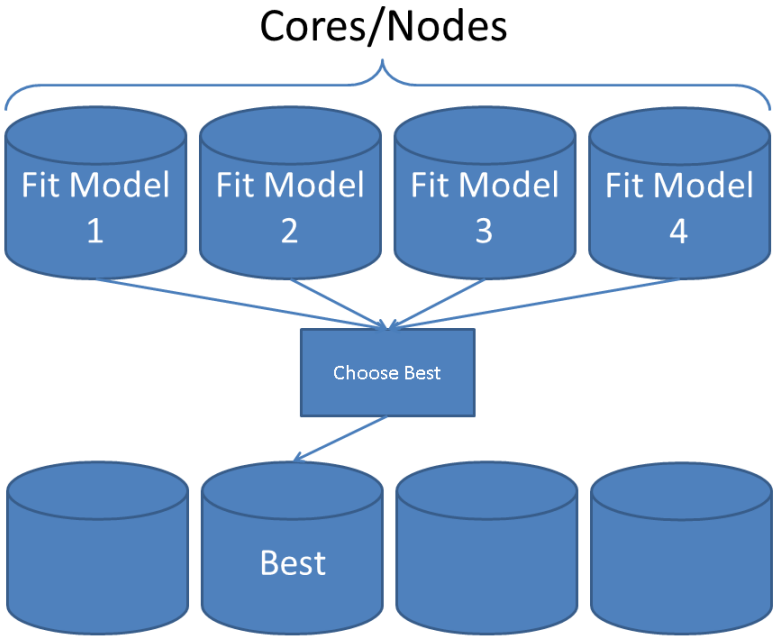

Figure 2.2: Task Parallelism Example

```
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 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.*, 2012b) 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/workers[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]Sometimes referred to as "master/slaves" or "master/workers"

2.3 Notation

Note that we tend to use suffix `.gbd` for an object when we wish to indicate that the object is "general block 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 `.gbd` 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.

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 `.gbd` object. As with `.gbd`, this carries no semantic meaning, and is merely used to improve the readability of example code (especially when managing both "`.gbd`" 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 Code

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

id <- get.jid(N)

```

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

See Section **??** for more efficient task parallelism.

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.

References

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

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

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

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

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.

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 (2013). "pbdNCDF4: Programming with Big Data – Interface to Parallel Unidata NetCDF4 Format Data Files." R Package, URL http://cran.r-project.org/package=pbdNCDF4.

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 (2013). "pbdDEMO: Programming with Big Data – Demonstrations of pbd Packages." R Package, URL http://cran.r-project.org/package=pbdDEMO.

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.

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