Guide to the pbdDMAT Package

Version 2.0

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

[1]Remote Data Analysis and Visualization Center
University of Tennessee,
Knoxville, TN, USA

[2]Computer Science and Mathematics Division,
Oak Ridge National Laboratory,
Oak Ridge, TN, USA

Contents

This publication was typeset using LaTeX.

Acknowledgement

Ostrouchov, Schmidt, 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 thank our colleague, Ed D'Azevedo from the Computational Mathematics Group, Computer Science and Mathematics Division, Oak Ridge National Laboratory (ORNL), for his discussions and illuminating advice using ScaLAPACK and distributed matrix computation.

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.

We also thank Douglas Bates, for the enlightening discussions in numerically solving linear least squares problems. Finally, we thank Dirk Eddelbuettel for his discussions on using Rcpp in large scale, high performance computing.

Abstract

With the size of data ever growing, the use of multiple processors in a single analysis becomes more and more a necessity. The Programming Big Data in R (pbdR) project attempts to address the R language's current shortcomings in parallel distributed computations. The **pbdDMAT** package for R provides high level S4 classes and methods for block-cyclic distributed matrices. These methods focus on computationally burdensome problems with block-cyclically distributed data, such as linear algebra and statistics, in such a way that the syntax for operating on new data types mimics R's syntax for ordinary matrices as closely as possible. This allows someone already familiar with R syntax to achieve vast speed improvements via parallelism with a mostly already familiar syntax, and without the need to learn complicated parallel programming techniques. Much of the heavy lifting is performed by the **pbdBASE** package, which utilizes the PBLAS and ScaLAPACK libraries. In addition to performance improvements through parallelism, use of this system with more than one processor allows the user to break R's local memory barrier, namely the requirement that a vector be indexed by a 32-bit integer, by only storing subsets of the vector on each processor.

1. Introduction

The Programming with Big Data in R (Ostrouchov *et al.* 2012) project, abbreviated pbdR or just pbd, seeks to elevate the R language to supercomputers. The **pbdDMAT** package (Programming with Big Data: Distributed Matrix Algebra Computation) (Schmidt *et al.* 2012b) is a (mostly) implicitly parallel system for doing distributed matrix computations in R. (R Core Team 2012) It offers a new `S4` class `ddmatrix` and numerous high level methods for this object, which (intentionally) very closely resemble the existing R syntax for non-distributed matrices. Much of the heavy lifting — especially that involving distributed linear algebra — is handled by the **pbdBASE** (Schmidt *et al.* 2012a) package, which efficiently utilizes the well-known high performance libraryes ScaLAPACK and PBLAS (Blackford *et al.* 1997).

Ordinarily, a user of these libraries would have to deal with a great many more headaches than the user of **pbdDMAT**. Of note, a user of the **pbdDMAT** system can achieve massive performance boosts with only the most minimal interaction with the more cumbersome sides of ScaLAPACK, such as the MPI layer for ScaLAPACK, the Basic Linear Algebra Communication Subroutines (BLACS) (Dongarra and Whaley 1995). The peculiarities of using these libraries, such as local storage issues, descriptor arrays, and BLACS grids are very much still there, but almost all of these problems have been abstracted away for the user. Distributed matrix algebra has never been easier, and more importantly, massively scalable computational statistics has never been more accessible.

The principal goal of the **pbdDMAT** package is to provide R users with access to extremely powerful distributed, implicitly parallel computation, all while preserving the friendly and familiar R syntax for these computations, so that effectively, much existing R code could used with this system with only trivial modifications, yet receive massive performance boosts. Updates and bug releases for this and other **pbd** projects may, especially while in infancy, be much more frequent than CRAN releases. So for up to date packages, as well as evolving information about the **pbd** project, see the pbdR project's github `http://code.r-pbd.org` or our website http://r-pbd.org/.

1.1. Achievements

The pbdR series of packages offer the R user several new advancements. First, by hiding the distributed details of ScaLAPACK, we offer (near) ScaLAPACK speeds and scaling to many cores, but with R syntax. Second, by distributing the objects across processors, we are able to largely overcome R's memory barrier.

At present[1], it is impossible to index native R objects with a 32-bit integer. Since a matrix in R is really just an array, this means that the largest square matrix it is possible to store in R is roughly a $46,000 \times 46,000$ matrix. This imposes two restrictions on the **pbd** system. First, the global dimension of any matrix used at this time with the **pbd** toolchain must have dimensions indexable by a 32-bit integer. Namely, no single dimension of the "full", global matrix may have more than

$$\left(2^{32-1} - 1\right)^2 \approx 4.612 \times 10^{18}$$

because each dimension must be an integer, and in R terms, that means a 32-bit integer.

By comparison, the largest matrix which a single R process can hold has

$$\begin{aligned} 2^{32-1} - 1 &= 2,147,483,647 \\ &\approx 2 \times 10^9 \end{aligned} \tag{1}$$

numeric elements. Thus, the largest amount of data that can be analyzed in serial R at the time of writing is 16gb. However, while benchmarking pbdR packages, we have been working well in excess of this limit no problems. For example, the largest square matrix that serial R can store is $46,340 \times 46,340$, for a total of 15.99934gb of data. However, for example, we have published a benchmark (Schmidt *et al.* 2012d) which uses a $100,000 \times 100,000$ matrix, for a total of 74.50581gb of data.

However, we note that getting near the theoretical upper bound in (1) with the **pbd** system is effectively impossible, because each local R process will store at most roughly 10^9 elements. So even with 100,000 cores, you are still solidly within this boundary. Indeed, a user with N processors is able to store a square distributed matrix up to size

$$N \times \left(2^{32-1} - 1\right)$$

So at this time, a user would need 1024 cores to comfortably be able to analyze a terabyte of data, and over 100,000 cores to approach petabyte scale.

1.2. Intended Audience

The **pbdDMAT** package depends on **pbdBASE**, and so anyone who wishes to use the former package must first install **pbdBASE**. However, much of the direct use of **pbdBASE** is intended only for extremely advanced users and developers. A few exceptions are the `init.grid()` and `finalize()` functions, which are detailed in the **pbdBASE** vignette (Schmidt *et al.* 2012c). The overwhelming majority of the remaining functions are either internal or for people deeply

[1] Though this is expected to change by summer 2013

familiar with ScaLAPACK, and so, especially while learning, focus should be spent on **pbdDMAT**.

1.3. Installation

The **pbdDMAT** package is available from the CRAN at http://cran.r-project.org, and can be installed via a simple

R Script

```
install.packages("pbdDMAT")
```

This assumes only that you have MPI installed and properly configured on your system. If the user can successfully install the package's three principal dependencies, **pbdMPI** (Chen *et al.* 2012a), **pbdSLAP** (Chen *et al.* 2012c), and **pbdBASE** (Schmidt *et al.* 2012a) (each available from the CRAN), then the installation for pbdDMAT should go smoothly. If you experience difficulty installing either these packages, you should see their documentation.

1.4. Package Examples

One can quickly get started with **pbdDMAT** by learning from the following eight examples:

Shell Script

```
### Under command mode, run the demo with 2 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 2 Rscript -e "demo(basic1, package='pbdDMAT', ask=F,
   echo=F)"
mpiexec -np 2 Rscript -e "demo(basic2, package='pbdDMAT', ask=F,
   echo=F)"
mpiexec -np 2 Rscript -e "demo(basic3, package='pbdDMAT', ask=F,
   echo=F)"
mpiexec -np 2 Rscript -e "demo(reductions, package='pbdDMAT', ask=F,
   echo=F)"
mpiexec -np 2 Rscript -e "demo(matprod, package='pbdDMAT', ask=F,
   echo=F)"
mpiexec -np 2 Rscript -e "demo(solve, package='pbdDMAT', ask=F,
   echo=F)"
mpiexec -np 2 Rscript -e "demo(svd, package='pbdDMAT', ask=F, echo=F)"
mpiexec -np 2 Rscript -e "demo(cholesky, package='pbdDMAT', ask=F,
   echo=F)"
```

2. Terminology

Before beginning, we will make frequent use of concepts from the Single Program/Multiple Data (SPMD) paradigm. If you are entirely unfamiliar with this approach to parallelism, or if you are unfamiliar with the **pbdMPI** package, then you are strongly encouraged to read the vignette (Chen *et al.* 2012b) contained in the **pbdMPI** package, as well as examine and digest its many examples in order to better understand what follows.

A concise explanation of SPMD is that it is an approach to parallel, distributed programming in which one program is written, and each processor runs that same program, though that program locally will often be interacting with different data. This, in contrast to the manager/worker paradigm where one processor, the manager, is in charge of its workers, each of whom swear fealty to the manager. So in SPMD, each processor believes itself to be the manager, the one in charge. As a colleague, Dr. Russell Zaretzki put it, "it's like academia."

Throughout the remainder, we will be discussing a distributed matrix object, and wish to do so with some standardized terminology. A matrix is of course a rectangular collection of numbers. A *distributed matrix* then is just a matrix which has been decomposed in some fashion so that each processor only owns a piece of the "whole" matrix. For us, this decomposition is non-overlapping, so that if one processor owns some piece of the "full" matrix, then no other processor owns that same piece. The "whole" matrix (which need not ever actually exist, except theoretically, at any time), rather than pieces of it distributed among the processors, will be referred to as a/the *global* matrix. Loosely speaking, the global matrix is what we are really thinking of when we deal with the sub-pieces of the distributed matrix.

In the SPMD paradigm, each processor, though only owning a piece of the whole (henceforth referred to as the *local matrix* or *submatrix*, relative to that processor), will call functions on that matrix exactly as one would with an ordinary, non-distributed matrix on a single processor. The difference for the user is minimal; all the "heavy lifting" which explicitly handles the distributed nature of the object is performed in the background.

Matrices, distributed or otherwise, have dimensions — that is, lengths of the number of rows and the number of columns in the rectangle. The global matrix has a *global dimension*, and this is a global value, i.e., this value does not vary from processor to processor. Every processor agrees as to the size of the "full" matrix, otherwise we would have anarchy. However, the local matrices, in practice, will differ from processor to processor, and so too should their *local dimensions*. A local dimension, as the name implies, is the dimension of the submatrix, relative to a particular processor.

3. Classes and Methods

3.1. Class ddmatrix

The package **pbdDMAT** contains one new class, namely the class `ddmatrix` which stands for **d**istributed **d**ense **matrix**. This S4 class serves as a container for a distributed matrix type, consisting of the members:

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

with prototype

$$\texttt{new("ddmatrix")} = \begin{cases} \textbf{Data} & = \texttt{matrix(0.0)} \\ \textbf{dim} & = \texttt{c(1,1)} \\ \textbf{ldim} & = \texttt{c(1,1)} \\ \textbf{bldim} & = \texttt{c(1,1)} \\ \textbf{CTXT} & = 0 \end{cases}$$

We will discuss the last two items in more detail in the later sections, particularly Section 6 and Section 8.

3.2. Computational Methods

The **pbdDMAT** package also contains numerous methods for class `ddmatrix`, including `'['` for subsetting (with global indices), `lm.fit()` for linear models, and numerous others. These methods, when replicas of serial R functions, have very similar (usually identical) look and feel to their R counterparts. A complete list can be obtained by opening an interactive R session and entering the command:

```
showMethods(class="pbdDMAT::ddmatrix")
```

For complete details, see the **pbdDMAT** reference manual.

4. Information for Users

4.1. The General Procedure for Using the System

To use **pbdDMAT**, there are a few special considerations one must keep in mind which separate the system over using ordinary R. These are

1. In addition to the ordinary MPI communicator provided by **pbdMPI**, a special, rectangular MPI communicator is used. (Dongarra and Whaley 1995)
2. Data is block cyclically distributed across the rectangular process grid. (Blackford *et al.* 1997) (link)

The first item is fairly simple. Simply, you will start every analysis using the **pbdDMAT** package by making a call to the function `init.grid()`. The latter is slightly more complicated; for the moment, we will merely say that one achieves this with a parallel reader or data distribution function. See the package reference manual, as well as the later sections of this vignette for more details.

A generic skeleton of what a typical analysis using **pbdDMAT** looks like involves 4 steps (which will be discussed in detail in the sections to follow):

1. Initialize the process grid. (`init.grid()`)

2. Read the data into the processes, store it as a distributed matrix.
3. Call R functions exactly as you would with ordinary matrices (when applicable).
4. Collect your results and finalize (`finalize()`).

One can generally use the above steps on existing R scripts with only a few minor modifications, and quickly parallelize his or her serial code. For most users, this will amount to simply adding in the appropriate calls to `init.grid()`, `finalize()`, and a parallel data reader or data distributor function (such as `as.ddmatrix()`). However, there are a few hitches with setting up a process grid and with distributing/parallel reading. We discuss these issues in the following two sections.

4.2. Printing in Parallel

Printing in parallel can take some getting used to, especially in SPMD style programs. If you simply issue the order `print(x)`, then every process will print, but it is often a cacophony of undecipherable writing out to the terminal, with each process trampling the others.

To this end, you should learn to make extensive use of the **pbdMPI** function `comm.print()`. For instance, if you have two processors, each with an object `x`, but with disagreement to what `x` actually is — say for example, one thinks `x` is 1, and the other thinks it is 2. Then calling `comm.print(x)` will print two pieces of information: the processor printing the value (here process 0) and the value itself, 1. By default, the only processor to print is that with MPI communicator address 0. You can have all processor ranks print using the optional argument `all.rank=TRUE`. This will print all values stored for the requested object, but will do so "one at a time". You can also disable the printing of which processor is doing the printing via the optional argument `quiet=TRUE`. See the official `pbdMPI` documentation for details.

Additionally, there is a method for the `print()` function when applied to a distributed class such as `ddmatrix`. This will print some brief information about the matrix from processor 0. This has the optional argument `all=`, which can be used to print the entire matrix, one line at a time.

This function should not be called from `comm.print()`. Actually, one of the easiest ways to get yourself into trouble and hang up all the processors is to call a function which requires communication between processors from inside something like `comm.print()`.

4.3. Process Grid Size

Recall that we will very frequently visualize the processors as being in a 2-dimensional processor grid. This grid is initialized via the function `init.grid()`, which accepts the optional arguments `nprows=` and `npcols=`. If these are left blank, then a reasonable choice will be made based on the number of available processors. Here "reasonable" means "as close to square as possible." The inspired reader can find more detail within the ScaLAPACK User's Guide (Blackford *et al.* 1997) as to why this is a good choice. (link)

To reiterate, in most cases, taking `nprow` and `npcol` as close to each other as possible is "sufficiently good". Leaving the `nprow=` and `npcol=` options blank will make this choice for

you. For example, we note that the user should be aware that providing 37 cores may not perform as well as providing 36 cores in the form of a 6×6 process grid. There is a strong connection between the process grid and the block-cyclic distribution, which we will discuss further in the following section.

Additionally, we note that the `init.grid()` function accepts an additional argument `ICTXT=`, which we will discuss in Section 8. This argument is also discussed in detail in the package reference manual.

4.4. Distributing Data

When distributing data, you must use a *blocking factor*. This is a pair of numbers (a, b), and unless you think you have a great reason to do otherwise, you should have $a = b$. If you have no intuition, the just make them equal. The scale of these numbers should generally correspond to the size of your process grid and the scale of the data. The choice of blocking factor can seriously impact performance, because it is intimately tied to the data distribution. We will spare the details for the moment, and merely say that the blocking factor constitutes a tradeoff. Smaller values, down to $(1, 1)$, mean that there will be more parallelism in many of the matrix algebra routines but also increase communication costs. On the other hand, larger values, which could be larger than the dimensions of any one of your matrices, will limit communication between the processors, but naturally also limit the parallelism.

A good choice of blocking factor can be difficult, and in the author's opinion, requires some intuition gained through experimentation. A discussion on this topic can be found in the ScaLAPACK User's Guide. (Blackford *et al.* 1997) (link) However, we note that **pbdDMAT** defaults to a 4×4 blocking factor, which is probably an acceptable choice if you do not know what else to do. For very large matrices, this blocking could be a bit small, and so scaling it up by powers of 2 (8×8, 16×16, $\ldots$, 256×256) may be warranted. The blocking does not have to be by power of 2, but this is a convenient way to do business. The performance-hungry user is encouraged to experiment with various blocking factors across various processor grids with various matrix sizes to develop intuition.

4.5. Reading Data In Parallel

To really get the most out of this system, you need to read the data into R in a parallel distributed fashion. This generally necessitates the use of a parallel file system, such as Lustre. These are the kinds of resources that one generally does not have on his/her laptop, unfortunately. It is possible to read all of the data in on one core and have that core distribute the data to all the other processes, which is what one should do in the absence of a parallel file system. However, these added communication costs could overtake the gains provided by distributed computation, depending on the task. Worse, the user is again trapped in the world of 32-bit integer indexing, meaning the size of problem that it is possible to solve shrinks.

As a general rule, if you are on a smaller system with limited resources, you do what you must. If you are on a larger system with luxurious resources, you really should know better, and act accordingly. Failure to do so will significantly negatively impact performance with this system.

Some functions which are useful in this regard are `as.ddmatrix()`, `distribute()`, and `redistribute()`. See the reference manual for details.

4.6. Controlling Defaults

The internal default blocking factor can be modified very simply by modifying the object `.BLDIM`. So for example, if the user wishes to have a default data blocking of 16×16 then it is sufficient to enter the command:

```
.BLDIM <- c(16, 16)
```

Similarly, the internal default BLACS context is 0, but it can be controlled by changing the value of the object `.ICTXT`. So if the user wishes to have BLACS context 2 as the default, then he/she need only enter the command:

```
.ICTXT <- 2
```

5. Examples

5.1. Example 1

Let's take a look at some example code. Here, we will do some distributed matrix multiplication, as well as solving some systems of equations. You probably should not use a large process grid for this problem. Anything bigger than 8×8 is *massive* overkill. A 2×2, 2×4, or 4×4 process grid (4, 8, or 16 processes respectively) should be more than plenty — so a humble laptop or desktop should more than suffice. You've got to crawl before you can drag race.

Throughout, we will preface distributed objects by a *d*, purely for pedagogical reasons. Non-distributed objects will not have any preface. So `x` is not distributed, but `dx` is.

To convince you that this new stuff is really doing the same things as the old stuff, we are going to randomly generate a 500×500 matrix on process 0, and then distribute that matrix across the process grid, using a 32×32 blocking dimension. If you are using more than 4 processes, you might consider backing that off to 16×16, but it's not really necessary; remember, the purpose here is to learn.

Generating Test Data

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

# Number of rows and columns to generate
nrows <- 5e2
ncols <- 5e2
```

```
mn <- 10
sdd <- 100

# ScaLAPACK blocking dimension
bldim <- c(4, 4)

# Generate data on process 0, then distribute to the others
if (comm.rank()==0) {
   x <- matrix(rnorm(n=nrows*ncols, mean=mn, sd=sdd),
      nrow=nrows, ncol=ncols)
   b <- matrix(rnorm(n=ncols*2, mean=mn, sd=sdd), nrow=ncols,
      ncol=2)
} else {
  x <- NULL
  b <- NULL
}

dx <- as.ddmatrix(x=x, bldim=bldim)
db <- as.ddmatrix(x=b, bldim=bldim)

# continued in the next block of code ...
```

The basic explanation for what this is doing is:

- **Load the package** with `library(pbdDMAT)`
- **Use `init.grid()` to initialize the process grid** and MPI communicator(s). You can optionally specify `nprow=` for the number of process rows here, and `npcol=` for the number of process columns. Not specifying means that the function will choose the "best" option for you, meaning a grid that is as close to square as possible for the number of processors you have given it.
- **Generate the** 500×500 **ordinary non-distributed R matrix** of random normal data with mean 10 and sd 100 on process 0. Here, we use the **pbdMPI** function `comm.rank()` to make sure that only process 0 generates the data. The other processes store `NULL` in `x`. Likewise, we do the same for the 500×2 vector `b`/`db`, the "right hand sides" for the systems we will be solving.
- **Distribute the data** from process 0 and store it as a distributed matrix named `dx`. Here we specify a blocking dimension of 32, so really 32×32. ScaLAPACK and PBLAS routines usually require square blocking, so while we could block in many other ways, like 32×16, this may not be a good plan. Whenever the `bldim=` option is present, specifying only a single integer `n` will always be equivalent to specifying `c(n, n)`.

Before continuing, it is good to reiterate that this is not an efficient way to do business if you are using many processes. You need to use multiple processes to either read in the matrix into pieces from disk in parallel, or you need to do random generation in parallel using multiple

(perhaps all) processes.

Now that we have our data in the right format, it's pure easy street from here. Now we just forget that there is anything distributed at all going on and write our code exactly as we would with plain, vanilla R. First, we will multiply the transpose of `dx` and calculate the inverse of this product, storing the result in `dx_inv`. Finally, we solve the system of equations with two right hand sides `dx_inv * solns = db`.

Simple Matrix Operations

```
# Computations in parallel
dx_inv <- solve( t(dx) %*% dx )
solns <- solve(dx_inv, db)

# continued in the next block of code ...
```

Notice that we're doing matrix computations just the same way you would with vanilla R. And to prove that it really is the same, we can undistribute our results and "check our work":

Comparing Results to R

```
# Undistribute solutions to process 0
pbd_dx_inv <- as.matrix(dx_inv, proc.dest=0)
pbd_solns <- as.matrix(solns, proc.dest=0)

# Compare our solution with R's --- not in parallel
if (comm.rank()==0) {
  r_x_inv <- solve( t(x) %*% x )
  r_solns <- solve(r_x_inv, b)

  print(all.equal(pbd_dx_inv, r_x_inv))
  print(all.equal(pbd_solns, r_solns))
}

# shut down the MPI communicators
finalize()
```

The above script is in the **pbdDMAT** directory, located at `inst/examples/pbddmat_example1.R`. To run the code, you would make a batch execution call. Say you have 4 processors you wish to use for this analysis. Then you could execute the script via the command:

```
# replace the 4 below with hoever many processors you actually want
    to use
mpirun -np 4 Rscript pbddmat_example1.R
```

from a terminal. If everything works correctly, then two `TRUE`'s will print to the terminal.

5.2. Example 2

First we will generate some data on process 0, or (0,0) using the grid notation. We will do this using a simple `if()` together with the **pbdMPI** function `comm.rank()`, which returns

the MPI communicator number for the calling process. Any process not initially storing any data should store `NULL` for the object. This probably isn't the way you will want to run your production code, especially if you are randomly generating data; in that case, it would be much more efficient to just generate what is needed on each processor. Although doing this requires that you have access to good seeds for parallel random number generation; for more information, see the documentation on setting seeds via `comm.set.seed()` in the **pbdMPI** package, or for more control, see the **rlecuyer** or **rsprng** packages.

There is merit, however, in operating in this way. This is somewhat like the process necessary for reading in data onto a subset of processors (just 1 if you do not have access to a parallel file system) and then distributing that out to the larger grid, so it is a useful skill. For more information about this procedure, see Section 7.

Generating Test Data

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

comm.set.seed(diff=TRUE) # independent RNG streams via rlecuyer

nrows <- ncols <- 10 # generate a 10x10 matrix
.BLDIM <- 2 # the blocking factor for distribution


dx <- ddmatrix("rnorm", mean=10, sd=100, nrow=nrows, ncol=ncols)
x <- as.matrix(dx) # for confirmation

# continued in the next block of code ...
```

To convince ourselves that the data is distributed, we can inspect the new object in several ways:

Printing the Object

```
print(dx)

comm.print(submatrix(dx))

comm.print(dx)

# continued in the next block of code ...
```

Here, `print()` is a special method that shows you the slots of your distributed matrix. The `submatrix()` function will show the local submatrix (syntactic sugar for printing `dx@Data`). Use of **pbdMPI**'s `comm.print()` ensures that only process 0 will print the result. Finally, just using R's print method on the object in `comm.print(dx)` will produce an uglier version of `print(dx)` and `comm.print(submatrix(dx))`.

We can also do insertions and extractions:

Insertion and Extraction

```
dx[1,1] <- NA # insertion indices are global
comm.print(submatrix(dx)[1,1], all.rank=T) # see?

comm.print(dim(dx))
dx <- dx[, -2]
comm.print(dim(dx))

nona <- na.exclude(dx)

# continued in the next block of code ...
```

Finally, we can convert the distributed matrix back into an ordinary R matrix on processor 0. You probably will not need to do this very often in production code, because in practice, you could be dealing with matrices with so many elements that they will not fit into a single R process. For testing however, this process can be very useful. It could also conceivably have utility for dealing with $n \times 1$ matrices.

Insertion and Extraction

```
# convert back
nona <- as.matrix(nona, proc.dest=0)

# compare our results with R --- notice the syntax is
   essentially identical
if (comm.rank()==0){
  x[1,1] <- NA
  x <- x[, -2]
  r_nona <- na.exclude(x)

  all.equal(r_nona, nona)
}

finalize()
```

In the above script, there is one addition over the previous pieces. Namely, we include several calls to `comm.cat()`. All this does is demand line breaks (via the regular expression `\n`) for more human-readable printing.

The script file is available in the **pbdDMAT** directory, under `inst/examples/pbddmat_example2.R`, and you can run this script from the command line with the following command:

```
# replace the 4 below with your number of processors
mpirun -np 4 Rscript pbddmat_example2.R
```

If you want to ramp up the size of the problem and the number of cores, you may want to change the `nrows`, `ncols`, and `BL` definitions (these are good to experiment with regardless).

6. Information for Advanced Users

In this section, we will discuss the block-cyclic distribution of data across a 2-dimensional processor grid in lengthy detail. It probably goes without saying that before beginning this section, the reader should be familiar with all sections prior.

6.1. Blocking Factor

The idea is fairly simple, even if the execution can sometimes be cumbersome. We want to try to evenly balance the data distribution for "large" matrices across the process grid, but in a way that is congruent with the natural blocking of matrices when performing LAPACK operations on them. As the name implies, we imagine taking a large global matrix and chopping it into blocks, and assigning those blocks to the processors in the grid. The way this is done is far from arbitrary, however.

For simplicity, let us explicitly assume for the moment that we are going to distribute a 9×9 matrix across a 2×3 process grid, using a 2×2 blocking factor. The above link may be extremely useful to your understanding the following. For those who flatly refuse to click the link, we can visualize the process as follows. We can imagine our global matrix (even if we never actually have a globally stored matrix, this is the way we would imagine it in our heads) as looking like:

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

and our process grid looks like:

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

with the usual MPI processor rank on the left, and the corresponding BLACS processor grid position on the right.

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.

Two visual representations of this block cyclic data decomposition can be seen in Figure 1

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

Figure 1: Block Cyclic Data Decomposition Example

and Figure 2 . As you can see "all animals are equal, but some animals are more equal than

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

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

Figure 2: Local Processor Storage View of the Block Cyclic Data Decomposition

others." Meaning that we don't inherently penalize any processor or go out of our way to imbalance the data load, but that it is possible to do so with poor choice of blocking factor, and that generally process $(0, 0)$ in the BLACS grid will receive the most data. Additionally, notice that matrices that are closer to being square will distribute better than will vectors. In this case, a vector (here, a 9×1 matrix) would only distribute across the processes in the first process column, namely $(0, 0)$ and $(1, 0)$. This isn't necessarily the great imbalance problem you might instinctively think it is. These matrices are generally fairly small, so it's not really that big of a deal that they do not distribute well. On the other hand, things close to square are really quite large, and so we should go out of our way to make sure that they distribute "well", whatever that really means.

Now, the user should never need to develop an algorithm to perform this kind of block-cyclic

decomposition of the data; it may be necessary, especially for a developer who wishes to use this system, to have to deal with this data distribution business in a very up-front way, but many helper functions are provided by the package to that end. We bring up this issue to illustrate a critical point: the choice of blocking can make a large difference in performance, and generally should be tailored to the process grid.

If the blocking factor (in the above, 2×2) is too "big" (relative to the process grid), then the data distribution will be very uneven. This has the net effect of reducing communication times between processes, but also limiting the amount of parallelism possible. On the other hand, if the blocking factor is too "small", then the data distribution is "too fair", gaining much parallelism but massively inflating communication costs. At the extreme ends of this are, for the former, using a blocking factor that would encompass the entirety of the matrix (so that the data is distributed to only one process — no communnication, no parallelism), and for the latter, using a blocking factor of $(1, 1)$, so that there is effectively nothing *but* communication and parallelism.

Very loosely speaking, parallelism is good and communication is bad, but in practice, we can't really have one without the other.

6.2. Managing BLACS Contexts

A very advanced user, perhaps someone already familiar with ScaLAPACK, may wish to be able to use a variety of BLACS contexts. Both the creation and destruction of BLACS contexts is discussed in detail in the **pbdBASE** vignette and reference manual, and so the reader should refer to those sources for details.

7. Advanced Example

In this section, we will look at an example of how to create and utilize additional BLACS contexts, and make a pretend parallel reader.

Let's take a look at an example using the ideas discussed in Section 6. We will again be using 4 processors in the form of a 2×2 grid. However, this time, we are going to randomly generate a matrix on 2 processors and then distribute that data onto the full 2×2 grid. We will achieve this by creating a new BLACS context with 2 processor rows and 1 processor column. Processors not in this grid will have the BLACS communicator grid locatiosn of $(-1, -1)$.

For clarity, because this can be a confusing and deeply frustrating concept at first. Every processor is part of the BLACS context; *all of them*. The context is just a collection of information that makes it possible to perform local operations with global communications. Think of it like an R list; it's just a place to put things. Each processor stores its own copy of this "list", with some pieces common to all processors and some differing from process to process. The objects within that list are the BLACS context number (just an identifying name more so than actual information), the number of global processor rows/columns, and that processor's position in that grid. Whenever a function relying on BLACS (like all PBLAS and ScaLAPACK functions) is called, a context is required. The local processor's grid location value of

$(-1, -1)$ when that processor is not *really* part of the grid is the BLACS equivalent of `NULL`. It's just a placeholder that lets the local processor know that it isn't part of the party. But that processor is still aware of the BLACS context, which is a global number, and the total number of processor rows/columns (also global). So it is the location in the processor grid that is $(-1, -1)$, not the context. The context is just a "name", which happens to be a global integer common to all processors.

The choice of 1 processor column is not random. Doing so demands that contiguous rows are stored on the processes, and the cyclic distribution is occurring between rows. This is a very convenient way to do business if you must read in data from files (rather than randomly generate it in memory). Here, we can think of each processor "reading in" the parts of the (imaginary) file, and then distributing that data out to BLACS grid 0 for analytics. Other, "non-vector" BLACS grids are certainly possible as well.

A word of caution; the seeds here on the different processors are not guaranteed to be independent. This is just for the sake of demonstration. Refer back to the comments in Section 5.

Generating in Parallel

```
library(pbdBASE, quiet=TRUE)

init.grid(nprow=2, npcol=2)

# find the minimum value possible for a new BLACS context
# the value should be 3
newctxt <- minctxt()
comm.print(newctxt)

# create new grid
init.grid(nprow=2, npcol=1, ICTXT=newctxt)

# store new grid information
grid <- blacs(ICTXT=newctxt)

# "read in" the data and distribute
if (grid$MYROW == -1 || grid$MYCOL == -1){
  x <- matrix(0)
} else {
  x <- matrix(rnorm(50), ncol=10)
}

dx <- new("ddmatrix",
          Data=x,
          dim=c(10,10), ldim=dim(x), bldim=c(5,10),
             CTXT=newctxt)

dx <- redistribute(dx, bldim=2, ICTXT=0)
```

```
print(dx)

# close grid
gridexit(newctxt)

blacsexit()

comm.print("MPI still works")

finalize()
```

A few comments. First, we do not technically need to call `gridexit()`, since the following `blacsexit()` call will shut down *all* BLACS grids. However, inbetween those two calls, we could make more calls to routines using BLACS grids 0, 1, and 2. Second, if we explicitly call `blacsexit()` as we do here, then all BLACS grids are shut down without shutting down the MPI communicator. However, `finalize()` will do this for us if we forget, because a failure to do so can cause memory leaks. So you do not have to explicitly call `blacsexit()` unless you are done with BLACS, have yet more MPI work to do, and want to free up the resources.

Additionally, notice that here we get friendly with the `new` constructor. This call is part of R's S4 methods, and instantiates a — as the name implies — new object of specified class with specified slots. Notice that the blocking dimension is the dimension of the local matrix. This is so because we are imagining reading in large, contiguous blocks for each processor (here with just 1 cycle). This is fairly ad hoc, but is useful for demonstration purposes.

Finally, you may be wondering why we would even bother with this approach with contexts rather than simply explicitly choosing a subset of processors from context 0. We could do this as well, but this isn't quite as simple as you might think, especially with the tools already built (meaning you may have to work much, much harder for this). You are encouraged to simply construct a new BLACS context as in the example, because for this very low-level data wrangling, it can make your life much simpler.

Finally, this script is available in the **pbdBASE** directory, under `inst/examples/base_eg.R` and is meant to be run with 4 processors.

8. Information for Developers

In this section, we will discuss some issues that are not really important for anyone except those who need to develop new methods which rely on **pbdBASE**. For the remainder, we will make mention and use of R's S4 method of object oriented programming, and we will assume that the reader has at least a working familiarity with S4. There are several fine introductions to S4 methods available, including those from (Chambers 2006) and (Genolini 2008). It probably goes without saying that before beginning this section, the reader should be familiar with all sections prior.

8.1. Class `ddmatrix`

Every local submatrix must be a matrix containing at least one value, even if it technically should not have anything. You will unleash unspeakable misery on yourself if you use `matrix(nrow=0, ncol=0)` for the submatrix, in particular when passing down to ScaLAPACK routines (the motivating example for the existence of **pbdBASE**). Instead, you should use the routine `ownany()`, a simple wrapper on `numroc()`, to determine if the process actually owns any data from the global matrix. If you consider this an extravagant waste of memory, then I have some very interesting things to tell you about R.

Likewise, the slot `@ldim` is the dimension of the local storage, and not, necessarily, the "actual" local dimension (which could effectively be NULL). So for example, if you imagine all data for a distributed matrix living on process 0, then all the other processes should store `matrix(0)` in the `@Data` slot, and `c(1,1)` in the `@ldim` slot.

Let's take a look at an example. Suppose we want to develop a way of taking logs of the entries of our distributed matrix in a way that is a natural extension to that of R (this is actually already done in **pbdDMAT**, but is a good illustration). A quick call to `isGeneric(f="log")` shows that the function `log()` in R is already S4 generic, so we can easily enough define a new method for it.

For some problems, some processors will own `matrix(0)` under the `@Data` slot when really, technically, they store nothing. We don't want to take the log of these zeros, since there is no real point. Worse, failure to exclude these values across several methods can accumulate in incorrect answers (think of taking the log on all submatrices and then running a `sum()` method on a distributed matrix). Using `ownany()`, this is trivial.

Generating in Parallel

```
mylog <- function(x, base=exp(1))
{
  if (ownany(dim=x@dim, bldim=x@bldim, CTXT=x@CTXT))
    x@Data <- log(x=x@Data, base=base)
  return(x)
}
```

Now we just need to set the method:

Generating in Parallel

```
setMethod("log", signature(x="ddmatrix"),
  mylog
)
```

This is more or less how things are done in **pbdDMAT**.

8.2. BLACS

The most critical piece of information to impart is that no matter what, "block-cyclicality" can not be destroyed. Most of the routines for distributed matrices (and almost all of the really complicated ones) assume this structure. This is why 3 BLACS contexts are created by default when initializing the process grid; namely, context 0 is the optimal (unless otherwise

specified) rectangular process grid, context 1 is the $PQ \times 1$ process grid (assuming P process rows and Q process columns), and context 2 is the $1 \times PQ$ process grid. As defined above, context 0 is optimal for many linear algebra routines, and not really any worse (aside from being cumbersome) for most other computations. The other contexts are important because they are more natural for adding/removing columns and rows (respectively).

Redistributing the data from a $P \times Q$ process grid to a $1 \times PQ$ process grid can be very useful, despite the communication overhead involved. When a matrix is distributed across this context, while it is not trivial to "drop" rows (certainly not as easy as it is in context 1), doing so, whichever rows we do indeed drop, results in a block-cyclically distributed matrix. In general, the same cannot be said for other contexts. The `ddmatrix` methods `[` and `na.exclude()` rely on such a redistribution (two redistributions in the case of the former).

References

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/.

Chambers J (2006). "How S4 Methods Work." *Technical report*, The R-Project for Statistical Computing. URL http://developer.r-project.org/howMethodsWork.pdf.

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.

Dongarra J, Whaley RC (1995). "A User's Guide to the BLACS." *Technical report*, University of Tennessee. UT-CS-95-281.

Genolini C (2008). "A (Not So) Short Introduction to S4." *Technical report*, The R-Project for Statistical Computing. URL http://cran.r-project.org/doc/contrib/Genolini-S4tutorialV0-5en.pdf.

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

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

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

Schmidt D, Chen WC, Ostrouchov G, Patel P (2012b). "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, Ostrouchov G, Chen WC, Patel P (2012d). "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.