# Guide to the **pbdBASE** Package

ScaLAPACK Wrappers and Helpers

February 13, 2014

Drew Schmidt
*National Institute for Computational Sciences*
*University of Tennessee*

Wei-Chen Chen
*Department of Ecology and Evolutionary Biology*
*University of Tennessee*

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

Pragneshkumar Patel
*National Institute for Computational Sciences*
*University of Tennessee*

Version 0.3-0

## 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 was supported in part by the Department of Ecology and Evolutionary Biology at the University of Tennessee, Knoxville, and a grant from the National Science Foundation (MCB-1120370.)

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.

## Disclaimer

The findings and conclusions in this article have been formally disseminated neither by the U.S. Department of Energy, nor by the University of Tennessee, and as such should not be construed to represent any determination or policy of any University, Agency, and/or National Laboratory.



This publication was typeset using LaTeX.

# Contents

**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 **pbdBASE** package for R provides a set of BLACS, PBLAS, and ScaLAPACK wrappers, as well as numerous new functionality in the block-cyclic matrix paradigm. 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 (**?**), abbreviated pbdR or just pbd, is a project which seeks to elevate the R language to supercomputers. This package, **pbdBASE** (**?**), contains a set of wrappers of the high performance libraries BLACS, PBLAS, and ScaLAPACK (**?**), and also a host of new subroutines for performing distributed matrix computations in R. The package is a dependency of **pbdDMAT** (**?**), which is meant to greatly simplify the **pbdBASE** system into something that intimately resembles the R language. Since these two packages ultimately rely on the ScaLAPACK library, the data type used with each is the block-cyclic distributed matrix. See the **pbdDMAT** vignette for more details.

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 Installation

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

Installing pbdBASE

```
1 install.packages("pbdBASE")
```

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

## 1.2 Intended Audience

The **pbdBASE** package is a dependency of **pbdDMAT**, and so anyone who wishes to use the latter package must first install **pbdBASE**. However, much of the direct use of **pbdBASE** is in-

tended only for extremely advanced users and developers. A few exceptions are the `init.grid()` and `finalize()` functions, which will be outlined in the sections to follow. The overwhelming majority of the remaining functions are either internal or for people deeply familiar with ScaLAPACK.

### 1.3 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 (**?**) 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."

## 2 Using pbdBASE

### 2.1 BLACS Communicators

Briefly, distributed matrix computations using ScaLAPACK require specialized MPI communicators, via the BLACS library. As with any MPI communicator, you must initialize it before getting started with communications, and you must terminate it when you are finished with communications. For most users, this will amount to calling

```
library(pbdBASE, quiet = TRUE)
init.grid() # initialize

# ...

finalize() # terminate
```

This special communicator may be used with **pbdMPI** communicator(s) without causing problems, and by default one `finalize()` call will terminate all communicators, whether they be from **pbdMPI** or **pbdBASE** (see the **pbdBASE** reference manual for more details and options).

#### 2.1.1 Construction

BLACS communicators are not identical to **pbdMPI** communicators. Indeed, while a **pbdMPI** communicator is a one-dimensional array of processors, BLACS communicators are two-

dimensional (row-major) grids. These values are simply referred to as the number of processor rows and the number of processor columns, as a communicator really is thought of as a matrix of processors. When a grid is initialized with `init.grid()` and no arguments are passed, then three communicators are created. These grids are referenced by their "integer context" value, or `ICTXT`. These grids are numbered 0, 1, and 2. Context 0 tries to be the "best possible" context (see (**?**)). Here we make 2 choices:

1. Grids are always as close to square as possible.
2. In the event a grid can not be made to be square, the larger value is used for the number of processor rows.

So for example, if we have 4 processors, then by default this would create a $2 \times 2$ grid for context 0. However, if we have 6 processors, then by default this will create a $3 \times 2$ grid of processors.

On the other hand, context 1 is always a $1 \times n$ grid, where $n$ is the total number of processors. Likewise, context 2 is always a $n \times 1$ grid of processors. These can be extremely valuable, especially for performing data movement operations.

The function `init.grid()` does a great deal of (useful) hand-holding, so the much more advanced user who is familiar with BLACS may be more interested in the function `blacs_gridinit()`, which does not reserve contexts 0, 1, or 2. However, many **pbdDMAT** functions make assumptions about the existence and shapes of contexts 0, 1, and 2 (as described above), so this functionality is not supported when using that package.

### 2.1.2 Destruction

The user can halt all communicators — both BLACS communicators and, optionally, those created by **pbdMPI** (or others) — by calling `finalize()`. To destroy just a single BLACS context (for example, one used to read in data on a subset of processors), then the user should use `gridexit()`. See the **pbdBASE** reference manual for full details.

## 2.2 Notes for Developers

The **pbdBASE** package also has several useful routines for package developers who need to deal with distributed matrices (such as **pbdDMAT**'s `ddmatrix` object). Chief among these is the `numroc()` function. Here, `numroc` stands for **num**ber of **r**ows **o**r **c**olumns. This routine is used for determining local storage dimensions. If you need to construct a distributed matrix and know its (global) dimension, blocking factor, and BLACS context, then you can determine the local problem size by making the call:

```
1 numroc(dim=dim, bldim=bldim, ICTXT=ICTXT)
```

This will return a numeric pair of values, with the first being the number of rows of the local matrix, and the second being the number of columns in the local matrix. No communication is performed with this call. However, it is possible that the above can return seemingly nonsensical values. For example, if a processor owns no piece of the global matrix, then the local dimension

information returned from `numroc()` could be less than 1 in some dimension (rows, columns, or both). By default, this should not happen because of an automatic correction, with the smallest return possible being 1. To allow for the aforementioned possibility, pass the additional argument `fixme=FALSE`.

We always make the convention that every processor owns *something*, even if one does not actually own any portion of the global matrix. The default in this even is a 1 row, 1 column matrix consisting of the single entry 0.0. This convention is to prevent problems when passing off data to compiled code (C and Fortran), and care should be taken to preserve this. As such, the reader may wish to exclusively use `numroc()` for its intended purpose (with correction), but may still need to know about the case when the local storage is "in name only." For this, use the `ownany()` routine, which answers the question "does the calling processor own any of the global matrix?" with a `TRUE` ("yes") or `FALSE` ("no"). The call is virtually identical to `numroc()`:

```
1 ownany(dim=dim, bldim=bldim, ICTXT=ICTXT)
```

See the **pbdBASE** reference manual for full details.

## References