A Quick Guide for the pbdMPI Package (Ver. 0.2-2)

Wei-Chen Chen[1], **George Ostrouchov**[1,2], **Drew Schmidt**[2], **Pragneshkumar Patel**[2], **Hao Yu**[3]

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

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

[3]University of Western Ontario,
London, Ontario, Canada

Contents

Acknowledgement

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. 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.

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

We thank our colleagues from the Scientific Data Group, Computer Science and Mathematics Division, Oak Ridge National Laboratory (ORNL), Hasan Abbasi, Jong Youl Choi, Scott Klasky, and Nobert Podhorszki for discussing windows MPI systems, compiler issues, dynamic libraries, and generally improving our knowledge of MPI performance issues.

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

Warning: 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 **pbdMPI** (Chen *et al.* 2012), version 0.2-2. 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/` (Ostrouchov *et al.* 2012).

1. Introduction

Our intent is to bring the most common parallel programming model from supercomputing, Single Program Multiple Data (SPMD), to R and enable distributed handling of truly large data. Consequently, **pbdMPI** is intended for batch mode programming with big data (pbd). Unlike **Rmpi** (Yu 2002), **snow** (Tierney *et al.* 2012), or **parallel** (R Core Team 2012), interactive mode is not supported. We think that interaction with a large distributed parallel computing platform is better handled with a client/server relationship, and we are developing other packages in this direction. **pbdMPI** simplifies MPI interaction, but leaves low and mid level functions available for advanced programmers. For example, it is easy to hand communicators to **pbdMPI** from other applications through MPI array pointers. This is intended for integration with other, possibly non-R, parallel software.

Under the SPMD parallel programming model, the identical program runs on every processor but typically works on different parts of a large data set, while communicating with other copies of itself as needed. Differences in execution stem from `comm.rank`, which is typically different on every processor. While on the surface this sounds complicated, after some experience and a new mindset, programming is surprisingly simple. There is no master. There is only cooperation among the workers. Although we target very large distributed computing platforms, SPMD works well even on small multicore platforms.

In the following, we list the main features of **pbdMPI**.

1. Under the SPMD batch programming model, a single program is written, which is spawned by `mpirun`. No spawning and broadcasting from within R are required.

2. S4 methods are used for most collective functions so it is easy to extend them for general R objects.

3. Default methods (like `Robj` functions in **Rmpi**) have homogeneous checking for data type so they are safe for general users.

4. The API in all functions is simplified, with all default arguments in control objects.

5. Methods for array or matrix types are implemented without serialization and un-serialization, resulting in faster communication than **Rmpi**.

6. Basic data types of integer, double and raw in **pbdMPI** are communicated without further checking. This is risky but fast for advanced programmers.

7. Character data type is serialized and communicated by raw type.

System requirements and installation of **pbdMPI** are described next. Section 2 gives a short example for comparing performance of **pbdMPI** and **Rmpi** (Yu 2002). In Section 3, a few quick answers for questions are given. Section 4 provides settings for Windows environments. Finally, in Section 5, two examples from **parallel** are shown as SPMD **pbdMPI** programs.

1.1. System Requirements

pbdMPI requires MPI (http://en.wikipedia.org/wiki/Message_Passing_Interface). The package is mainly developed and tested under OpenMPI (http://www.open-mpi.org/) in xubuntu 11.04 (http://xubuntu.org/) and should also work with LAM/MPI (http://www.lam-mpi.org/) and MPICH2 (http://www.mcs.anl.gov/research/projects/mpich2/). In addition to unix, **pbdMPI** should also run under other operating systems such as Mac OS X with OpenMPI or Windows 7 with MPICH2 if MPI is installed and launched properly, although we have not tested on multiple machines yet. Please let us know about your experience.

For normal installation, see Sec. 1.2. To build as a static library, which may be required on some large systems, use

Shell Command

```
./configure --enable-static --prefix=${MPI_ROOT}
make
make install
```

where `--enable-static` can build a static library (optional), and `${MPI_ROOT}` is the path to MPI root. Note that the static library is not necessary for **pbdMPI** but may avoid dynamic loading problems.

To make sure your MPI system is working, test with

Shell Command

```
mpiexec -np 2 hostname
```

This should list two host names where MPI jobs are running. Note to use `hostname.exe` with the extension on a Windows system.

1.2. Installation and Quick Start

One can download **pbdMPI** from CRAN at http://cran.r-project.org, and the intallation can be done with the following commands (using OpenMPI library)

Shell Command

```
tar zxvf pbdMPI_0.1-0.tar.gz
R CMD INSTALL pbdMPI
```

Further configure arguments include

Argument	Default
`--with-mpi-type`	`OPENMPI`
`--with-mpi-include`	`${MPI_ROOT}/include`
`--with-mpi-libpath`	`${MPI_ROOT}/lib`
`--with-mpi`	`${MPI_ROOT}`

where `${MPI_ROOT}` is the path to the MPI root. For non-default and unusual installations of MPI systems, the commands may be

Shell Command

```
### Under command mode
R CMD INSTALL pbdMPI \
  --configure-args="--with-mpi-type=OPENMPI \
                    --with-mpi=/usr/local"
R CMD INSTALL pbdMPI \
  --configure-args="--with-mpi-type=OPENMPI \
                    --with-mpi-include=/usr/local/ompi/include \
                    --with-mpi-libpath=/usr/local/ompi/lib"
```

See the package source file `pbdMPI/configure` for details.

One can get started quickly with **pbdMPI** by learning from the following six examples.

Shell Command

```
### At the shell prompt, run the demo with 2 processors by
### (Use Rscript.exe for windows system)
mpiexec -np 2 Rscript -e "demo(allgather,'pbdMPI',ask=F,echo=F)"
mpiexec -np 2 Rscript -e "demo(allreduce,'pbdMPI',ask=F,echo=F)"
mpiexec -np 2 Rscript -e "demo(bcast,'pbdMPI',ask=F,echo=F)"
mpiexec -np 2 Rscript -e "demo(gather,'pbdMPI',ask=F,echo=F)"
mpiexec -np 2 Rscript -e "demo(reduce,'pbdMPI',ask = F,echo=F)"
mpiexec -np 2 Rscript -e "demo(scatter,'pbdMPI',ask=F,echo=F)"
```

1.3. Basic Steps

In the SPMD world, every processor is a worker, every worker knows about all the others, and each worker does its own job, possibly communicating with the others. Unlike the manager/workers style, SPMD is more likely to fully use the computer resources. The following shows typical basic steps of using **pbdMPI**.

1. Initialize. (`init`)
2. Read your portion of the data.
3. Compute. (`send`, `recv`, `barrier`, ...)
4. Communicate results among workers. (`gather`, `allgather`, `reduce`, `allreduce`, ...)
5. Finalize. (`finalize`)

In a given application, the Compute and Communicate steps may be repeated several times for intermediate results. The Compute and Communicate steps are more general than the "map" and "reduce" steps of the map-reduce paradigm but similar in spirit. One big difference is that the Communicate step may place the "reductions" on all processors rather than just one (the manager for map-reduce) for roughly the same time cost. With some experience, one can easily convert existing R scripts, and quickly parallelize serial code. **pbdMPI** tends to reduce programming effort, avoid complicated MPI techniques, and gain computing performance.

The major communication functions of **pbdMPI** and corresponding similar functions of **Rmpi** are listed in the following.

pbdMPI (S4)	**Rmpi**
`allgather`	`mpi.allgather`, `mpi.allgatherv`, `mpi.allgather.Robj`
`allreduce`	`mpi.allreduce`
`bcast`	`mpi.bcast`, `mpi.bcast.Robj`
`gather`	`mpi.gather`, `mpi.gatherv`, `mpi.gather.Robj`
`recv`	`mpi.recv`, `mpi.recv.Robj`
`reduce`	`mpi.reduce`
`scatter`	`mpi.scatter`, `mpi.scatterv`, `mpi.scatter.Robj`
`send`	`mpi.send`, `mpi.send.Robj`

1.4. More Examples

The package source files provide several examples based on **pbdMPI**, such as

Directory	Examples
`pbdMPI/inst/examples/test_spmd/`	main SPMD functions
`pbdMPI/inst/examples/test_rmpi/`	comparison to **Rmpi**
`pbdMPI/inst/examples/test_parallel/`	comparison to **parallel**
`pbdMPI/inst/examples/test_performance/`	performance testing
`pbdMPI/inst/examples/test_s4/`	S4 extension
`pbdMPI/inst/examples/test_cs/`	client/server examples
`pbdMPI/inst/examples/test_long_vector/`	long vector examples

where `test_long_vector/` requires to recompile with setting

pkg_constant.h

```
#define MPI_LONG_DEBUG 1
```

in `pbdMPI/src/pkg_constant.h`. See Scetion 6 for details.

Further examples can be found at including:

- "Introduction to distributed computing with pbdR at the UMBC High Performance Com puting Facility (Technical Report, 2013)." (Raim 2013)

2. Performance

There are more examples for testing performance in `pbdMPI/inst/examples/test_rmpi`. Here, we only show a simple comparison of **pbdMPI** to **Rmpi**. The two scripts are equivalent for **pbdMPI** and **Rmpi**. We run them with two processors and obtain computing times listed below.

Save the following script in `demo_spmd.r` and run it with two processors by

Shell Command

```
mpiexec -np 2 Rscript demo_spmd.r
```

to see the computing time on your platform.

pbdMPI R Script

```
### Save this script in "demo_spmd.r".
library(pbdMPI, quiet = TRUE)
init()

time.proc <- list()
time.proc$default <- system.time({
  for(i in 1:1000) y <- allgather(list(x = 1:10000))
  barrier()
})

time.proc$matrix <- system.time({
  for(i in 1:1000) y <- allgather(matrix(1:10000, nrow = 100))
  barrier()
})

comm.print(time.proc, quiet = TRUE)
finalize()
```

Save the following script in `demo_rmpi.r` and run with two processors by

Shell Command

```
mpiexec -np 2 Rscript demo_rmpi.r
```

to see the computing time on your platform.

Rmpi R Script

```
### Save this script in "demo_rmpi.r".
library(Rmpi)
invisible(mpi.comm.dup(0, 1))

time.proc <- list()
time.proc$Robj <- system.time({
  for(i in 1:1000) y <- mpi.allgather.Robj(list(x = 1:10000))
  mpi.barrier()
})

time.proc$matrix <- system.time({
```

```
  for(i in 1:1000) y <- mpi.allgather.Robj(matrix(1:10000, nrow =
     100))
  mpi.barrier()
})

if(mpi.comm.rank(1) == 0) print(time.proc)
mpi.quit()
```

The following shows the computing time of the above two scripts on a single machine with two processors Intel(R) Core(TM) i5-2410M CPU @ 2.30 GHz, xubuntu 11.04 system, and OpenMPI 1.6. The **pbdMPI** is more efficient than **Rmpi** with `list` and `matrix/array` data structures.

R Output

```
>> Output from demo_spmd.r
$default
   user  system elapsed
  1.680   0.030   1.706

$matrix
   user  system elapsed
  0.950   0.000   0.953

>> Output from demo_rmpi.r
$Robj
   user  system elapsed
  2.960   0.090   3.041

$matrix
   user  system elapsed
  3.120   0.030   3.147
```

3. FAQs

3.1. General

1. **Q:** Do I need MPI knowledge to run **pbdMPI**?
   A: Yes, but only the big picture, not the details. We provide several examples in `pbdMPI/inst/examples/test_spmd/` to introduce essential methods for learning MPI communication.

2. **Q:** Can I run **pbdMPI** on my laptop locally?
   A: Sure, as long as you have an MPI system. You even can run it on 1 CPU.

3. **Q:** Does **pbdMPI** support Windows clusters?
   A: Yes, the released binary currently supports OpenMPI and MPICH2. For other MPI systems, users have to compile from source.

4. **Q:** Can I run **pbdMPI** in OpenMPI and MPICH2 together?
   A: No, you can have both OpenMPI and MPICH2 installed in your OS, but you are only allowed to run **pbdMPI** with one MPI system. Just pick one.

5. **Q:** Does **pbdMPI** support any interactive mode?
   A: No, but yes. **pbdMPI** only considers batch execution and aims for programming with big data that do not fit on desktop platforms. We think that interaction with big data on a big machine is better handled with a client/server interface, where the server runs SPMD codes on big data and the client operates with reduced data representations.

   If you really need an interactive mode, such as for debugging, you can utilize **pbdMPI** scripts inside **Rmpi**. **Rmpi** mainly focuses on Manager/Workers computing environments, but can run SPMD codes on workers only with a few adjustments. See the "Programming with Big Data in R" website for details at `http://r-pbd.org/`.

   Note that **pbdMPI** uses communicators different from **Rmpi**. Be sure to free the memory correctly for both packages before quitting. `finalize(mpi.finalize = FALSE)` can free the memory allocated by **pbdMPI**, but does not terminate MPI before calling `mpi.quit` of **Rmpi**.

6. **Q:** Can I write my own collective functions for my own data type?
   A: Yes, S4 methods allow users to add their own data type, and functions. Quick examples can be found in `pbdMPI/inst/examples/test_s4/`.

7. **Q:** Does **pbdMPI** support long vector or 64-bit integer? **A:** See Section 6.

3.2. Programming

1. **Q:** Can I run task jobs by using **pbdMPI**?
   A: Yes, it is relatively straightforward for parallel tasks. Neither extra automatic functions nor further command/data communication is required. In other words, SPMD is easier for Monte Carlo, bootstrap, MCMC simulation and statistical analysis for ultra-large datasets. A more efficient way, such as task pull parallelism, can be found in next Q&A.

   Example 1:

SPMD R Script

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

id <- get.jid(total.tasks)

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

### or using apply-like functions.
lapply(id, function(i){
```

```
    # put independent task i script here.
  })

  finalize()
```

Note that `id` gets different values on different processors, accomplishing `total.tasks` across all processors. Also note that any data and partial results are not shared across the processors unless communicated.

Example 2:

SPMD R Script

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

  ### Directly using a loop
  for(i in 1:total.tasks){
    if(i %% comm.size() == comm.rank()){
    # put independent task i script here
  }
}

### or using apply-like function.
lapply(1:total.tasks, function(i){
  if(i %% comm.size() == comm.rank()){
  # put independent task i script here.
}
})

finalize()
```

2. **Q:** Can I run un-barrier task jobs, such as task pull parallelism, by using **pbdMPI**?
   A: Yes, it is relatively straightforward via **pbdMPI** API function `task.pull()` in SPMD. For example, the next is available in demo which has a user defined function `FUN()` run on workers, and master (rank 0) controls the task management.

Shell Command

```
mpiexec -np 4 Rscript -e "demo(task_pull,'pbdMPI',ask=F,echo=F)"
```

SPMD R Script (task_pull)

```
### Initial
library(pbdMPI, quiet = TRUE)

### Examples
FUN <- function(jid){
  Sys.sleep(1)
  jid * 10
}

ret <- task.pull(1:10, FUN)
```

```
comm.print(ret)

if(comm.rank() == 0){
  ret.jobs <- unlist(ret)
  ret.jobs <- ret.jobs[names(ret.jobs) == "ret"]
  print(ret.jobs)
}

### Finish
finalize()
```

3. **Q:** What if I want to run task push or pull by using **pbdMPI**?
   A: No problem. As in the two proceeding examples, task push or pull can be done in the same way by using rank 0 as the manager and the other ranks as workers. However, we do not recommend it except perhaps for inhomogeneous computing environments and independent jobs.

4. **Q:** Are S4 methods more efficient?
   A: Yes and No. S4 methods are a little less efficient than using `switch ... case ...` in C, but most default methods use `raw` with `un-` and `serialize` which may cost `3-10` times more than using `integer` or `double`. Instead of writing C code, it is easier to take advantage of S4 methods to extend to general R objects (`matrix`, `array`, `list`, `data.frame`, and `class` ...) by communicating with basic data types (`integer` and `double`) and avoiding serialization.

5. **Q:** Can I disable the MPI initialization of **pbdMPI** when I call `library(pbdMPI)`?
   A: Yes, you can set a hidden variable `.__DISABLE_MPI_INIT__` in the `.GlobalEnv` before calling `library(pbdMPI)`. For example,

   SPMD R Script

   ```
   assign(".__DISABLE_MPI_INIT__", TRUE, envir = .GlobalEnv)
   library(pbdMPI)
   ls(all.names = TRUE)
   init()
   ls(all.names = TRUE)
   finalize(mpi.finalize = FALSE)
   ```

   Note that we are *NOT* supposed to kill MPI in the `finalize` step if MPI is initialized by external applications. But some memory allocated by **pbdMPI** has to be free, `mpi.finalize = FALSE` is set above.

   To avoid some initialization issues of MPI, **pbdMPI** uses a different way than **Rmpi**. **pbdMPI** allows you to disable initializing communicators when loading the library, and later on you can call `init` to initialize or obtain communicators through `.__MPI_APTS__` as in the next question.

6. **Q:** Can **pbdMPI** take or export communicators?
   A: Yes, the physical memory address is set to the variable `.__MPI_APTS__` in the `.GlobalEnv` through a call to `init()`. The variable points to a structure containing MPI structure arrays preallocated while **pbdMPI** is loaded. `pbdMPI/src/pkg_*` provides a mechanism to take or export external/global variables at the C language level.

3.3. MPI Errors

1. **Q:** If compilation successful, but load fails with segfault

Error Message

```
** testing if installed package can be loaded
sh: line 1:  2905 Segmentation fault
'/usr/local/R/3.0.0/intel13/lib64/R/bin/R' --no-save --slave
   2>&1 <
/tmp/RtmpGkncGK/file1e541c57190
ERROR: loading failed

*** caught segfault ***
address (nil), cause 'unknown'
```

A: Basically, **pbdMPI** and all pbdR are tested and have stable configuration in GNU environment. However, other compilers are also possible such as Intel compiler. This message may come from the system of login node does not have a MPI system, MPI system is only allowed to be loaded in computing node, or MPI shared library is not loaded correctly and known to R. The solution is to use extra flag to `R CMD INSTALL -no-test-load pbdMPI*.tar.gz`, and use `export LD_PRELOAD=...` as the answer to the next question.

2. **Q:** If installation fails with

Error Message

```
Error in dyn.load(file, DLLpath = DLLpath, ...) :
  unable to load shared object '/.../pbdMPI/libs/pbdMPI.so':
  libmpi.so: cannot open shared object file: No such file or
     directory
```

A: OpenMPI may not be installed in the usual location, so the environment variable `LD_LIBRARY_PATH` should be set to the `libmpi.so` path, such as

Shell Command

```
export LD_LIBRARY_PATH=/usr/local/openmpi/lib:$LD_LIBRARY_PATH
```

where `/usr/local/openmpi/lib` should be replaced by the path to `libmpi.so`. Or, use `export LD_PRELOAD=...` to preload the MPI library if the library name is not conventional, such as

Shell Command

```
export LD_PRELOAD=/usr/local/openmpi/lib/libmpi.so:$LD_PRELOAD
```

Another solution may be to use the unix command `ldconfig` to setup the correct path.

3. **Q:** **pbdMPI** installs successfuly, but fails at initialization when calling the function `init()` with error message

Error Message

```
/usr/lib/R/bin/exec/R: symbol lookup error:
/usr/lib/openmpi/lib/openmpi/mca_paffinity_linux.so: undefined
    symbol:
mca_base_param_reg_int
```

A: The linked library at installation may be different from the runtime library, especially when your system has more than one MPI systems. Since the library at installation is detected by `autoconf` (`configure`) and `automake` (`Makevars`), it can be linked with OpenMPI library, but MPICH2 or LAM/MPI is searched before OpenMPI according to `$PATH`.

Solutions:

- Check which MPI system is your favorite to call. If you use OpenMPI, then you have to link with OpenMPI. Similarly, for MPICH2.
- Or, only kepp the MPI system you do like and drop others.
- Use `--with-mpi-type` to specify the MPI type.
- Use `--with-mpi-include` and `--with-mpi-libpath` to specify the right version.

4. **Q:** (Linux) If OpenMPI `mpiexec` fails with

Error Message

```
mca: base: component_find: unable to open
/.../openmpi/lib/openmpi/mca_paffinity_hwloc:
/.../openmpi/lib/openmpi/mca_paffinity_hwloc.so:
undefined symbol: opal_hwloc_topology (ignored)
...
mca: base: component_find: unable to open
/.../openmpi/lib/openmpi/mca_carto_auto_detect:
/.../openmpi/lib/openmpi/mca_carto_auto_detect.so:
undefined symbol: opal_carto_base_graph_get_host_graph_fn
    (ignored)
...
```

A: The linked MPI library `libmpi.so` may be missing or have a different name. OpenMPI builds shared/dynamic libraries by default and the target file `libmpi.so` is used by `pbdMPI/src/spmd.c` through `#include <dlfcn.h>` and `dlopen(...)` in the file `pbdMPI/src/pkg_dl.c`.

Solutions:

- Check if the path and version of `libmpi.so` are correct. In particular, one may have different MPI systems installed.
- When linking with `libmpi.so` in OpenMPI, one must run/load **pbdMPI** with OpenMPI's `libmpi.so`. The same for LAM/MPI and MPICH2.
- Use `export LD_PRELOAD=$PATH_TO_libmpi.so.*` in command mode.
- Use the file `/etc/ld.so.conf` and the command `ldconfig` to manage personal MPI installation.

- Or, recompile OpenMPI with a static library, and use `libmpi.a` instead.

5. **Q:** (Windows) If OpenMPI `mpiexec` fails with

Error Message

```
ORTE_ERROR_LOG: Error in file ..\..\..\openmpi-1
.6\orte\mca\ess\hnp\ess_hnp_module.c at line 194
...
ORTE_ERROR_LOG: Error in file ..\..\..\openmpi-1
.6\orte\runtime\orte_init.c at line 128
...
```

A: Check if the network is unplugged, the network should be "ON" even on a single machine. At least, the status of network interface should be correct.

6. **Q:** For MPICH2 users, if installation fails with

Error Message

```
/usr/bin/ld: libmpich.a(comm_get_attr.o): relocation R_X86_64_32
against `MPIR_ThreadInfo' can not be used when making a shared
object; recompile with -fPIC
libmpich.a: could not read symbols: Bad value
collect2: ld returned 1 exit status
```

A: MPICH2 by default does not install a shared library which means `libmpich.so` is missing and **pbdMPI** trys to link with a static library `libmpich.a` instead. Try to recompile MPICH2 with a flag `-enable-shared` and reinstall **pbdMPI** again.

7. **Q:** For MPICH2 and MPICH3 users, if installation fails with

Error Message

```
/usr/bin/ld: cannot find -lopa
collect2: error: ld returned 1 exit status
make: *** [pbdMPI.so] Error 1
ERROR: compilation failed for package 'pbdMPI'
```

A: By default, `-lopa` is required for some systems. However, some systems may not have it and can be disable with a configuration flag when install **pbdMPI**, such as `R CMD INSTALL pbdMPI*.tar.gz -configure-args="-disable-opa"`.

3.4. Other Errors

1. **Q:** **pbdMPI** is linked with **pbdPROF** (Chen *et al.* 2013) and **mpiP** (Vetter and McCracken 2001). (i.e. `-enable-pbdPROF` is used in **pbdMPI** and `-with-mpiP` is used in **pbdPROF**.) If **pbdMPI** compilation successful, but load fails with

Error Message

```
Error : .onLoad failed in loadNamespace() for 'pbdMPI', details:
```

```
  call: dyn.load(file, DLLpath = DLLpath, ...)
  error: unable to load shared object 'pbdMPI.so':
  pbdMPI/libs/pbdMPI.so: undefined symbol: _Ux86_64_getcontext
```

A: Some prerequisite packages by **mpiP** is installed correctly. Reinstall **mpiP** by

R Script

```
./configure --disable-libunwind CPPFLAGS="-fPIC
   -I/usr/lib/openmpi/include" LDFLAGS="-L/usr/lib/openmpi/lib
   -lmpi"
```

and followed by reinstall **pbdPROF** and **pbdMPI**.

4. Windows Systems

Currently, **pbdMPI** supports Windows with MPICH2 (http://www.mcs.anl.gov/research/projects/mpich2/). The binary installations of both MPI systems are available from the website. `mpich2-1.4.1p1-win-ia32.msi` is for 32-bits and `mpich2-1.4.1p1-win-x86-64.msi` is for 64-bits. The installation is easily done with a few clicks. The default environment and path are recommended.

For running MPI and R, users need to set `PATH` to the `mpiexec.exe` and `Rscript.exe`. By default,

Shell Command

```
### Under command mode, or save in a batch file.
SET R_HOME=C:\Program Files\R\R-2.15.1
SET MPI_ROOT=C:\Program Files\MPICH2
SET PATH=%MPI_ROOT%\bin\;%R_HOME%\bin\;%PATH%
```

is for the 64-bit MPICH2, but replace

Shell Command

```
SET MPI_ROOT=C:\Program Files (x86)\MPICH2
```

for the 32-bit MPICH2.

4.1. Install from Binary

The binary packages of **pbdMPI** are available on the website: "Programming with Big Data in R" at http://r-pbd.org/. Note that different MPI systems require different binaries. The binary can be installed by

Shell Command

```
R CMD INSTALL pbdMPI_0.1-0.zip
```

As on Unix systems, one can start quickly with **pbdMPI** by learning from the following demos. There are six basic examples.

Shell Command

```
### Run the demo with 2 processors by
mpiexec -np 2 Rscript.exe -e "demo(allgather,'pbdMPI',ask=F,echo=F)"
mpiexec -np 2 Rscript.exe -e "demo(allreduce,'pbdMPI',ask=F,echo=F)"
mpiexec -np 2 Rscript.exe -e "demo(bcast,'pbdMPI',ask=F,echo=F)"
mpiexec -np 2 Rscript.exe -e "demo(gather,'pbdMPI',ask=F,echo=F)"
mpiexec -np 2 Rscript.exe -e "demo(reduce,'pbdMPI',ask=F,echo=F)"
mpiexec -np 2 Rscript.exe -e "demo(scatter,'pbdMPI',ask=F,echo=F)"
```

Warning: Note that spacing inside `demo` is not working for Windows systems and `Rscript.exe` should be evoked rather than `Rscript`.

4.2. Build from Source

Warning: This section is only for building binary in 32- and 64-bit Windows system. A more general way can be found in the file `pbdMPI/INSTALL.win`.

Make sure that R, **Rtools**, and **MINGW** are in the `PATH`. See details on the website "Building R for Windows" at http://cran.r-project.org/bin/windows/Rtools/. But, if both 32- and 64-bits MPICH2 are installed, two different environment variables `MPI_ROOT_32` and `MPI_ROOT_64` need to be set for building binaries.

For example, the minimum requirement may be

Shell Command

```
### Under command mode, or save in a batch file.
SET R_HOME=C:\Program Files\R\R-2.15.1
SET RTOOLS=C:\Rtools\bin\
SET MINGW=C:\Rtools\gcc-4.6.3\bin
SET PATH=%R_HOME%;%R_HOME%\BIN\;%RTOOLS%;%MINGW%;%PATH%
SET MPI_ROOT_64=C:\Program Files\MPICH2
SET MPI_ROOT_32=C:\Program Files (x86)\MPICH2
```

With a correct `PATH`, one can use the R commands to install/build the **pbdMPI**:

Shell Command

```
### Under command mode, build and install the binary.
tar zxvf pbdMPI_0.1-0.tar.gz
R CMD INSTALL --build pbdMPI
R CMD INSTALL pbdMPI_0.1-0.zip
```

5. SPMD in Examples from package parallel

We demonstrate how a simple script from **parallel** can be written in batch by using **pbdMPI**. Each time, we first give the version using **parallel** followed by the version using **pbdMPI**. All codes are available in `pbdMPI/inst/examples/test_parallel/`.

Example 1: (`mclapply()` originates in **multicore** (Urbanek 2011))
Save the following script in a file and run with

Shell Command

```
Rscript 01_mclapply_par.r
```

to see the computing time on your platform.

multicore R Script

```
### File Name: 01_mclapply_par.r
library(parallel)

system.time(
  unlist(mclapply(1:32, function(x) sum(rnorm(1e7))))
)
```

Now save this script in a file and run with

Shell Command

```
mpirun -np 2 Rscript 01_mclapply_spmd.r
```

to see the computing time on your platform.

SPMD R Script

```
### File Name: 01_mclapply_spmd.r
library(pbdMPI, quiet = TRUE)
init()

time.proc <- system.time({
  id <- get.jid(32)
  ret <- unlist(lapply(id, function(i) sum(rnorm(1e7))))
  ret <- allgather(ret, unlist = TRUE)
})
comm.print(time.proc)

finalize()
```

The following shows the computing time of the above codes on a single local machine with two cores Intel(R) Core(TM) i5-2410M CPU @ 2.30 GHz, xubuntu 11.04 system, and OpenMPI 1.6. There is not much communication latency in this example since all computings are on one "node" which is also a limitation of **parallel**.

R Output

```
>> Test ./01_mclapply_par.r
   user  system elapsed
 16.800   0.570  17.419

>> Test ./01_mclapply_spmd.r
COMM.RANK = 0
   user  system elapsed
 17.130   0.460  17.583
```

Example 2: (`parMM()` originates in **snow** (Tierney *et al.* 2012))
Save the following code in a file and run with two processors

Shell Command

```
Rscript 02_parMM_par.r
```

to see the computing time on your platform.

snow R Script

```
### File Name: 02_parMM_par.r
library(parallel)
cl <- makeCluster(2)

splitRows <- function (x, ncl){
  lapply(splitIndices(nrow(x), ncl), function(i) x[i, , drop = FALSE])
}
parMM <- function (cl, A, B){
  do.call(rbind, clusterApply(cl, splitRows(A, length(cl)),
      get("%*%"), B))
}

set.seed(123)
A <- matrix(rnorm(1000000), 1000)
system.time(replicate(10, A %*% A))
system.time(replicate(10, parMM(cl, A, A)))

stopCluster(cl)
```

Now save this script in a file and run with

Shell Command

```
mpirun -np 2 Rscript 02_parMM_spmd.r
```

to see the computing time on your platform.

SPMD R Script

```
### File Name: 02_parMM_spmd.r
library(pbdMPI, quiet = TRUE)
init()

set.seed(123)
x <- matrix(rnorm(1000000), 1000)

parMM.spmd <- function(x, y){
  id <- get.jid(nrow(x))
  do.call(rbind, allgather(x[id,] %*% y))
}
time.proc <- system.time(replicate(10, parMM.spmd(x, x)))
comm.print(time.proc)

finalize()
```

The following shows the computing time of the above code on a single machine with two processors Intel(R) Core(TM) i5-2410M CPU @ 2.30 GHz, xubuntu 11.04 system, and OpenMPI

1.6. **pbdMPI** performs better than **snow** in this example even without communication over network.

R Output

```
>> Test ./02_parMM_par.r
   user  system elapsed
 12.460   0.170  12.625
   user  system elapsed
  1.780   0.820  10.095

>> Test ./02_parMM_spmd.r
COMM.RANK = 0
   user  system elapsed
   8.84    0.42    9.26
```

6. Long Vector and 64-bit for MPI

6.1. Long Vector for MPI

The current R (3.1.0) uses C structure to extend 32-bit length limitation ($2^{31}-1 = 2147483647$ defined as `R_SHORT_LEN_MAX`) to 52-bit length ($2^{51} - 1 = 4503599627370496$ defined as `R_XLEN_T_MAX`). In general, this is more portable and extensible when 128-bit integer coming on (who know when the day comes ...) However, a vector with elements larger than $2^{31} - 1$ needs extra effort to be accessed in R. See "R Internals" for details.

The reason is that an integer is 4 bytes in both of x86_64 system (64-bit) and i386 system (32-bit). Since the capacity of current machine and performance issues, there is no benefit to use 8 bytes for integer. In x86_64 system, computers or compilers use either `long` or `long long` for pointer address which is in `size_t` for unsigned address or in `ptrdiff_t` for signed address. For example, in GNU C (`gcc`), the flag `-m64` is to use 4 bytes for `int` and 8 bytes for `long` in x86_64 system.[1]

Therefore, the question is what are the differences of 64-bit and 32-bit system? One of them is "pointer size" which is 8 bytes in x86_64 machine and it is 4 bytes in i386 machine. This allows computer to lengthen memory and disk space. Note that address is indexed by `long` or `long long` which is no confilict with integer size, and 4 bytes integer is efficient and safe enough for general purpose. For example, `double *a` is a pointer (`a`) pointing to a real scaler (`*a`), but the pointer's address (`&a`) is in `size_t` (`long` or `long long`) which is 8 bytes in x86_64 system and is 4 bytes in i386 system.

To deal with long vector, **pbdMPI** uses the same framework as R to build up MPI collective functions. **pbdMPI** follows R's standard to assume a vector normally has length smaller than `R_SHORT_LEN_MAX` which can be handled by most 32-bit functions. If the vector length is greater than `R_SHORT_LEN_MAX`, then R names this as long vector which also has the maximum `R_XLEN_T_MAX`. The vector length is stored in type `R_xlen_t`. The `R_xlen_t` is `long` if `LONG_VECTOR_SUPPORT` is defined, otherwise it is `int`. R provides several C macro to check, access, and manipulate the vector in `VECSXP` or general `SEXP`. See `Rinternals.h` for details.

[1] Is there a way to have 8 bytes integer? The answer is that it is dependent on compiler.

The **pbdMPI** first checks if the data size for communication is greater than `SPMD_SHORT_LEN_MAX` or not. If the data is long vector, then **pbdMPI** evokes collective functions to send/receive chunk of data partitioned by `SPMD_SHORT_LEN_MAX` until all chunks are all received/sent. For some MPI collective functions such as `allgather()` and `gather()`, extra space may be allocated for receving chunks, then the chunks are copied to right memory address by the rank of communicator from the extra space to the receiving buffer.

The reason is that most MPI collective functions rely on arguments for indexing buff types and counting buffer sizes where the types and sizes are both in `int`. `SPMD_SHORT_LEN_MAX` is defined in `pbdMPI/src/spmd.h` and usually is equal to `R_SHORT_LEN_MAX`. Developers may want to use shorter length (such as `SPMD_INT8_LEN_MAX` which is $2^7 - 1 = 127$) for testing without a large memory machine or for debugging without recompiling R with shorter `R_SHORT_LEN_MAX`.

In **pbdMPI**, the implemented MPI collective functions for long vector are `bcast()`, `allreduce()`, `reduce()`, `send()`, `recv()`, `isend()`, `irecv()`, `allgather()`, `gather()`, and `scatter()`. The other MPI collective functions are "NOT" implemented due to the complexity of memory allocation for long vector including `allgatherv()`, `gatherv()`, `scatterv()`, `sendrecv()`, and `sendrecv.replace()`.

Further, **pbdMPI** provides a way to mimic long vector support. Users can set

pkg_constant.h

```
#define MPI_LONG_DEBUG 1
```

in `pbdMPI/src/pkg_constant.h` to turn on debugging mode and recompile **pbdMPI**. Then, run examples in `pbdMPI/inst/examples/test_long_vector/` to see how the mimic long vectors are communicated between processors. Also, users can also adjust the length limit of mimic long vector (buffer size) by changing

spmd.h

```
#define SPMD_SHORT_LEN_MAX R_SHORT_LEN_MAX
```

in `pbdMPI/src/spmd.h`.

6.2. 64-bit for MPI

The remaining question is that does MPI library support 64-bit system? The answer is yes, but users may need to recompile MPI libraries for 64-bit support. The same way as R to enable 64-bit system that MPI libraries may have 8 bytes pointer in order to communicate larger memory or disk space.[2]

For example, the OpenMPI provides next to check if 64-bit system is used.

Shell Command

```
ompi_info -a | grep 'int.* size: '
```

If the output is

[2] http://wiki.chem.vu.nl/dirac/index.php/How_to_build_MPI_libraries_for_64-bit_integers.

Shell Command

```
             C int size: 4
         C pointer size: 8
       Fort integer size: 8
      Fort integer1 size: 1
      Fort integer2 size: 2
      Fort integer4 size: 4
      Fort integer8 size: 8
     Fort integer16 size: -1
```

then the OpenMPI supports 64-bit system.[3] Otherwise, users may use the next to reinstall OpenMPI as

Shell Command

```
./configure --prefix=/path_to_openmpi \
            CFLAGS=-fPIC \
            FFLAGS="-m64 -fdefault-integer-8" \
            FCFLAGS="-m64 -fdefault-integer-8" \
            CFLAGS=-m64 \
            CXXFLAGS=-m64
```

and remember to reinstall **pbdMPI** as well.

Note that 64-bit pointer may only provide larger size of data, but may degrade hugely for other computing. In general, communication with a large amount of data is a very bad idea. Try to redesign algorithms to communicate lightly such as via sufficient statistics, or to rearrange and load large data partially or equally likely to every processors.

[3] The C integer is still in 4 bytes rather than 8 bytes.

References

Chen W-C Schmidt D, Sehrawat G, Patel P, Ostrouchov G (2013). "pbdPROF: Programming with Big Data – MPI Profiling Tools." R Package, URL http://cran.r-project.org/package=pbdPROF.

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

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/.

Raim A (2013). *Introduction to distributed computing with pbdR at the UMBC High Performance Computing Facility (Technical report HPCF-2013-2)*. UMBC High Performance Computing Facility, University of Maryland, Baltimore County.

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 (v:0.1-7), URL http://cran.r-project.org/package=multicore.

Vetter JS, McCracken MO (2001). "Statistical scalability analysis of communication operations in distributed applications." In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPoPP '01, pp. 123–132. ACM, New York, NY, USA. ISBN 1-58113-346-4. doi:10.1145/379539.379590. URL http://doi.acm.org/10.1145/379539.379590.

Yu H (2002). "Rmpi: Parallel Statistical Computing in R." *R News*, **2**(2), 10–14. URL http://cran.r-project.org/doc/Rnews/Rnews_2002-2.pdf.