Introduction to the Rhpc Package

Ei-ji Nakama

July 31, 2026

1 Overview

The `Rhpc` package provides an interface for achieving High-Performance Computing (HPC) in R using the Message Passing Interface (MPI). Unlike standard parallel processing libraries that rely on sockets or fork, it provides direct and low-latency access to MPI, making it scalable from multi-core PCs to large-scale computational clusters.

1.1 Key Features

- **MPI-Specific API**: All functions are prefixed with `Rhpc_` to avoid namespace collisions and explicitly indicate the MPI backend (e.g., `Rhpc_lapply`, `Rhpc_worker_call`).
- **Dynamic Worker Management**: Supports both static launch via `mpirun` and dynamic generation during execution via `MPI_Comm_spawn`.
- **Hybrid Communication**: Maximizes throughput by using Collective communication for instruction distribution and Point-to-Point communication for result collection.
- **Windows Support**: The `fakemaster` mechanism allows using MPI even from GUI environments like RGui or RStudio without complex command-line launches.
- **Long Vector Support**: Designed to handle somewhat large data by supporting long vectors.

2 Installation and Launch

2.1 Linux / Unix

An MPI library (e.g., OpenMPI, MPICH2) is required. Build and install the package:

```
R CMD INSTALL Rhpc_0.26.5.tar.gz
```

If using other MPI implementations like Fujitsu MPI, specify compile and link flags with `-configure-args`.

2.1.1 Launch via mpirun (Batch Execution)

Use the `Rhpc` shell generated within the package to launch R with multiple processes:

```
mpirun -n 4 ~/R/.../Rhpc/Rhpc CMD BATCH -q --no-save test.R
```

2.2 Windows (MS-MPI)

Install Microsoft MPI (MS-MPI) and set the environment variables `MSMPI_INC`, `MSMPI_LIB32`, and `MSMPI_LIB64`. Simply calling `Rhpc_initialize()` from RGui or RStudio will automatically launch `mpiexec` and `fakemaster`.

3 Basic Usage

To use `Rhpc`, first initialize the MPI environment and obtain a worker handle (`cl`). The behavior of `Rhpc_getHandle()` depends on how it is called:

- **No arguments**: Obtains existing worker processes if R was already launched under `mpirun` or `mpiexec`.
- **Numeric argument**: Dynamically generates the specified number of worker processes (e.g., `Rhpc_getHandle(4)`).

A typical workflow is: `Rhpc_initialize()` → `Rhpc_getHandle()` → Parallel Processing → `Rhpc_finalize()`.

```
> library(Rhpc)
> # Initialize Rhpc environment
> Rhpc_initialize()
> # Get handle (no arguments under mpirun, or Rhpc_getHandle(4) for dynamic generation)
> cl <- Rhpc_getHandle()
> # Execute function on all workers and gather results on the master
> pids <- Rhpc_worker_call(cl, Sys.getpid)
> print(pids)
> # Execute code on workers without returning results
> Rhpc_worker_noback(cl, function() {
+     cat("worker process:", Sys.getpid(), "\n")
+ })
> # Finalize
> Rhpc_finalize()
```

4 Backend Management on Windows

On Windows, `Rhpc_initialize()` performs the following internally:

1. Launches `fakemaster` if the current R process is not running under `mpiexec`.
2. Launches `fakemaster.exe` via `mpiexec` and connects it to the R process using a named pipe.
3. Imports the MPI environment variables generated by `fakemaster` into the R process.
4. This enables the use of `Rhpc_getHandle()` and `Rhpc_worker_call()` even in GUI-based R sessions.

Note:

- When `Rhpc_initialize()` is executed, windows for `mpiexec` and `fakemaster` will launch in the background. Closing these will break the MPI connection and cause the process to fail.
- You can specify the `mpiexec` command line using `options(Rhpc.mpiexec="mpiexec -n 1")`.

```
> library(Rhpc)
> oldoptions <- options(Rhpc.mpiexec = "mpiexec -n 1")
> Rhpc_initialize()
> cl <- Rhpc_getHandle(3)
> worker_env_info <- Rhpc_worker_call(cl, function() {
+     list(
+         computer_name = Sys.getenv("COMPUTERNAME"),
+         username = Sys.getenv("USERNAME")
+     )
+ })
> print(worker_env_info)
> Rhpc_finalize()
> options(oldoptions)
```

5 Choosing Between Collective and Point-to-Point Communication

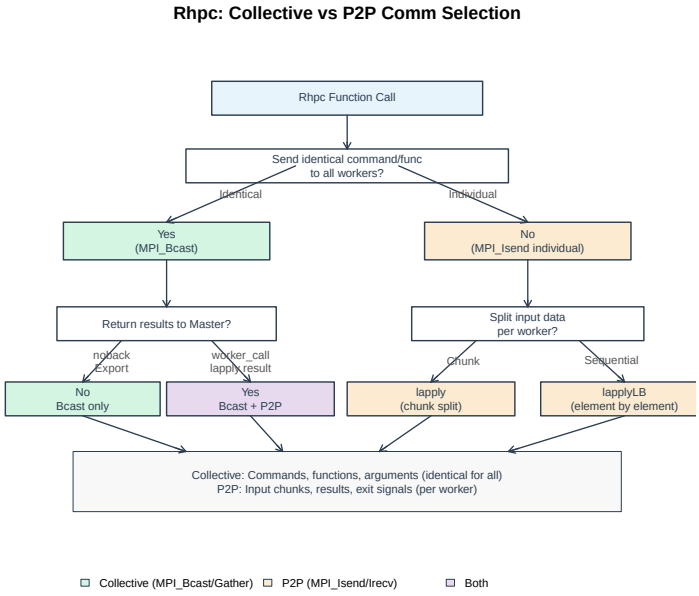
`Rhpc` uses either **Collective** communication or **Point-to-Point (P2P)** communication depending on the use case. The basic principles for selection are as follows:

- **Send identical content to all workers** → Collective (`MPI_Bcast`)
- **Content varies per worker** → P2P (`MPI_Isend` / `MPI_Irecv`)

- **Result size varies per worker** → P2P (Asynchronous receive)
- **No results returned** → Complete using only Collective

5.1 Communication Method Decision Flow

The following figure shows the flowchart of how Rhpc chooses the communication method for each operation.



5.2 MPI Functions and Roles

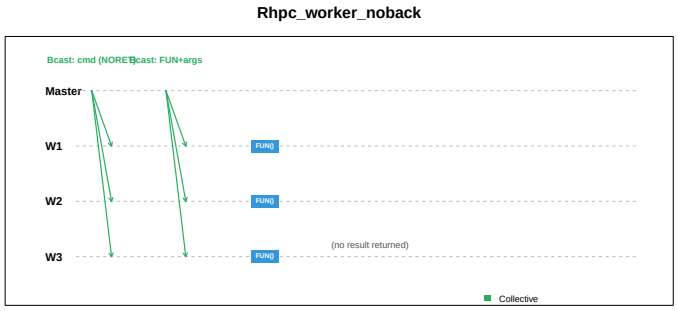
Comm. Type	MPI Function	Usage in Rhpc
Collective	`MPI_Bcast`	Broadcast command/function/args to all workers
Collective	`MPI_Gather`	Gather result size info from each worker to master
P2P	`MPI_Isend`	Master → Worker (input chunk), Worker → Master (result)
P2P	`MPI_Irecv`	Master asynchronously receives results from workers
P2P	`MPI_Probe`	Detect completed workers in `lapply`/`lapplyLB`

5.3 Sequence Diagrams by Function

The following are communication sequence diagrams for the main functions. All diagrams are drawn using R's `plot`.

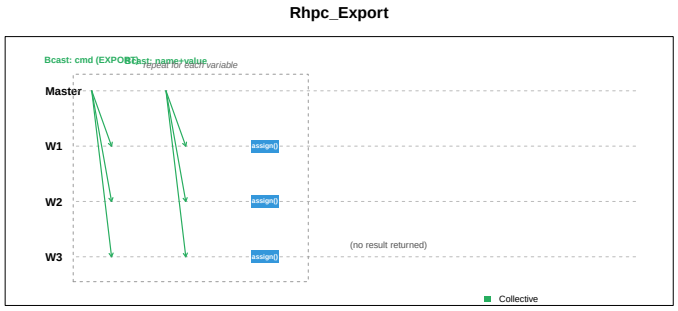
5.3.1 Rhpc_worker_noback()

Worker execution that returns no results. Completes using only Collective communication (`MPI_Bcast`).



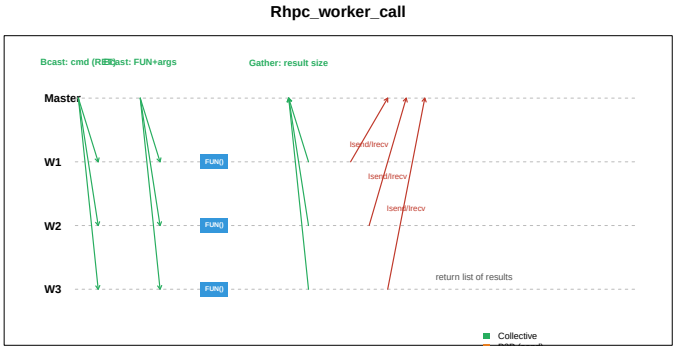
5.3.2 Rhpc_Export()

Distributes variables to the global environment of all workers. Like `Rhpc_worker_noback`, it uses only `MPI_Bcast`, but executes `assign()` on the workers. Bcast is repeated for each variable.



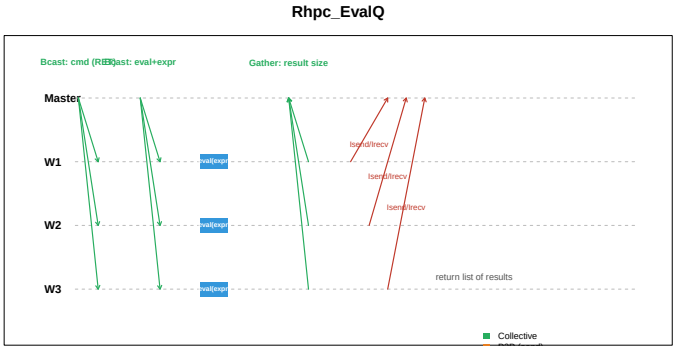
5.3.3 Rhpc_worker_call()

Executes the same function on all workers and returns results to the master. Command distribution is Bcast, result size is Gather, and result bodies are P2P (Isend/Irecv).



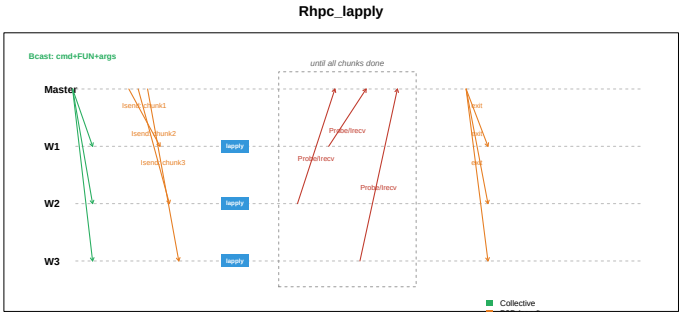
5.3.4 Rhpc_EvalQ()

A wrapper for `Rhpc_worker_call(cl, eval, substitute(expr))`. The communication pattern is identical to `Rhpc_worker_call`.



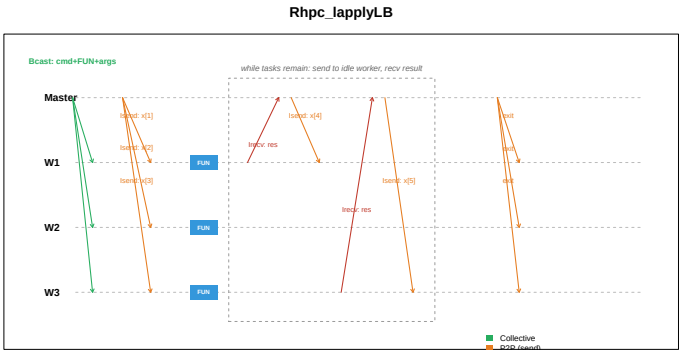
5.3.5 Rhpc_lapply()

Splits input `X` into chunks based on the number of workers, distributes the function via Bcast, and input chunks via P2P. Results are collected in completion order using `MPI_Probe`/`MPI_Irecv`, finally sending an exit signal via P2P.



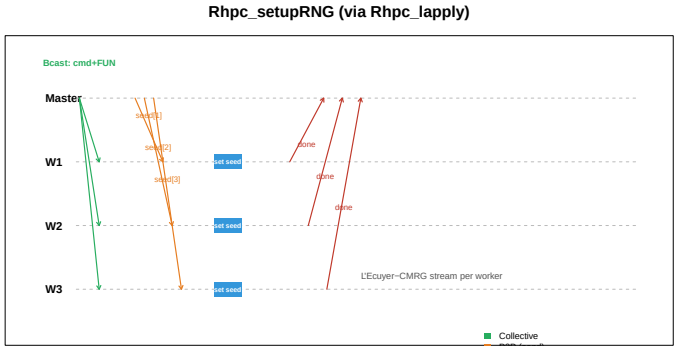
5.3.6 Rhpc_lapplyLB()

Broadcasts the function, but distributes inputs **element by element** to idle workers via P2P. Sends the next element as a worker finishes, and retrieves results sequentially via P2P.



5.3.7 Rhpc_setupRNG()

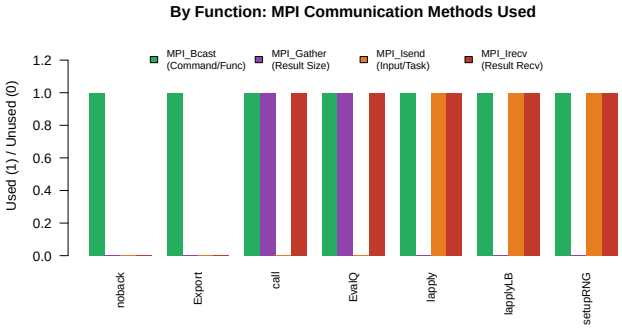
Automatically called within `Rhpc_getHandle()`. Uses `Rhpc_lapply` internally to distribute distinct random seeds to each worker via P2P (same communication pattern as `lapply`).



5.4 Comparison of Communication Methods

The following figure shows a summary of differences in communication methods across the main functions.

```
> draw_comm_comparison <- function() {
+     funcs <- c("noback", "Export", "call", "EvalQ", "lapply", "lapplyLB", "setupRNG")
+     bcast <- c(1, 1, 1, 1, 1, 1, 1)
+     gather <- c(0, 0, 1, 1, 0, 0, 0)
+     isend_in <- c(0, 0, 0, 0, 1, 1, 1)
+     irecv_out <- c(0, 0, 1, 1, 1, 1, 1)
+
+     mat <- rbind(
+         "MPI_Bcast\n(Command/Func)" = bcast,
+         "MPI_Gather\n(Result Size)" = gather,
+         "MPI_Isend\n(Input/Task)" = isend_in,
+         "MPI_Irecv\n(Result Recv)" = irecv_out
+     )
+     colnames(mat) <- funcs
+
+     oldpar <- par(mar = c(6, 8, 3, 2))
+     on.exit(par(oldpar))
+     barplot(mat, beside = TRUE, col = c("#27AE60", "#8E44AD", "#E67E22", "#C0392B"),
+             border = NA, las = 2, cex.names = 0.85,
+             main = "By Function: MPI Communication Methods Used",
+             ylab = "Used (1) / Unused (0)", ylim = c(0, 1.35))
+     legend("top", inset = 0.02, horiz = TRUE, bty = "n", cex = 0.75,
+            legend = rownames(mat),
+            fill = c("#27AE60", "#8E44AD", "#E67E22", "#C0392B"))
+ }
> draw_comm_comparison()
```



5.5 Reasons for Communication Choices

- **Why use Bcast for commands?**: Functions and arguments are identical for all workers. Using a single serialization and tree-based distribution ($O(\log N)$) is more efficient than sending sequentially to each worker ($O(N)$).
- **Why use P2P for input data (`lapply`)?**: Input chunks for each worker differ in content and size. Bcast is inappropriate since it sends identical data to everyone.
- **Why use P2P for results?**: The result size for each worker is unknown until runtime and varies between workers. Asynchronous reception with `MPI_Irecv` allows processing results from faster workers first.
- **Why `lapplyLB` uses P2P sequential distribution?**: When task execution times vary, dynamic allocation to idle workers (`lapplyLB`) yields higher CPU utilization than pre-chunking (`lapply`).

6 List of Main Functions

6.1 Environment Management

- `Rhpc_initialize()`: Initializes the MPI environment.
- `Rhpc_finalize()`: Terminates the MPI environment.
- `Rhpc_getHandle(procs)`: Obtains the worker cluster handle. Dynamically generates workers if a numeric value is specified for `procs`.
- `Rhpc_numberOfWorker(cl)`: Returns the number of workers.

6.2 Execution on Workers

- `Rhpc_worker_call(cl, FUN, ...)`: Executes `FUN` on all workers and returns a list of results.
- `Rhpc_worker_noback(cl, FUN, ...)`: Executes `FUN` on all workers but returns no results.
- `Rhpc_EvalQ(cl, expr)`: Evaluates the expression `expr` on all workers (equivalent to `clusterEvalQ`).

6.3 Parallel *apply Series

- `Rhpc_lapply(cl, X, FUN, ...)`: Processes a list/vector in parallel similarly to `lapply`. Splits input into chunks and distributes to workers.
- `Rhpc_lapplyLB(cl, X, FUN, ...)`: Load-balancing version. Sequentially assigns tasks to idle workers, effective when execution times vary.
- `Rhpc_sapply(cl, X, FUN, ...)`: Equivalent to `sapply`. Can simplify results into a vector or matrix.
- `Rhpc_sapplyLB(cl, X, FUN, ...)`: Load-balancing version of `sapply`.
- `Rhpc_apply(cl, X, MARGIN, FUN, ...)`: Equivalent to `apply`. Applies in parallel along specified dimensions of an array.

6.4 Data Sharing and Utilities

- `Rhpc_Export(cl, variableNames)`: Distributes variables defined on the master to the global environment of all workers.
- `Rhpc_setupRNG(cl, iseed)`: Sets an independent random number stream (L'Ecuyer-CMRG) for each worker. Automatically called within `Rhpc_getHandle()`.
- `Rhpc_enquote(...)`: Internal utility for quoting arguments.
- `Rhpc_splitList(var, num)`: Splits a list into a specified number of parts.
- `Rhpc_serialize()` / `Rhpc_unserialize()`: Custom serialization functions.

7 Parallel Mapping (Rhpc_lapply)

`Rhpc_lapply` has the same interface as `lapply`, splitting input data into chunks based on the number of workers for parallel execution.

```
> Rhpc_initialize()
> cl <- Rhpc_getHandle()
> input_data <- 1:10
> results <- Rhpc_lapply(cl, input_data, function(x) {
+     return(x^2)
+ })
> print(unlist(results))
> Rhpc_finalize()
```

8 Load Balancing (Rhpc_lapplyLB)

When execution times for each element vary, `Rhpc_lapplyLB` is effective. Instead of pre-chunking, it sequentially assigns tasks to idle workers, thereby improving CPU utilization.

```
> Rhpc_initialize()
> cl <- Rhpc_getHandle()
> input_data <- 1:10
> results_lb <- Rhpc_lapplyLB(cl, input_data, function(x) {
+     Sys.sleep(runif(1, 0, 1))  # Simulate execution time variance
+     return(x * 10)
+ })
> print(unlist(results_lb))
> Rhpc_finalize()
```

9 Using sapply / apply

`Rhpc_sapply` and `Rhpc_apply` can be used similarly to the `parallel` package.

```
> Rhpc_initialize()
> cl <- Rhpc_getHandle()
> # Equivalent to sapply
> res <- Rhpc_sapply(cl, 1:10000, sqrt)
> # Equivalent to apply
> df <- data.frame(a = 1:4, b = 5:8)
> Rhpc_apply(cl, df, 1, max)  # Row direction
> Rhpc_apply(cl, df, 2, max)  # Column direction
> Rhpc_finalize()
```

10 Exporting Variables to Workers

To use variables defined on the master inside workers, distribute them to the global environment using `Rhpc_Export`.

```
> Rhpc_initialize()
> cl <- Rhpc_getHandle()
> multiplier <- 10
> Rhpc_Export(cl, "multiplier")
> res <- Rhpc_worker_call(cl, function() {
+     return(multiplier * 2)
+ })
> print(res)
> Rhpc_finalize()
```

11 Options and MPI Attributes

After `Rhpc_initialize()`, the following options are set in `options()`:

- `Rhpc.mpi.rank`: Current MPI rank
- `Rhpc.mpi.procs`: Total number of processes
- `Rhpc.mpi.c.comm`: C language MPI communicator (pointer)
- `Rhpc.mpi.f.comm`: Fortran MPI communicator

The `usequote` argument (default `TRUE`) can be disabled with `options(Rhpc.usequote=FALSE)`, which may improve performance.

12 Conclusion

By using `Rhpc`, you can achieve both the flexibility of R's statistical programming and the high performance of MPI. The main advantages are:

- **Efficiency**: Smaller communication overhead compared to socket-based parallel processing.
- **Scalability**: Smooth transition from local multi-core to large-scale clusters.
- **Simplicity**: Can be parallelized using standard R style syntax like `lapply` and `apply`.
- **Windows Support**: Can utilize MPI even from GUI environments using `fakemaster`.