Introduction to *BiocParallel*

Valerie Obenchain, Vincent Carey, Michael Lawrence, Martin Morgan[1]

[1]Martin.Morgan@RoswellPark.org

Edited: January 23, 2021; Compiled: September 19, 2021

Contents

1 Introduction

Numerous approaches are available for parallel computing in *R*. The CRAN Task View for high performance and parallel computing provides useful high-level summaries and package categorization. http://cran.r-project.org/web/views/HighPerformanceComputing.html Most Task View packages cite or identify one or more of *snow*, *Rmpi*, *multicore* or *foreach* as relevant parallelization infrastructure. Direct support in *R* for parallel computing started with release 2.14.0 with inclusion of the *parallel* package which contains modified versions of *multicore* and *snow*.

A basic objective of *BiocParallel* is to reduce the complexity faced when developing and using software that performs parallel computations. With the introduction of the `BiocParallelParam` object, *BiocParallel* aims to provide a unified interface to existing parallel infrastructure where code can be easily executed in different environments. The `BiocParallelParam` specifies the environment of choice as well as computing resources and is invoked by 'registration' or passed as an argument to the *BiocParallel* functions.

BiocParallel offers the following conveniences over the 'roll your own' approach to parallel programming.

- unified interface: `BiocParallelParam` instances define the method of parallel evaluation (multi-core, snow cluster, etc.) and computing resources (number of workers, error handling, cleanup, etc.).
- parallel iteration over lists, files and vectorized operations: `bplapply`, `bpmapply` and `bpvec` provide parallel list iteration and vectorized operations. `bpiterate` iterates through files distributing chunks to parallel workers.
- cluster scheduling: When the parallel environment is managed by a cluster scheduler through *batchtools*, job management and result retrieval are considerably simplified.
- support of `foreach`: The *foreach* and *iterators* packages are fully supported. Registration of the parallel back end uses `BiocParallelParam` instances.

2 Quick start

The *BiocParallel* package is available at bioconductor.org and can be downloaded via `BiocManager`:

```
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
BiocManager::install("BiocParallel")
```

Load *BiocParallel*.

```
library(BiocParallel)
```

The test function simply returns the square root of "x".

```
FUN <- function(x) { round(sqrt(x), 4) }
```

Functions in *BiocParallel*use the registered back-ends for parallel evaluation. The default is the top entry of the registry list.

```
registered()
```

```
## $MulticoreParam
## class: MulticoreParam
##   bpisup: FALSE; bpnworkers: 4; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE; bpforceGC: TRUE
##   bplogdir: NA
##   bpresultdir: NA
##   cluster type: FORK
##
## $SnowParam
## class: SnowParam
##   bpisup: FALSE; bpnworkers: 4; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE; bpforceGC: FALSE
##   bplogdir: NA
##   bpresultdir: NA
##   cluster type: SOCK
##
## $SerialParam
## class: SerialParam
##   bpisup: FALSE; bpnworkers: 1; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE; bpforceGC: FALSE
##   bplogdir: NA
##   bpresultdir: NA
```

Configure your R session to always use a particular back-end configure by setting options named after the back ends in an `.Rprofile` file, e.g.,

```
options(MulticoreParam=quote(MulticoreParam(workers=4)))
```

When a *BiocParallel* function is invoked with no `BPPARAM` argument the default back-end is used.

```
bplapply(1:4, FUN)
```

Environment specific back-ends can be defined for any of the registry entries. This example uses a 2-worker SOCK cluster.

```
param <- SnowParam(workers = 2, type = "SOCK")
bplapply(1:4, FUN, BPPARAM = param)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.4142
##
```

```
## [[3]]
## [1] 1.7321
##
## [[4]]
## [1] 2
```

3 The *BiocParallel* Interface

3.1 Classes

3.1.1 `BiocParallelParam`

`BiocParallelParam` instances configure different parallel evaluation environments. Creating or `register()`ing a '`Param`' allows the same code to be used in different parallel environments without a code re-write. Params listed are supported on all of Unix, Mac and Windows except `MulticoreParam` which is Unix and Mac only.

- `SerialParam`:

  Supported on all platforms.

  Evaluate *BiocParallel*-enabled code with parallel evaluation disabled. This approach is useful when writing new scripts and trying to debug code.

- `MulticoreParam`:

  Supported on Unix and Mac. On Windows, `MulticoreParam` dispatches to `SerialParam`.

  Evaluate *BiocParallel*-enabled code using multiple cores on a single computer. When available, this is the most efficient and least troublesome way to parallelize code. Windows does not support multi-core evaluation (the `MulticoreParam` object can be used, but evaluation is serial). On other operating systems, the default number of workers equals the value of the global option `mc.cores` (e.g., `getOption("mc.cores")`) or, if that is not set, the number of cores returned by `parallel::detectCores() - 2`; when number of cores cannot be determined, the default is 1.

  `MulticoreParam` uses 'forked' processes with 'copy-on-change' semantics – memory is only copied when it is changed. This makes it very efficient to invoke compared to other back-ends.

  There are several important caveats to using `MulticoreParam`. Forked processes are not available on Windows. Some environments, e.g., *RStudio*, do not work well with forked processes, assuming that *R* code evaluation is single-threaded. Some external resources, e.g., access to files or data bases, maintain state in a way that assumes the resource is accessed only by a single thread. A subtle cost is that *R*'s garbage collector runs periodically, and 'marks' memory as in use. This effectively triggers a copy of the marked memory. *R*'s generational garbage collector is triggered at difficult-to-predict times; the effect in a long-running forked process is that the memory is eventually copied. See this post for additional details.

  `MulticoreParam` is based on facilities originally implemented in the *multicore* package and subsequently the *parallel* package in base *R*.

- `SnowParam`:

  Supported on all platforms.

  Evaluate *BiocParallel*-enabled code across several distinct *R* instances, on one or several computers. This is a straightforward approach for executing parallel code on one or several computers, and is based on facilities originally implemented in the *snow* package. Different types of *snow* 'back-ends' are supported, including socket and MPI clusters.

- `BatchtoolsParam`:

  Applicable to clusters with formal schedulers.

  Evaluate *BiocParallel*-enabled code by submitting to a cluster scheduler like SGE.

- `DoparParam`:

  Supported on all platforms.

  Register a parallel back-end supported by the *foreach* package for use with *BiocParallel*.

The simplest illustration of creating `BiocParallelParam` is

```
serialParam <- SerialParam()
serialParam
```

```
## class: SerialParam
##   bpisup: FALSE; bpnworkers: 1; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE; bpforceGC: FALSE
##   bplogdir: NA
##   bpresultdir: NA
```

Most parameters have additional arguments influencing behavior, e.g., specifying the number of 'cores' to use when creating a `MulticoreParam` instance

```
multicoreParam <- MulticoreParam(workers = 8)
multicoreParam
```

```
## class: MulticoreParam
##   bpisup: FALSE; bpnworkers: 8; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE; bpforceGC: TRUE
##   bplogdir: NA
##   bpresultdir: NA
##   cluster type: FORK
```

Arguments are described on the corresponding help page, e.g., `?MulticoreParam`.

3.1.2 `register()`ing `BiocParallelParam` instances

The list of registered `BiocParallelParam` instances represents the user's preferences for different types of back-ends. Individual algorithms may specify a preferred back-end, and different back-ends maybe chosen when parallel evaluation is nested.

The registry behaves like a 'stack' in that the last entry registered is added to the top of the list and becomes the "next used" (i.e., the default).

`registered` invoked with no arguments lists all back-ends.

```
registered()
```

```
## $MulticoreParam
## class: MulticoreParam
##   bpisup: FALSE; bpnworkers: 4; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE; bpforceGC: TRUE
##   bplogdir: NA
##   bpresultdir: NA
##   cluster type: FORK
##
## $SnowParam
## class: SnowParam
##   bpisup: FALSE; bpnworkers: 4; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE; bpforceGC: FALSE
##   bplogdir: NA
##   bpresultdir: NA
##   cluster type: SOCK
##
## $SerialParam
## class: SerialParam
##   bpisup: FALSE; bpnworkers: 1; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE; bpforceGC: FALSE
##   bplogdir: NA
##   bpresultdir: NA
```

`bpparam` returns the default from the top of the list.

```
bpparam()
```

```
## class: MulticoreParam
##   bpisup: FALSE; bpnworkers: 4; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE; bpforceGC: TRUE
##   bplogdir: NA
##   bpresultdir: NA
##   cluster type: FORK
```

Add a specialized instance with `register`. When `default` is TRUE, the new instance becomes the default.

```
default <- registered()
register(BatchtoolsParam(workers = 10), default = TRUE)
```

BatchtoolsParam has been moved to the top of the list and is now the default.

```
names(registered())
```

```
## [1] "BatchtoolsParam" "MulticoreParam"  "SnowParam"       "SerialParam"
```

```
bpparam()
```

```
## class: BatchtoolsParam
##   bpisup: FALSE; bpnworkers: 10; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: NA; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE; bpforceGC: FALSE
##   bplogdir: NA
##   bpresultdir: NA
##   cluster type: multicore
##   template: NA
##   registryargs:
##     file.dir: /tmp/RtmpB7xHKT/Rbuildee8b1eeb2110/BiocParallel/vignettes/fileef0633472a47
##     work.dir: getwd()
##     packages: character(0)
##     namespaces: character(0)
##     source: character(0)
##     load: character(0)
##     make.default: FALSE
##   saveregistry: FALSE
##   resources:
```

Restore the original registry

```
for (param in rev(default))
    register(param)
```

3.2 Functions

3.2.1 Parallel looping, vectorized and aggregate operations

These are used in common functions, implemented as much as possible for all back-ends. The functions (see the help pages, e.g., `?bplapply` for a full definition) include

`bplapply(X, FUN, ...)`:

> Apply in parallel a function `FUN` to each element of `X`. `bplapply` invokes `FUN` `length(X)` times, each time with a single element of `X`.

`bpmapply(FUN, ...)`:

> Apply in parallel a function `FUN` to the first, second, etc., elements of each argument in `...`.

`bpiterate(ITER, FUN, ...)`:

> Apply in parallel a function `FUN` to the output of function `ITER`. Data chunks are returned by `ITER` and distributed to parallel workers along with `FUN`. Intended for iteration though an undefined number of data chunks (i.e., records in a file).

`bpvec(X, FUN, ...)`:

Apply in parallel a function `FUN` to subsets of `X`. `bpvec` invokes function `FUN` as many times as there are cores or cluster nodes, with `FUN` receiving a subset (typically more than 1 element, in contrast to `bplapply`) of `X`.

`bpaggregate(x, data, FUN, ...)`:

Use the formula in `x` to aggregate `data` using `FUN`.

3.2.2 Parallel evaluation environment

These functions query and control the state of the parallel evaluation environment.

`bpisup(x)`: Query a `BiocParallelParam` back-end `x` for its status.

`bpworkers`; `bpnworkers`: Query a `BiocParallelParam` back-end for the number of workers available for parallel evaluation.

`bptasks`: Divides a job (e.g., single call to *lapply function) into tasks. Applicable to `MulticoreParam` only; `DoparParam` and `BatchtoolsParam` have their own approach to dividing a job among workers.

`bpstart(x)`: Start a parallel back end specified by `BiocParallelParam` `x`, if possible.

`bpstop(x)`: Stop a parallel back end specified by `BiocParallelParam` `x`.

3.2.3 Error handling and logging

Logging and advanced error recovery is available in `BiocParallel` 1.1.25 and later. For a more details see the vignette titled "Error Handling and Logging":

```
browseVignettes("BiocParallel")
```

3.2.4 Locks and counters

Inter-process (i.e., single machine) locks and counters are supported using `ipclock()`, `ipcyield()`, and friends. Use these to synchronize computation, e.g., allowing only a single process to write to a file at a time.

4 Use cases

Sample data are BAM files from a transcription profiling experiment available in the *RNAseqData.HNRNPC.bam.chr14* package.

```
library(RNAseqData.HNRNPC.bam.chr14)
fls <- RNAseqData.HNRNPC.bam.chr14_BAMFILES
```

4.1 Single machine

Common approaches on a single machine are to use multiple cores in forked processes, or to use clusters of independent processes.

For purely *R*-based computations on non-Windows computers, there are substantial benefits, such as shared memory, to be had using forked processes. However, this approach is not portable across platforms, and fails when code uses functionality, e.g., file or data base

access, that assumes only a single thread is accessing the resource. While use of forked processes with `MulticoreParam` is an attractive solution for scripts using pure *R* functionality, robust and complex code often requires use of independent processes and `SnowParam`.

4.1.1 Forked processes with `MulticoreParam`

This example counts overlaps between BAM files and a defined set of ranges. First create a GRanges with regions of interest (in practice this could be large).

```
library(GenomicAlignments) ## for GenomicRanges and readGAlignments()
gr <- GRanges("chr14", IRanges((1000:3999)*5000, width=1000))
```

A *ScanBamParam* defines regions to extract from the files.

```
param <- ScanBamParam(which=range(gr))
```

FUN counts overlaps between the ranges in 'gr' and the files.

```
FUN <- function(fl, param) {
  gal <- readGAlignments(fl, param = param)
  sum(countOverlaps(gr, gal))
}
```

All parameters necessary for running a job in a multi-core environment are specified in the *MulticoreParam* instance.

```
MulticoreParam()
```

```
## class: MulticoreParam
##   bpisup: FALSE; bpnworkers: 4; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE; bpforceGC: TRUE
##   bplogdir: NA
##   bpresultdir: NA
##   cluster type: FORK
```

The *BiocParallel* functions, such as `bplapply`, use information in the *MulticoreParam* to set up the appropriate back-end and pass relevant arguments to low-level functions.

```
> bplapply(fls[1:3], FUN, BPPARAM = MulticoreParam(), param = param)
$ERR127306
[1] 1185

$ERR127307
[1] 1123

$ERR127308
[1] 1241
```

Shared memory environments eliminate the need to pass large data between workers or load common packages. Note that in this code the GRanges data was not passed to all workers in `bplapply` and FUN did not need to load *GenomicAlignments* for access to the `readGAlignments` function.

Problems with forked processes occur when code implementating functionality used by the workers is not written in anticipation of use by forked processes. One example is the database connection underlying Bioconductor's `org.*` packages. This psudoe-code

```
library(org.Hs.eg.db)
FUN <- function(x, ...) {
    ...
    mapIds(org.Hs.eg.db, ...)
    ...
}
bplapply(X, FUN, ..., BPPARAM = MulticoreParam())
```

is likely to fail, because `library(org.Hs.eg.db)` opens a database connection that is accessed by multiple processes. A solution is to ensure that the database is opened independently in each process

```
FUN <- function(x, ...) {
    library(org.Hs.eg.db)
    ...
    mapIds(org.Hs.eg.db, ...)
    ...
}
bplapply(X, FUN, ..., BPPARAM = MulticoreParam())
```

4.1.2 Clusters of independent processes with `SnowParam`

Both Windows and non-Windows machines can use the cluster approach to spawn processes. *BiocParallel* back-end choices for clusters on a single machine are *SnowParam* for configuring a Snow cluster or the *DoparParam* for use with the *foreach* package.

To re-run the counting example, FUN needs to modified such that 'gr' is passed as a formal argument and required libraries are loaded on each worker. (In general, this is not necessary for functions defined in a package name space, see Section 6.)

```
FUN <- function(fl, param, gr) {
  suppressPackageStartupMessages({
     library(GenomicAlignments)
  })
  gal <- readGAlignments(fl, param = param)
  sum(countOverlaps(gr, gal))
}
```

Define a 2-worker SOCK Snow cluster.

```
snow <- SnowParam(workers = 2, type = "SOCK")
```

A call to `bplapply` with the *SnowParam* creates the cluster and distributes the work.

```
bplapply(fls[1:3], FUN, BPPARAM = snow, param = param, gr = gr)
```

```
## $ERR127306
## [1] 1185
##
## $ERR127307
```

```
## [1] 1123
##
## $ERR127308
## [1] 1241
```

The FUN written for the cluster adds some overhead due to the passing of the GRanges and the loading of *GenomicAlignments* on each worker. This approach, however, has the advantage that it works on most platforms and does not require a coding change when switching between windows and non-windows machines.

If several `bplapply()` statements are likely to require the same resource, it often makes sense to create a cluster once using `bpstart()`. The workers are re-used by each call to `bplapply()`, so they do not have to re-load packages, etc.

```
register(SnowParam())  # default evaluation
bpstart()              # start the cluster
...
bplapply(X, FUN1, ...)
...
bplapply(X, FUN2, ...)  # re-use workers
...
bpstop()
```

4.2 *Ad hoc* cluster of multiple machines

We use the term *ad hoc* cluster to define a group of machines that can communicate with each other and to which the user has password-less log-in access. This example uses a group of compute machines ("the rhinos") on the FHCRC network.

4.2.1 Sockets

On Linux and Mac OS X, a socket cluster is created across machines by supplying machine names as the `workers` argument to a *BiocParallelParam* instance instead of a number. Each name represents an *R* process; repeat names indicate multiple workers on the same machine.

Create a *SnowParam* with 2 cpus from 'rhino01' and 1 from 'rhino02'.

```
hosts <- c("rhino01", "rhino01", "rhino02")
param <- SnowParam(workers = hosts, type = "SOCK")
```

Execute FUN 4 times across the workers.

```
> FUN <- function(i) system("hostname", intern=TRUE)
> bplapply(1:4, FUN, BPPARAM = param)
[[1]]
[1] "rhino01"

[[2]]
[1] "rhino01"

[[3]]
[1] "rhino02"
```

```
[[4]]
[1] "rhino01"
```

When creating a cluster across Windows machines `workers` must be IP addresses (e.g., "140.107.218.57") instead of machine names.

4.2.2 MPI

An MPI cluster across machines is created with *mpirun* or *mpiexec* from the command line or a script. A list of machine names provided as the -hostfile argument defines the mpi universe.

The hostfile requests 2 processors on 3 different machines.

```
rhino01 slots=2
rhino02 slots=2
rhino03 slots=2
```

From the command line, start a single interactive *R* process on the current machine.

```
mpiexec --np 1 --hostfile hostfile R --vanilla
```

Load *BiocParallel* and create an MPI Snow cluster. The number of `workers` in *SnowParam* should match the number of slots requested in the hostfile. Using a smaller number of workers uses a subset of the slots.

```
> library(BiocParallel)
> param <- SnowParam(workers = 6, type = "MPI")
```

Execute FUN 6 times across the workers.

```
> FUN <- function(i) system("hostname", intern=TRUE)
> bplapply(1:6, FUN, BPPARAM = param)
bplapply(1:6, FUN, BPPARAM = param)
        6 slaves are spawned successfully. 0 failed.
[[1]]
[1] "rhino01"

[[2]]
[1] "rhino02"

[[3]]
[1] "rhino02"

[[4]]
[1] "rhino03"

[[5]]
[1] "rhino03"

[[6]]
[1] "rhino01"
```

Batch jobs can be launched with mpiexec and R CMD BATCH. Code to be executed is in 'Rcode.R'.

```
mpiexec --hostfile hostfile R CMD BATCH Rcode.R
```

4.3 Clusters with schedulers

Computer clusters are far from standardized, so the following may require significant adaptation; it is written from experience here at FHCRC, where we have a large cluster managed via SLURM. Nodes on the cluster have shared disks and common system images, minimizing complexity about making data resources available to individual nodes. There are two simple models for use of the cluster, Cluster-centric and R-centric.

4.3.1 Cluster-centric

The idea is to use cluster management software to allocate resources, and then arrange for an *R* script to be evaluated in the context of allocated resources. NOTE: Depending on your cluster configuration it may be necessary to add a line to the template file instructing workers to use the version of R on the master / head node. Otherwise the default R on the worker nodes will be used.

For SLURM, we might request space for 4 tasks (with `salloc` or `sbatch`), arrange to start the MPI environment (with `orterun`) and on a single node in that universe run an *R* script `BiocParallel-MPI.R`. The command is

```
$ salloc -N 4 orterun -n 1 R -f BiocParallel-MPI.R
```

The *R* script might do the following, using MPI for parallel evaluation. Start by loading necessary packages and defining `FUN` work to be done

```
library(BiocParallel)
library(Rmpi)
FUN <- function(i) system("hostname", intern=TRUE)
```

Create a *SnowParam* instance with the number of nodes equal to the size of the MPI universe minus 1 (let one node dispatch jobs to workers), and register this instance as the default

```
param <- SnowParam(mpi.universe.size() - 1, "MPI")
register(param)
```

Evaluate the work in parallel, process the results, clean up, and quit

```
xx <- bplapply(1:100, FUN)
table(unlist(xx))
mpi.quit()
```

The entire session is as follows:

```
$ salloc -N 4 orterun -n 1 R --vanilla -f BiocParallel-MPI.R
salloc: Job is in held state, pending scheduler release
salloc: Pending job allocation 6762292
salloc: job 6762292 queued and waiting for resources
salloc: job 6762292 has been allocated resources
salloc: Granted job allocation 6762292
## ...
> FUN <- function(i) system("hostname", intern=TRUE)
>
> library(BiocParallel)
> library(Rmpi)
> param <- SnowParam(mpi.universe.size() - 1, "MPI")
```

```
> register(param)
> xx <- bplapply(1:100, FUN)
        4 slaves are spawned successfully. 0 failed.
> table(unlist(xx))

gizmof13 gizmof71 gizmof86 gizmof88
      25       25       25       25
>
> mpi.quit()
salloc: Relinquishing job allocation 6762292
salloc: Job allocation 6762292 has been revoked.
```

One advantage of this approach is that the responsibility for managing the cluster lies firmly with the cluster management software – if one wants more nodes, or needs special resources, then adjust parameters to `salloc` (or `sbatch`).

Notice that workers are spawned within the `bplapply` function; it might often make sense to more explicitly manage workers with `bpstart` and `bpstop`, e.g.,

```
param <- bpstart(SnowParam(mpi.universe.size() - 1, "MPI"))
register(param)
xx <- bplapply(1:100, FUN)
bpstop(param)
mpi.quit()
```

4.3.2 R-centric

A more *R*-centric approach might start an *R* script on the head node, and use *batchtools* to submit jobs from within the *R* session. One way of doing this is to create a file containing a template for the job submission step, e.g., for SLURM; a starting point might be found at

```
tmpl <- system.file(package="batchtools", "templates", "slurm-simple.tmpl")
noquote(readLines(tmpl))
```

```
##  [1] #!/bin/bash
##  [2]
##  [3] ## Job Resource Interface Definition
##  [4] ##
##  [5] ## ntasks [integer(1)]:       Number of required tasks,
##  [6] ##                           Set larger than 1 if you want to further parallelize
##  [7] ##                           with MPI within your job.
##  [8] ## ncpus [integer(1)]:        Number of required cpus per task,
##  [9] ##                           Set larger than 1 if you want to further parallelize
## [10] ##                           with multicore/parallel within each task.
## [11] ## walltime [integer(1)]:     Walltime for this job, in seconds.
## [12] ##                           Must be at least 60 seconds for Slurm to work properly.
## [13] ## memory   [integer(1)]:     Memory in megabytes for each cpu.
## [14] ##                           Must be at least 100 (when I tried lower values my
## [15] ##                           jobs did not start at all).
## [16] ##
## [17] ## Default resources can be set in your .batchtools.conf.R by defining the variable
## [18] ## 'default.resources' as a named list.
```

```
## [19]
## [20] <%
## [21] # relative paths are not handled well by Slurm
## [22] log.file = fs::path_expand(log.file)
## [23] -%>
## [24]
## [25]
## [26] #SBATCH --job-name=<%= job.name %>
## [27] #SBATCH --output=<%= log.file %>
## [28] #SBATCH --error=<%= log.file %>
## [29] #SBATCH --time=<%= ceiling(resources$walltime / 60) %>
## [30] #SBATCH --ntasks=1
## [31] #SBATCH --cpus-per-task=<%= resources$ncpus %>
## [32] #SBATCH --mem-per-cpu=<%= resources$memory %>
## [33] <%= if (!is.null(resources$partition)) sprintf(paste0("#SBATCH --partition='", resources$partition, "
## [34] <%= if (array.jobs) sprintf("#SBATCH --array=1-%i", nrow(jobs)) else "" %>
## [35]
## [36] ## Initialize work environment like
## [37] ## source /etc/profile
## [38] ## module add ...
## [39]
## [40] ## Export value of DEBUGME environemnt var to slave
## [41] export DEBUGME=<%= Sys.getenv("DEBUGME") %>
## [42]
## [43] <%= sprintf("export OMP_NUM_THREADS=%i", resources$omp.threads) -%>
## [44] <%= sprintf("export OPENBLAS_NUM_THREADS=%i", resources$blas.threads) -%>
## [45] <%= sprintf("export MKL_NUM_THREADS=%i", resources$blas.threads) -%>
## [46]
## [47] ## Run R:
## [48] ## we merge R output with stdout from SLURM, which gets then logged via --output option
## [49] Rscript -e 'batchtools::doJobCollection("<%= uri %>")'
```

The *R* script, run interactively or from the command line, might then look like

```
## define work to be done
FUN <- function(i) system("hostname", intern=TRUE)

library(BiocParallel)

## register SLURM cluster instructions from the template file
param <- BatchtoolsParam(workers=5, cluster="slurm", template=tmpl)
register(param)

## do work
xx <- bplapply(1:100, FUN)
table(unlist(xx))
```

The code runs on the head node until `bplapply`, where the *R* script interacts with the SLURM scheduler to request a SLURM allocation, run jobs, and retrieve results. The argument `4` to `BatchtoolsParam` specifies the number of workers to request from the scheduler;

`bplapply` divides the 100 jobs among the 4 workers. If `BatchtoolsParam` had been created without specifying any workers, then 100 jobs implied by the argument to `bplapply` would be associated with 100 tasks submitted to the scheduler.

Because cluster tasks are running in independent *R* instances, and often on physically separate machines, a convenient 'best practice' is to write `FUN` in a 'functional programming' manner, such that all data required for the function is passed in as arguments or (for large data) loaded implicitly or explicitly (e.g., via an *R* library) from disk.

5 Analyzing genomic data in *Bioconductor*

General strategies exist for handling large genomic data that are well suited to *R* programs. A manuscript titled *Scalable Genomics with R and Bioconductor* (http://arxiv.org/abs/1409.2864) by Michael Lawrence and Martin Morgan, reviews several of these approaches and demonstrate implementation with *Bioconductor* packages. Problem areas include scalable processing, summarization and visualization. The techniques presented include restricting queries, compressing data, iterating, and parallel computing.

Ideas are presented in an approachable fashion within a framework of common use cases. This is a benificial read for anyone anyone tackling genomics problems in *R*.

6 For developers

Developers wishing to use *BiocParallel* in their own packages should include *BiocParallel* in the `DESCRIPTION` file

```
Imports: BiocParallel
```

and import the functions they wish to use in the `NAMESPACE` file, e.g.,

```
importFrom(BiocParallel, bplapply)
```

Then invoke the desired function in the code, e.g.,

```
system.time(x <- bplapply(1:3, function(i) { Sys.sleep(i); i }))
```

```
##    user  system elapsed
##   0.058   0.103   3.083
```

```
unlist(x)
```

```
## [1] 1 2 3
```

This will use the back-end returned by `bpparam()`, by default a `MulticoreParam()` instance or the user's preferred back-end if they have used `register()`. The `MulticoreParam` back-end does not require any special configuration or set-up and is therefore the safest option for developers. Unfortunately, `MulticoreParam` provides only serial evaluation on Windows.

Developers should document that their function uses *BiocParallel* functions on the man page, and should perhaps include in their function signature an argument `BPPARAM=bpparam()`. Developers should NOT use 'register()' in package code – this sets a preference that influences use of 'bplapply()' and friends in all packages, not just their package.

Developers wishing to invoke back-ends other than `MulticoreParam`, or to write code that works across Windows, macOS and Linux, need to take special care to ensure that required packages, data, and functions are available and loaded on the remote nodes.

In `bplapply()`, the environment of `FUN` (other than the global environment) is serialized to the workers. A consequence is that, when `FUN` is inside a package name space, other functions available in the name space are available to `FUN` on the workers.

7 sessionInfo()

```
toLatex(sessionInfo())
```

- R version 4.1.1 (2021-08-10), `x86_64-pc-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_GB`, `LC_COLLATE=C`, `LC_MONETARY=en_US.UTF-8`, `LC_MESSAGES=en_US.UTF-8`, `LC_PAPER=en_US.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_US.UTF-8`, `LC_IDENTIFICATION=C`
- Running under: `Ubuntu 20.04.3 LTS`
- Matrix products: default
- BLAS: `/home/biocbuild/bbs-3.14-bioc/R/lib/libRblas.so`
- LAPACK: `/home/biocbuild/bbs-3.14-bioc/R/lib/libRlapack.so`
- Base packages: base, datasets, grDevices, graphics, methods, stats, stats4, utils
- Other packages: AnnotationDbi 1.55.1, Biobase 2.53.0, BiocGenerics 0.39.2, BiocParallel 1.27.9, Biostrings 2.61.2, GenomeInfoDb 1.29.8, GenomicAlignments 1.29.0, GenomicFeatures 1.45.2, GenomicRanges 1.45.0, IRanges 2.27.2, MatrixGenerics 1.5.4, RNAseqData.HNRNPC.bam.chr14 0.31.0, Rsamtools 2.9.1, S4Vectors 0.31.3, SummarizedExperiment 1.23.4, TxDb.Hsapiens.UCSC.hg19.knownGene 3.2.2, VariantAnnotation 1.39.0, XVector 0.33.0, matrixStats 0.61.0
- Loaded via a namespace (and not attached): BSgenome 1.61.0, BiocFileCache 2.1.1, BiocIO 1.3.0, BiocManager 1.30.16, BiocStyle 2.21.3, DBI 1.1.1, DelayedArray 0.19.3, GenomeInfoDbData 1.2.6, KEGGREST 1.33.0, Matrix 1.3-4, R6 2.5.1, RCurl 1.98-1.5, RSQLite 2.2.8, Rcpp 1.0.7, XML 3.99-0.8, assertthat 0.2.1, backports 1.2.1, base64url 1.4, batchtools 0.9.15, biomaRt 2.49.4, bit 4.0.4, bit64 4.0.5, bitops 1.0-7, blob 1.2.2, brew 1.0-6, cachem 1.0.6, checkmate 2.0.0, compiler 4.1.1, crayon 1.4.1, curl 4.3.2, data.table 1.14.0, dbplyr 2.1.1, debugme 1.1.0, digest 0.6.27, dplyr 1.0.7, ellipsis 0.3.2, evaluate 0.14, fansi 0.5.0, fastmap 1.1.0, filelock 1.0.2, fs 1.5.0, generics 0.1.0, glue 1.4.2, grid 4.1.1, highr 0.9, hms 1.1.0, htmltools 0.5.2, httr 1.4.2, knitr 1.34, lattice 0.20-44, lifecycle 1.0.0, magrittr 2.0.1, memoise 2.0.0, parallel 4.1.1, pillar 1.6.2, pkgconfig 2.0.3, png 0.1-7, prettyunits 1.1.1, progress 1.2.2, purrr 0.3.4, rappdirs 0.3.3, restfulr 0.0.13, rjson 0.2.20, rlang 0.4.11, rmarkdown 2.11, rtracklayer 1.53.1, snow 0.4-3, stringi 1.7.4, stringr 1.4.0, tibble 3.1.4, tidyselect 1.1.1, tools 4.1.1, utf8 1.2.2, vctrs 0.3.8, withr 2.4.2, xfun 0.26, xml2 1.3.2, yaml 2.2.1, zlibbioc 1.39.0