Introduction to *BiocParallel*

Vincent Carey, Michael Lawrence, Martin Morgan[*]

Edited: February 16, 2014; Compiled: May 22, 2014

1 Introduction

The *BiocParallel* package provides a consistent way of specifying parallel evaluation within *Bioconductor*. Enable its use by attaching the package

```
library(BiocParallel)
```

To use, invoke a *BiocParallel* function like bplapply, or use a *BiocParallel*-enabled function provided by another package.

1.1 Why use BiocParallel?

The Task View document at http://cran.r-project.org/web/views/HighPerformanceComputing.html shows that numerous approaches to parallel computing are available in *R*. Most applications cited in the task view identify one or more of *snow*, *Rmpi* or *foreach* as relevant parallelization infrastructure.

A basic objective of *BiocParallel* is reduction of complexity faced by developers and users in creating and using software that benefits from performing computations in parallel. This is accomplished by defining abstractions of the key components of parallel computing environments. Information on the parallel environment can be stored in formally structured "parameter" arguments that are used at run time to define the approach to parallel execution. This allows developers to focus on *what* is to be computed, leaving the *how* to the infrastructure.

Advantages for developers and power users of *BiocParallel* over *ad hoc R* programming for parallel computation include the following.

- A uniform idiom (using `BiocParallelParam` instances) is available for defining parallel computing resources; sensible default definitions are generated when *BiocParallel* is loaded.
- `bplapply` and `bpvec` address iteration in parallel and parallel evaluation of vectorized functions respectively.
- When the parallel environment is managed by a cluster scheduler through *BatchJobs*, job management and result retrieval are considerably simplified.
- `foreach` and programming with the *iterators* package are fully supported, but registration of the parallel back end uses `BiocParallelParam` instances.

2 The BiocParallel Interface

The *BiocParallel* work flow is simple:

1. Invoke *BiocParallel*-enabled functions. The functions use the registered back-ends for evaluation.

An optional step is to register appropriate back-ends for your particular configuration by

[*]mtmorgan@fhcrc.org

1. Creating a `BiocParallelParam` instance to describe how parallel evaluation is to be implemented.
2. Registering the `BiocParallelParam` instance for use in your *R* session.

The registry is a 'stack', with the last entry added to the stack used first, so your own back-ends generally take precedence over the back-ends established when the *BiocParallel* package is loaded.

2.1 *Param objects to describe parallel evaluation environments

Different types of parallel computation are supported by creating and `register()`ing a '`Param`'. Supported `Param` objects are:

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

`MulticoreParam` 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. Unfortunately, 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()`.

`SnowParam` Evaluate *BiocParallel*-enabled code across several distinct *R* instances, on one or several computers. This can be an easy way to parallelize code when working with 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.

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

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

The simplest illustration of creating `BiocParallelParam` is

```r
serialParam <- SerialParam()
serialParam

## class: SerialParam; bpisup: TRUE; bpworkers: 1; catch.errors: TRUE
```

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

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

## class: MulticoreParam; bpisup: TRUE; bpworkers: 8; catch.errors: TRUE
## setSeed: TRUE; recursive: TRUE; cleanup: TRUE; cleanupSignal: 15; verbose: FALSE
```

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

2.2 `register()`ing `BiocParallelParam` instances

The `register()` function registers a `BiocParallelParam` instance for use in parallel evaluation.

```r
register(multicoreParam)
```

View registered parameters with `registered()`

```r
registered()

## $MulticoreParam
## class: MulticoreParam; bpisup: TRUE; bpworkers: 8; catch.errors: TRUE
## setSeed: TRUE; recursive: TRUE; cleanup: TRUE; cleanupSignal: 15; verbose: FALSE
##
## $SnowParam
## class: SnowParam; bpisup: FALSE; bpworkers: 6; catch.errors: TRUE
## cluster spec: 6; type: PSOCK
##
## $BatchJobsParam
## class: BatchJobsParam; bpisup: TRUE; bpworkers: NA; catch.errors: TRUE
## cleanup: TRUE; stop.on.error: FALSE; progressbar: TRUE
##
## $SerialParam
## class: SerialParam; bpisup: TRUE; bpworkers: 1; catch.errors: TRUE
```

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.

2.3 Functions for parallel computation

There are facilities for querying and controlling parallel evaluation environments.

`bpisup(x)` Query a `BiocParallelParam` back-end `x` for its status.
`bpworkers` Query a `BiocParallelParam` back-end for the number of workers available for parallel evaluation.
`bpstart(x)` Start a parallel back end specified by `BiocParallelParam` `x`, if possible.
`bpstop(x)` Stop a parallel back end specified by `BiocParallelParam` `x`.

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 ....
`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`.

There are facilities for recovering from errors

`bplasterror` Report the last error reported from a *BiocParallel* evaluation.
`bpresume` Attempt to resume computation after an error.

3 Use cases

3.1 Single computer

3.2 Ad hoc clusters

3.3 Clusters with schedulers

4 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
##   0.008   0.008   3.013

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 wishing to invoke back-ends other than MulticoreParam need to take special care to ensure that required packages, data, and functions are available and loaded on the remote nodes.