Errors, Logs and Debugging in *BiocParallel*

Valerie Obenchain and Martin Morgan

Edited: May 13, 2015; Compiled: September 29, 2015

Contents

1 Introduction

This vignette is part of the *BiocParallel* package and focuses on error handling and logging. A section at the end demonstrates how the two can be used together as part of an effective debugging routine.

BiocParallel provides a unified interface to the parallel infrastructure in several packages including *snow*, *parallel*, *BatchJobs* and *foreach*. When implementing error handling in *BiocParallel* the primary goals were to enable the return of partial results when an error is thrown (vs just the error) and to establish logging on the workers. In cases where error handling existed, such as *BatchJobs* and *foreach*, those behaviors were preserved. Clusters created with *snow* and *parallel* now have flexible error handling and logging available through `SnowParam` and `MulticoreParam` objects.

In this document the term "job" is used to describe a single call to a bp*apply function (e.g., the `X` in `bplapply`). A "job" consists of one or more "tasks", where each "task" is run separately on a worker.

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

```
source("http://bioconductor.org/biocLite.R")
biocLite("BiocParallel")
```

Load the package:

```
library(BiocParallel)
```

2 Error Handling

2.1 Catching errors

By default, *BiocParallel* attempts all computations and returns any warnings and errors along with successful results. The `stop.on.error` field controls if the job is terminated as soon as one task throws an error. This is useful when debugging or when running large jobs (many tasks) and you want to be notified of an error before all runs complete.

`stop.on.error` is `FALSE` by default.

```
param <- SnowParam()
param
```

The field can be set when constructing the param or modified with the `bpstopOnError` accessor.

```
param <- SnowParam(2, stop.on.error = TRUE)
param
bpstopOnError(param) <- FALSE
```

In this example `X` is length 6. By default, the elements of `X` are divided as evenly as possible over the number of workers and run in chunks. To more clearly demonstrate the affect of `stop.on.error` the number of tasks is set equal to the length of `X`. This forces each element of `X` to be executed separately (6 tasks) vs chunked.

```
X <- list(1, 2, "3", 4, 5, 6)
param <- SnowParam(3, tasks = length(X), stop.on.error = TRUE)
```

The output list contains results for tasks 1 and 2 and an error for task 3. Tasks 4, 5, and 6 are not attempted.

```
bplapply(X, sqrt, BPPARAM = param)
```

Next we look at an example where the elements of `X` are grouped instead of run separately. The default value for `tasks` is 0 which means 'X' is split as evenly as possible across the number of workers. There are 3 workers so the first task consists of list(1, 2), the second is list("3", 4) and the third is list(5, 6).

```
X <- list(1, 2, "3", 4, 5, 6)
param <- SnowParam(3, stop.on.error = TRUE)
```

To simulate a longer running computation sleep time is added to 'FUN'. The sleep forces task 2 to finish before task 3.

```
FUN <- function(i) { Sys.sleep(i); sqrt(i) }
```

The output shows an error in task 2 (vs 3 in the previous example) and a result for '4' is included because it was part of the second task.

```
bplapply(X, FUN, BPPARAM = param)
```

Side Note: Results are collected from workers as they finish which is not necessarily the same order in which they were loaded. Depending on how tasks are divided among workers it is possible that the task with an error completes after all others. In that situation the output will includes all results along with the error message and it may appear that `stop.on.error` is not doing much good. This is simply a heads up that the usefulness of `stop.on.error` may vary with run time and distribution of tasks over workers.

2.2 Identify failures with `bpok()`

The `bpok()` function is a quick way to determine which (if any) tasks failed. In this example the second element fails.

```
res <- bplapply(list(1, "2", 3), sqrt)
res
```

bpok returns TRUE if the task was successful.

```
bpok(res)
```

Once errors are identified with `bpok` the traceback can be retrieved with the `attr` function. This is possible because errors are returned as `condition` objects with the traceback as an attribute.

```
fail <- !bpok(res)
tail(attr(res[[2]], "traceback"))
```

2.3 Rerun failed tasks with `BPREDO`

Tasks can fail due to hardware problems or bugs in the input data. The *BiocParallel* functions support a `BPREDO` (re-do) argument for recomputing only the tasks that failed. A list of partial results and errors is supplied to `BPREDO` in a second call to the function. The failed elements are identified, recomputed and inserted into the original results.

The bug in this example is the second element of 'X' which is a character when it should be numeric.

```
X <- list(1, "2", 3)
res <- bplapply(X, sqrt)
res
```

First fix the input data.

```
X.redo <- list(1, 2, 3)
```

Repeat the call to `bplapply` this time supplying the partial results as `BPREDO`.

```
bplapply(X.redo, sqrt, BPREDO = res)
```

3 Logging

NOTE: Logging as described in this section is supported for SnowParam, MulticoreParam and SerialParam.

3.1 Parameters

Logging in *BiocParallel* is controlled by 3 fields in the `BiocParallelParam`:

```
log:       TRUE or FALSE
logdir:    location to write log file
threshold: one of "TRACE", "DEBUG", "INFO", "WARN", "ERROR", "FATAL"
```

When `log = TRUE` the *futile.logger* package is loaded on each worker. *BiocParallel* uses a custom script on the workers to collect log messages as well as additional statistics such as gc, runtime and node information. Output to stderr and stdout is also captured.

By default `log` is FALSE and `threshold` is *INFO*.

```
param <- SnowParam()
param
```

Turn logging on and set the threshold to *TRACE*.

```
bplog(param) <- TRUE
bpthreshold(param) <- "TRACE"
param
```

3.2 Setting a threshold

All thresholds defined in *futile.logger* are supported: *FATAL*, *ERROR*, *WARN*, *INFO*, *DEBUG* and *TRACE*. All messages greater than or equal to the severity of the threshold are shown. For example, a threshold of *INFO* will print all messages tagged as *FATAL*, *ERROR*, *WARN* and *INFO*.

In this code chunk an *INFO*-level message is emitted when *futile.logger* is loaded on the workers and a *ERROR*-level message when attempting the square root of a character ("2").

```
bpthreshold(param) <- "INFO"
bplapply(list(1, "2", 3), sqrt, BPPARAM = param)
```

All user-supplied messages written in the *futile.logger* syntax are also captured. This function performs argument checking and includes a couple of *WARN* and *DEBUG*-level messages.

```
FUN <- function(i) {
  flog.debug(paste0("value of 'i': ", i))

  if (!length(i)) {
      flog.warn("'i' is missing")
      NA
  } else if (!is(i, "numeric")) {
      flog.info("coercing to numeric")
      as.numeric(i)
  } else {
      i
  }
}
```

Turn logging on and set the threshold to *WARN*.

```
param <- SnowParam(2, log = TRUE, threshold = "WARN")
bplapply(list(1, "2", integer()), FUN, BPPARAM = param)
```

Changing the threshold to *DEBUG* catches all *WARN*, *INFO* and *DEBUG* messages.

```
param <- SnowParam(2, log = TRUE, threshold = "DEBUG")
bplapply(list(1, "2", integer()), FUN, BPPARAM = param)
```

3.3 Log files

When `log == TRUE`, log messages are written to the console by default. If `logdir` is given the output is written out to files, one per task. File names are prefixed with the name in `bpjobname(BPPARAM)`; default is 'BPJOB'.

```
> param <- SnowParam(2, log = TRUE, threshold = "DEBUG", logdir = tempdir())
> res <- bplapply(list(1, "2", integer()), FUN, BPPARAM = param)
INFO [2015-07-08 08:47:40] loading futile.logger on workers
> list.files(bplogdir(param))
[1] "BPJOB.task1.log" "BPJOB.task2.log"
```

Read in BPJOB.task2.log:

```
> readLines(paste0(bplogdir(param), "/BPJOB.task2.log"))
 [1] "################ LOG OUTPUT ################"
 [2] "Task: 2"
 [3] "Node: 2"
 [4] "Timestamp: 2015-07-08 09:03:59"
 [5] "Success: TRUE"
 [6] "Task duration: "
 [7] "   user  system elapsed "
 [8] "  0.009   0.000   0.011 "
 [9] "Memory use (gc): "
[10] "         used (Mb) gc trigger (Mb) max used (Mb)"
[11] "Ncells 325664 17.4     592000 31.7   393522 21.1"
[12] "Vcells 436181  3.4    1023718  7.9   530425  4.1"
[13] "Log messages:"
[14] "DEBUG [2015-07-08 09:03:59] value of 'i': 2"
[15] "INFO [2015-07-08 09:03:59] coercing to numeric"
[16] "DEBUG [2015-07-08 09:03:59] value of 'i': "
[17] "WARN [2015-07-08 09:03:59] 'i' is missing"
[18] ""
[19] "stderr and stdout:"
[20] "character(0)"
```

4 Worker timeout

NOTE: `timeout` is supported for SnowParam and MulticoreParam.

For long running jobs or untested code it can be useful to set a time limit. The `timeout` field is the time, in seconds, allowed for each worker to complete a task; default is `Inf`. If the task takes longer than `timeout` a timeout error is returned.

Time can be changed during param construction with the `timeout` arg,

```
param <- SnowParam(timeout = 20)
param
```

or with the `bptimeout` setter:

```
bptimeout(param) <- 2
param
```

A timeout error is returned if

```
fun <- function(i) {
  Sys.sleep(i)
  i
}
bplapply(1:3, fun, BPPARAM = param)
```

5 Debugging

Effective debugging strategies vary by problem and often involve a combination of error handling and logging techniques. In general, when debugging *R*-generated errors the traceback is often the best place to start followed by adding debug

messages to the worker function. When trouble shooting unexpected behavior (i.e., not a formal error or warning) adding debug messages or switching to `SerialParam` are good approaches. Below is an overview of these different strategies.

5.1 Accessing the traceback

The traceback is a good place to start when tracking down *R*-generated errors. Because the function is executed on the workers it's not accessible for interactive debugging with functions such as `trace` or `debug`. The traceback provides a snapshot of the state of the worker at the time the error was thrown.

This function takes the square root of the absolute value of a vector.

```
fun1 <- function(x) {
    v <- abs(x)
    sapply(1:length(v), function(i) sqrt(v[i]))
}
```

Calling "fun1" with a character throws an error:

```
res <- bplapply(list(c(1,3), 5, "6"), fun1)
res
```

Identify which elements failed with `bpok`:

```
bpok(res)
```

The error (i.e., third element of "res") is a `condition` object:

```
is(res[[3]], "condition")
```

The traceback is an attribute of the `condition` and can be accessed with the `attr` function.

```
noquote(tail(attr(res[[3]], "traceback")))
```

5.2 Adding debug messages

When a `numeric()` is passed to "fun1" no formal error is thrown but the length of the second list element is 2 when it should be 1.

```
bplapply(list(c(1,3), numeric(), 6), fun1)
```

Without a formal error we have no traceback so we'll try adding a few debug messages to "fun1". The *futile.logger* syntax tags messages with different levels of severity. A message created with `flog.debug` will only print if the threshold is *DEBUG* or lower.

"fun2" has debug statements that show the value of 'x', length of 'v' and the index 'i'.

```
fun2 <- function(x) {
    v <- abs(x)
    flog.debug(
      paste0("'x' = ", paste(x, collapse=","), ": length(v) = ", length(v))
    )
    sapply(1:length(v), function(i) {
      flog.debug(paste0("'i': ", i))
      sqrt(v[i])
    })
}
```

Create a param that logs at a threshold level of *DEBUG*.

```
param <- SnowParam(3, log = TRUE, threshold = "DEBUG")
```

The debug messages reveal the problem occurs when 'x' is `numeric()`. The index for `sapply` is along 'v' which in this case has length 0. This forces 'i' to take values of '1' and '0' giving an output of length 2 for the second element (i.e., `NA` and `numeric(0)`).

```
param <- SnowParam(3, log = TRUE, threshold = "DEBUG")
res <- bplapply(list(c(1,3), numeric(), 6), fun2, BPPARAM = param)
res
```

"fun2" can be fixed by using `seq_along(v)` to create the index instead of `1:length(v)`.

5.3 Local debugging with `SerialParam`

Errors that occur on parallel workers can be difficult to debug. Often the traceback sent back from the workers is too much to parse or not informative. We are also limited in that our interactive strategies of `browser` and `trace` are not available.

One option for further debugging is to run the code in serial with `SerialParam`. This removes the "parallel" component and is the same as running a straight `*apply` function. This approach may not help if the problem was hardware related but can be very useful when the bug is in the *R* code.

We use the now familiar square root example with a bug in the second element of `X`.

```
res <- bplapply(list(1, "2", 3), sqrt, BPPARAM = SnowParam(3))
res
```

`sqrt` is an internal function. The problem is likely with our data going into the function and not the `sqrt` function itself. We can write a small wrapper around `sqrt` so we can see the input.

```
fun3 <- function(i) sqrt(i)
```

Debug the new function:

```
debug(fun3)
```

We want to recompute only elements that failed and for that we use `BPREDO`. The BPPARAM has been changed to `SerialParam` so the job is run locally (in the workspace) and in serial.

```
> bplapply(list(1, "2", 3), fun3, BPREDO = res, BPPARAM = SerialParam())
Resuming previous calculation ...
debugging in: FUN(...)
debug: sqrt(i)
Browse[2]> objects()
[1] "i"
Browse[2]> i
[1] "2"
Browse[2]>
```

The local browsing allowed us to see the problematic input which was the character "2".

6 `sessionInfo()`

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

```
## \begin{itemize}\raggedright
##   \item R version 3.2.2 Patched (2015-08-14 r69078), \verb|x86_64-apple-darwin10.8.0|
```

```
##   \item Locale: \verb|en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8|
##   \item Base packages: base, datasets, graphics, grDevices, methods, stats,
##     utils
##   \item Other packages: BiocParallel~1.2.22
##   \item Loaded via a namespace (and not attached): BiocStyle~1.6.0,
##     evaluate~0.8, formatR~1.2.1, futile.logger~1.4.1, futile.options~1.0.0,
##     highr~0.5.1, knitr~1.11, lambda.r~1.1.7, magrittr~1.5, parallel~3.2.2,
##     stringi~0.5-5, stringr~1.0.0, tools~3.2.2
## \end{itemize}
```