Errors, Logs and Debugging in *BiocParallel*

Valerie Obenchain and Martin Morgan

Edited: December 16, 2015; Compiled: November 6, 2020

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*, *batchtools* 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 *batchtools* 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 `BiocManager::install`:

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

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

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

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

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

```
## class: SnowParam
##   bpisup: FALSE; bpnworkers: 2; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
```

```
##   bpexportglobals: TRUE
##   bplogdir: NA
##   bpresultdir: NA
##   cluster type: SOCK

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. The number of tasks is set equal to the length of `X` which forces each element of `X` to be executed separately (6 tasks).

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

Tasks 1, 2, and 3 are assigned to the three workers, and are evaluated. Task 2 fails, stopping further computation. All successfully completed tasks are returned as the 'result' attribute. Usually, this means that the results of tasks 1, 2, and 3 will be returned.

```
res <- tryCatch({
    bplapply(X, sqrt, BPPARAM = param)
}, error=identity)
res

## <bplist_error: BiocParallel errors
##   element index: 2
##   first error: non-numeric argument to mathematical function>
## results and errors available as 'attr(x, "result")'

attr(res, "result")

## [[1]]
## [1] 1
##
## [[2]]
## <remote_error in FUN(...): non-numeric argument to mathematical function>
## traceback() available as 'attr(x, "traceback")'
##
## [[3]]
## [1] 1.732051
##
## [[4]]
## [1] 2
##
## [[5]]
## [1] 2.236068
```

Using `stop.on.error=FALSE`, all tasks are evaluated.

```
X <- list("1", 2, 3, 4, 5, 6)
param <- SnowParam(3, tasks = length(X), stop.on.error = FALSE)
res <- tryCatch({
    bplapply(X, sqrt, BPPARAM = param)
}, error=identity)
res
```

```
## <bplist_error: BiocParallel errors
##   element index: 1
##   first error: non-numeric argument to mathematical function>
## results and errors available as 'attr(x, "result")'
```

```
attr(res, "result")
```

```
## [[1]]
## <remote_error in FUN(...): non-numeric argument to mathematical function>
## traceback() available as 'attr(x, "traceback")'
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
##
## [[4]]
## [1] 2
##
## [[5]]
## [1] 2.236068
##
## [[6]]
## [1] 2.44949
```

`bptry()` is a convenient way of trying to evaluate a `bpapply`-like expression, returning the evaluated results without signalling an error.

```
bptry({
    bplapply(X, sqrt, BPPARAM=param)
})
```

```
## [[1]]
## <remote_error in FUN(...): non-numeric argument to mathematical function>
## traceback() available as 'attr(x, "traceback")'
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
##
## [[4]]
## [1] 2
##
## [[5]]
## [1] 2.236068
##
## [[6]]
## [1] 2.44949
```

In the next example 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)
```

The output shows an error in when evaluating the third element, but also that the fourth element, in the same chunk as 3, was not evaluated. All elements are evaluated because they were assigned to workers before the first error occurred.

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

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## <remote_error in FUN(...): non-numeric argument to mathematical function>
## traceback() available as 'attr(x, "traceback")'
##
## [[4]]
## <unevaluated_error: not evaluated due to previous error>
##
## [[5]]
## [1] 2.236068
##
## [[6]]
## [1] 2.44949
```

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 it is possible that the task with the error completes after all others so essentially all workers complete before the job is stopped. In this situation the output includes all results along with the error message and it may appear that `stop.on.error=TRUE` did not stop the job soon enough. This is just a heads up that the usefulness of `stop.on.error=TRUE` 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 we use `bptry()` to retrieve the partially evaluated expression, including the failed elements.

```
param <- SnowParam(2, stop.on.error=FALSE)
result <- bptry(bplapply(list(1, "2", 3), sqrt, BPPARAM=param))
```

`bpok` returns TRUE if the task was successful.

```
bpok(result)
```

```
## [1]  TRUE FALSE  TRUE
```

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.

```
tail(attr(result[[which(!bpok(result))]], "traceback"))
```

```
## [1] "4: tryCatch({"
## [2] "       FUN(...)"
## [3] "   }, error = handle_error)"
## [4] "3: tryCatchList(expr, classes, parentenv, handlers)"
## [5] "2: tryCatchOne(expr, names, parentenv, handlers[[1L]])"
## [6] "1: value[[3L]](cond)"
```

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)
param <- SnowParam(2, stop.on.error=FALSE)
result <- bptry(bplapply(X, sqrt, BPPARAM=param))
result
```

```
## [[1]]
## [1] 1
##
## [[2]]
## <remote_error in FUN(...): non-numeric argument to mathematical function>
## traceback() available as 'attr(x, "traceback")'
##
## [[3]]
## [1] 1.732051
```

First fix the input data.

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

Repeat the call to `bplapply` this time supplying the partial results as `BPREDO`. Only the failed calculations are computed, in the present case requiring only one worker.

```
bplapply(X.redo, sqrt, BPREDO=result, BPPARAM=param)
```

```
## [[1]]
## [1] 1
```

```
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
```

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(stop.on.error=FALSE)
param
```

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

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

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

```
## class: SnowParam
##   bpisup: FALSE; bpnworkers: 4; bptasks: 0; bpjobname: BPJOB
##   bplog: TRUE; bpthreshold: TRACE; bpstopOnError: FALSE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE
##   bplogdir: NA
```

```
##   bpresultdir: NA
##   cluster type: SOCK
```

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*.

Because the default threshold is *INFO* it catches the *ERROR*-level message thrown when attempting the square root of a character ("2").

```
tryCatch({
    bplapply(list(1, "2", 3), sqrt, BPPARAM = param)
}, error=function(e) invisible(e))
```

```
## ############### LOG OUTPUT ###############
## Task:  3
## Node:  3
## Timestamp:  2020-11-06 19:49:38
## Success:  TRUE
##
## Task duration:
##    user  system elapsed
##       0       0       0
##
## Memory used:
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  817451 43.7    1283244 68.6  1283244 68.6
## Vcells 1467219 11.2    8388608 64.0  2374948 18.2
##
## Log messages:
## INFO [2020-11-06 19:49:38] loading futile.logger package
##
## stderr and stdout:
```

```
## ############### LOG OUTPUT ###############
## Task:  1
## Node:  1
## Timestamp:  2020-11-06 19:49:38
## Success:  TRUE
##
## Task duration:
##    user  system elapsed
##       0       0       0
##
## Memory used:
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  817999 43.7    1283244 68.6  1283244 68.6
## Vcells 1468473 11.3    8388608 64.0  2374948 18.2
```

```
##
## Log messages:
## INFO [2020-11-06 19:49:38] loading futile.logger package
##
## stderr and stdout:

## ############### LOG OUTPUT ###############
## Task:  2
## Node:  2
## Timestamp:  2020-11-06 19:49:39
## Success:  FALSE
##
## Task duration:
##    user  system elapsed
##       0       0       0
##
## Memory used:
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  818034 43.7    1283244 68.6  1283244 68.6
## Vcells 1468600 11.3    8388608 64.0  2374948 18.2
##
## Log messages:
## INFO [2020-11-06 19:49:38] loading futile.logger package
## ERROR [2020-11-06 19:49:38] non-numeric argument to mathematical function
##
## stderr and stdout:
```

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) {
  futile.logger::flog.debug(paste("value of 'i':", i))

  if (!length(i)) {
      futile.logger::flog.warn("'i' has length 0")
      NA
  } else if (!is(i, "numeric")) {
      futile.logger::flog.debug("coercing 'i' to numeric")
      as.numeric(i)
  } else {
      i
  }
}
```

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

```
param <- SnowParam(2, log = TRUE, threshold = "WARN", stop.on.error=FALSE)
result <- bplapply(list(1, "2", integer()), FUN, BPPARAM = param)

## ############### LOG OUTPUT ###############
## Task:  1
## Node:  1
## Timestamp:  2020-11-06 19:49:40
```

```
## Success:  TRUE
##
## Task duration:
##    user  system elapsed
##    0.02    0.00    0.01
##
## Memory used:
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  818092 43.7    1283244 68.6  1283244 68.6
## Vcells 1469891 11.3    8388608 64.0  2374948 18.2
##
## Log messages:
##
##
## stderr and stdout:

## ############### LOG OUTPUT ###############
## Task:  2
## Node:  2
## Timestamp:  2020-11-06 19:49:40
## Success:  TRUE
##
## Task duration:
##    user  system elapsed
##    0.01    0.00    0.01
##
## Memory used:
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  818113 43.7    1283244 68.6  1283244 68.6
## Vcells 1469943 11.3    8388608 64.0  2374948 18.2
##
## Log messages:
## WARN [2020-11-06 19:49:40] 'i' has length 0
##
## stderr and stdout:

simplify2array(result)
```

Changing the threshold to *DEBUG* catches both *WARN* and *DEBUG* messages.

```
param <- SnowParam(2, log = TRUE, threshold = "DEBUG", stop.on.error=FALSE)
result <- bplapply(list(1, "2", integer()), FUN, BPPARAM = param)

## ############### LOG OUTPUT ###############
## Task:  1
## Node:  1
## Timestamp:  2020-11-06 19:49:41
## Success:  TRUE
##
## Task duration:
##    user  system elapsed
```

```
##      0      0      0
##
## Memory used:
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  818090 43.7    1283244 68.6  1283244 68.6
## Vcells 1470211 11.3    8388608 64.0  2374948 18.2
##
## Log messages:
## INFO [2020-11-06 19:49:40] loading futile.logger package
## DEBUG [2020-11-06 19:49:41] value of 'i':  1
##
## stderr and stdout:

## ############### LOG OUTPUT ###############
## Task:  2
## Node:  2
## Timestamp:  2020-11-06 19:49:41
## Success:  TRUE
##
## Task duration:
##    user  system elapsed
##    0.03    0.00    0.03
##
## Memory used:
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  818113 43.7    1283244 68.6  1283244 68.6
## Vcells 1470296 11.3    8388608 64.0  2374948 18.2
##
## Log messages:
## INFO [2020-11-06 19:49:40] loading futile.logger package
## DEBUG [2020-11-06 19:49:41] value of 'i':  2
## DEBUG [2020-11-06 19:49:41] coercing 'i' to numeric
## DEBUG [2020-11-06 19:49:41] value of 'i':
## WARN [2020-11-06 19:49:41] 'i' has length 0
##
## stderr and stdout:

simplify2array(result)
```

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)
## 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, stop.on.error=FALSE)
param
```

```
## class: SnowParam
##   bpisup: FALSE; bpnworkers: 4; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: FALSE
##   bpRNGseed: ; bptimeout: 20; bpprogressbar: FALSE
##   bpexportglobals: TRUE
##   bplogdir: NA
##   bpresultdir: NA
##   cluster type: SOCK
```

or with the `bptimeout` setter:

```
param <- SnowParam(timeout = 2, stop.on.error=FALSE)
fun <- function(i) {
  Sys.sleep(i)
  i
}
bptry(bplapply(1:3, fun, BPPARAM = param))
```

```
## [[1]]
## [1] 1
##
## [[2]]
## <remote_error in Sys.sleep(i): reached elapsed time limit>
## traceback() available as 'attr(x, "traceback")'
##
## [[3]]
## <remote_error in Sys.sleep(i): reached elapsed time limit>
## traceback() available as 'attr(x, "traceback")'
```

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:

```
param <- SnowParam(stop.on.error=FALSE)
result <- bptry({
    bplapply(list(c(1,3), 5, "6"), fun1, BPPARAM = param)
}, error=identity)
result
```

```
## [[1]]
## [1] 1.000000 1.732051
##
## [[2]]
## [1] 2.236068
##
## [[3]]
## <remote_error in abs(x): non-numeric argument to mathematical function>
## traceback() available as 'attr(x, "traceback")'
```

Identify which elements failed with `bpok`:

```
bpok(result)
```

```
## [1]  TRUE  TRUE FALSE
```

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

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

```
## [1] TRUE
```

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

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

```
## [1]         call <- sapply(sys.calls(), deparse)
## [2]         e <- structure(e, class = c("remote_error", "condition"),
## [3]             traceback = capture.output(traceback(call)))
## [4]         invokeRestart("abort", e)
## [5]     }, "non-numeric argument to mathematical function", quote(abs(x)))
## [6] 1: h(simpleError(msg, call))
```

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, BPPARAM = param)
```

```
## [[1]]
## [1] 1.000000 1.732051
##
## [[2]]
## [[2]][[1]]
## [1] NA
##
## [[2]][[2]]
## numeric(0)
##
## [[3]]
## [1] 2.44949
```

Without a formal error we have no traceback so we'll add a few debug messages. 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. So in this case it will catch both INFO and DEBUG messages.

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

```
fun2 <- function(x) {
    v <- abs(x)
    futile.logger::flog.debug(
      paste0("'x' = ", paste(x, collapse=","), ": length(v) = ", length(v))
    )
    sapply(1:length(v), function(i) {
      futile.logger::flog.info(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)`).

```
res <- bplapply(list(c(1,3), numeric(), 6), fun2, BPPARAM = param)
```

```
## ################ LOG OUTPUT ################
## Task:  3
## Node:  3
## Timestamp:  2020-11-06 19:49:46
## Success:  TRUE
##
## Task duration:
##    user  system elapsed
##    0.02    0.00    0.01
##
## Memory used:
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  818418 43.8    1283244 68.6  1283244 68.6
## Vcells 1472597 11.3    8388608 64.0  2374948 18.2
##
## Log messages:
## INFO [2020-11-06 19:49:46] loading futile.logger package
## DEBUG [2020-11-06 19:49:46] 'x' = 6:  length(v) = 1
## INFO [2020-11-06 19:49:46] 'i' = 1
##
## stderr and stdout:
```

```
## ################ LOG OUTPUT ################
## Task:  1
## Node:  1
## Timestamp:  2020-11-06 19:49:46
```

```
## Success:  TRUE
##
## Task duration:
##    user  system elapsed
##    0.02    0.00    0.01
##
## Memory used:
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  818435 43.8    1283244 68.6  1283244 68.6
## Vcells 1472660 11.3    8388608 64.0  2374948 18.2
##
## Log messages:
## INFO [2020-11-06 19:49:46] loading futile.logger package
## DEBUG [2020-11-06 19:49:46] 'x' = 1,3:  length(v) = 2
## INFO [2020-11-06 19:49:46] 'i' = 1
## INFO [2020-11-06 19:49:46] 'i' = 2
##
## stderr and stdout:

## ################ LOG OUTPUT ################
## Task:  2
## Node:  2
## Timestamp:  2020-11-06 19:49:46
## Success:  TRUE
##
## Task duration:
##    user  system elapsed
##    0.02    0.00    0.02
##
## Memory used:
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  818462 43.8    1283244 68.6  1283244 68.6
## Vcells 1472739 11.3    8388608 64.0  2374948 18.2
##
## Log messages:
## INFO [2020-11-06 19:49:46] loading futile.logger package
## DEBUG [2020-11-06 19:49:46] 'x' = :  length(v) = 0
## INFO [2020-11-06 19:49:46] 'i' = 1
## INFO [2020-11-06 19:49:46] 'i' = 0
##
## stderr and stdout:

res
```

```
## numeric(0)
##
##
## [[3]]
## [1] 2.44949
```

"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 <- bptry({
    bplapply(list(1, "2", 3), sqrt,
             BPPARAM = SnowParam(3, stop.on.error=FALSE))
})
result
```

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

`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 the `BPREDO` argument. The BPPARAM has been changed to `SerialParam` so the job is run in the local workspace in serial.

```
> bplapply(list(1, "2", 3), fun3, BPREDO = result, 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 problem input was the character "2".

6 sessionInfo()

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

- R version 4.0.3 (2020-10-10), `x86_64-w64-mingw32`
- Locale: `LC_COLLATE=C`, `LC_CTYPE=English_United States.1252`, `LC_MONETARY=English_United States.1252`, `LC_NUMERIC=C`, `LC_TIME=English_United States.1252`
- Running under: `Windows Server 2012 R2 x64 (build 9600)`
- Matrix products: default
- Base packages: base, datasets, grDevices, graphics, methods, stats, utils
- Other packages: BiocParallel 1.24.1
- Loaded via a namespace (and not attached): BiocManager 1.30.10, BiocStyle 2.18.0, R6 2.5.0, backports 1.2.0, base64url 1.4, batchtools 0.9.14, brew 1.0-6, checkmate 2.0.0, compiler 4.0.3, crayon 1.3.4, data.table 1.13.2, debugme 1.1.0, digest 0.6.27, ellipsis 0.3.1, evaluate 0.14, fs 1.5.0, highr 0.8, hms 0.5.3, htmltools 0.5.0, knitr 1.30, lifecycle 0.2.0, magrittr 1.5, parallel 4.0.3, pillar 1.4.6, pkgconfig 2.0.3, prettyunits 1.1.1, progress 1.2.2, rappdirs 0.3.1, rlang 0.4.8, rmarkdown 2.5, snow 0.4-3, stringi 1.5.3, stringr 1.4.0, tibble 3.0.4, tools 4.0.3, vctrs 0.3.4, withr 2.3.0, xfun 0.19, yaml 2.2.1