Package 'BiocParallel'

December 10, 2024

Type Package

Title Bioconductor facilities for parallel evaluation

Version 1.41.0

Description This package provides modified versions and novel implementation of functions for parallel evaluation, tailored to use with Bioconductor objects.

URL https://github.com/Bioconductor/BiocParallel

BugReports https://github.com/Bioconductor/BiocParallel/issues

biocViews Infrastructure

License GPL-2 | GPL-3

SystemRequirements C++11

Depends methods, R (>= 3.5.0)

Imports stats, utils, futile.logger, parallel, snow, codetools

Suggests BiocGenerics, tools, foreach, BBmisc, doParallel, GenomicRanges, RNAseqData.HNRNPC.bam.chr14, TxDb.Hsapiens.UCSC.hg19.knownGene, VariantAnnotation, Rsamtools, GenomicAlignments, ShortRead, RUnit, BiocStyle, knitr, batchtools, data.table

Enhances Rmpi

Collate AllGenerics.R DeveloperInterface.R prototype.R bploop.R ErrorHandling.R log.R bpbackend-methods.R bpisup-methods.R bplapply-methods.R bpiterate-methods.R bpstart-methods.R bpstop-methods.R BiocParallelParam-class.R bpmapply-methods.R bpschedule-methods.R bpvec-methods.R bpvectorize-methods.R bpworkers-methods.R bpaggregate-methods.R bpvalidate.R SnowParam-class.R MulticoreParam-class.R TransientMulticoreParam-class.R register.R SerialParam-class.R DoparParam-class.R SnowParam-utils.R BatchtoolsParam-class.R progress.R ipcmutex.R worker-number.R utilities.R rng.R bpinit.R reducer.R worker.R bpoptions.R cpp11.R BiocParallel-defunct.R

LinkingTo BH, cpp11

VignetteBuilder knitr

RoxygenNote 7.1.2

git_url https://git.bioconductor.org/packages/BiocParallel

git_branch devel

git_last_commit 0f688b6

git_last_commit_date 2024-10-29

Repository Bioconductor 3.21

Date/Publication 2024-12-10

Author Martin Morgan [aut, cre],
Jiefei Wang [aut],
Valerie Obenchain [aut],
Michel Lang [aut],
Ryan Thompson [aut],
Nitesh Turaga [aut],
Aaron Lun [ctb],
Henrik Bengtsson [ctb],
Madelyn Carlson [ctb] (Translated 'Random Numbers' vignette from Sweave to RMarkdown / HTML.),
Phylis Atieno [ctb] (Translated 'Introduction to BiocParallel' vignette from Sweave to Rmarkdown / HTML.),
Sergio Oller [ctb] (Improved bpmapply() efficiency., ORCID: <https://orcid.org/0000-0002-8994-1549>)

Maintainer Martin Morgan <mtmorgan.bioc@gmail.com>

Contents

`BiocParallel-package` *Bioconductor facilities for parallel evaluation*

Description

This package provides modified versions and novel implementation of functions for parallel evaluation, tailored to use with Bioconductor objects.

Details

This package uses code from the `parallel` package,

Author(s)

See `packageDescription("BiocParallel")`.

`BatchtoolsParam-class` *Enable parallelization on batch systems*

Description

This class is used to parameterize scheduler options on managed high-performance computing clusters using batchtools.

`BatchtoolsParam()`: Construct a BatchtoolsParam-class object.

`batchtoolsWorkers()`: Return the default number of workers for each backend.

`batchtoolsTemplate()`: Return the default template for each backend.

`batchtoolsCluster()`: Return the default cluster.

`batchtoolsRegistryargs()`: Create a list of arguments to be used in batchtools' `makeRegistry`; see `registryargs` argument.

Usage

```
BatchtoolsParam(
    workers = batchtoolsWorkers(cluster),
    cluster = batchtoolsCluster(),
    registryargs = batchtoolsRegistryargs(),
    saveregistry = FALSE,
    resources = list(),
    template = batchtoolsTemplate(cluster),
    stop.on.error = TRUE, progressbar = FALSE, RNGseed = NA_integer_,
    timeout = WORKER_TIMEOUT, exportglobals=TRUE,
    log = FALSE, logdir = NA_character_, resultdir=NA_character_,
    jobname = "BPJOB"
)
batchtoolsWorkers(cluster = batchtoolsCluster())
batchtoolsCluster(cluster)
batchtoolsTemplate(cluster)
batchtoolsRegistryargs(...)
```

Arguments

`workers` `integer(1)`

Number of workers to divide tasks (e.g., elements in the first argument of `bplapply`) between. On 'multicore' and 'socket' backends, this defaults to `multicoreWorkers()` and `snowWorkers()`. On managed (e.g., slurm, SGE) clusters `workers` has no default, meaning that the number of workers needs to be provided by the user.

`cluster` `character(1)`

Cluster type being used as the backend by `BatchtoolsParam`. The available options are "socket", "multicore", "interactive", "sge", "slurm", "lsf", "torque" and "openlava". The cluster type if available on the machine registers as the backend. Cluster types which need a `template` are "sge", "slurm", "lsf", "openlava", and "torque". If the template is not given then a default is selected from the `batchtools` package.

`registryargs` `list()`

Arguments given to the registry created by `BatchtoolsParam` to configure the registry and where it's being stored. The `registryargs` can be specified by the function `batchtoolsRegistryargs()` which takes the arguments `file.dir`, `work.dir`, `packages`, `namespaces`, `source`, `load`, `make.default`. It's useful to configure these option, especially the `file.dir` to a location which is accessible to all the nodes on your job scheduler i.e master and workers. `file.dir` uses a default setting to make a registry in your working directory.

`saveregistry` `logical(1)`

Option given to store the entire registry for the job(s). This functionality should only be used when debugging. The storage of the entire registry can be time and space expensive on disk. The registry will be saved in the directory specified by `file.dir` in `registryargs`; the default locatoin is the current working directory. The saved registry directories will have suffix "-1", "-2" and so on, for each time the `BatchtoolsParam` is used.

`resources` named `list()`

Arguments passed to the `resources` argument of `batchtools::submitJobs` during evaluation of `bplapply` and similar functions. These name-value pairs are used for substitution into the template file.

`template` `character(1)`

Path to a template for the backend in `BatchtoolsParam`. It is possible to check which template is being used by the object using the getter `bpbackend(BatchtoolsParam())`. The template needs to be written specific to each backend. Please check the list of available templates in the `batchtools` package.

`stop.on.error` `logical(1)`

Stop all jobs as soon as one jobs fails (`stop.on.error == TRUE`) or wait for all jobs to terminate. Default is `TRUE`.

`progressbar` `logical(1)`

Suppress the progress bar used in BatchtoolsParam and be less verbose. Default is `FALSE`.

`RNGseed` `integer(1)`

Set an initial seed for the RNG. Default is `NULL` where a random seed is chosen upon initialization.

`timeout` `list()`

Time (in seconds) allowed for worker to complete a task. If the computation exceeds `timeout` an error is thrown with message 'reached elapsed time limit'.

`exportglobals` `logical(1)`

Export `base::options()` from manager to workers? Default `TRUE`.

`log` `logical(1)`

Option given to save the logs which are produced by the jobs. If `log=TRUE` then the `logdir` option must be specified.

`logdir` `character(1)`

Path to location where logs are stored. The argument `log=TRUE` is required before using the logdir option.

`resultdir` `logical(1)`

Path where results are stored.

`jobname` `character(1)`

Job name that is prepended to the output log and result files. Default is "BPJOB".

`...` name-value pairs

Names and values correspond to arguments from batchtools `makeRegistry`.

BatchtoolsParam constructor

Return an object with specified values. The object may be saved to disk or reused within a session.

Methods

The following generics are implemented and perform as documented on the corresponding help page: bpworkers, bpnworkers, bpstart, bpstop, bpisup, bpbackend.

bplapply handles arguments X of classes derived from S4Vectors::List specially, coercing to list.

Author(s)

Nitesh Turaga, mailto:nitesh.turaga@roswellpark.org

See Also

getClass("BiocParallelParam") for additional parameter classes.

register for registering parameter classes for use in parallel evaluation.

The batchtools package.

Examples

```
## Pi approximation
piApprox = function(n) {
    nums = matrix(runif(2 * n), ncol = 2)
    d = sqrt(nums[, 1]^2 + nums[, 2]^2)
    4 * mean(d <= 1)
}

piApprox(1000)

## Calculate piApprox 10 times
param <- BatchtoolsParam()
result <- bplapply(rep(10e5, 10), piApprox, BPPARAM=param)

## Not run:
## see vignette for additional explanation
library(BiocParallel)
param = BatchtoolsParam(workers=5,
                        cluster="sge",
                        template="script/test-sge-template.tmpl")
## Run parallel job
result = bplapply(rep(10e5, 100), piApprox, BPPARAM=param)

## bpmapply
param = BatchtoolsParam()
result = bpmapply(fun, x = 1:3, y = 1:3, MoreArgs = list(z = 1),
                  SIMPLIFY = TRUE, BPPARAM = param)

## bpvec
```

```
param = BatchtoolsParam(workers=2)
result = bpvec(1:10, seq_along, BPPARAM=param)

## bpvectorize
param = BatchtoolsParam(workers=2)
## this returns a function
bpseq_along = bpvectorize(seq_along, BPPARAM=param)
result = bpseq_along(1:10)

## bpiterate
ITER <- function(n=5) {
        i <- 0L
        function() {
            i <<- i + 1L
            if (i > n)
                return(NULL)
        rep(i, n)
        }
    }

param <- BatchtoolsParam()
res <- bpiterate(ITER=ITER(), FUN=function(x,y) sum(x) + y, y=10, BPPARAM=param)

## save logs
logdir <- tempfile()
dir.create(logdir)
param <- BatchtoolsParam(log=TRUE, logdir=logdir)
res <- bplapply(rep(10e5, 10), piApprox, BPPARAM=param)

## save registry (should be used only for debugging)
file.dir <- tempfile()
registryargs <- batchtoolsRegistryargs(file.dir = file.dir)
param <- BatchtoolsParam(saveregistry = TRUE, registryargs = registryargs)
res <- bplapply(rep(10e5, 10), piApprox, BPPARAM=param)
dir(dirname(file.dir), basename(file.dir))

## End(Not run)
```

`BiocParallel-defunct` *Defunct Objects in Package 'BiocParallel'*

Description

These functions and objects are defunct and no longer available.

Details

Defunct functions are: `bprunMPIslave()`.

Defunct classes: `BatchJobsParam`.

`BiocParallel-deprecated`

Deprecated Functions in Package 'BiocParallel'

Description

There are currently no deprecated functions in 'BiocParallel'.

`BiocParallelParam-class`

BiocParallelParam objects

Description

The `BiocParallelParam` virtual class stores configuration parameters for parallel execution. Concrete subclasses include `SnowParam`, `MulticoreParam`, `BatchtoolsParam`, and `DoparParam` and `SerialParam`.

Details

`BiocParallelParam` is the virtual base class on which other parameter objects build. There are 5 concrete subclasses:

`SnowParam`**:** distributed memory computing

`MulticoreParam`**:** shared memory computing

`BatchtoolsParam`**:** scheduled cluster computing

`DoparParam`**:** foreach computing

`SerialParam`**:** non-parallel execution

The parameter objects hold configuration parameters related to the method of parallel execution such as shared memory, independent memory or computing with a cluster scheduler.

Construction

The `BiocParallelParam` class is virtual and has no constructor. Instances of the subclasses can be created with the following:

- `SnowParam()`
- `MulticoreParam()`
- `BatchtoolsParam()`
- `DoparParam()`
- `SerialParam()`

Accessors

Back-end control: In the code below BPPARAM is a BiocParallelParam object.

bpworkers(x), bpworkers(x, ...): integer(1) or character(). Gets the number or names of the back-end workers. The setter is supported for SnowParam and MulticoreParam only.

bpnworkers(x): integer(1). Gets the number of the back-end workers.

bptasks(x), bptasks(x) <- value: integer(1). Get or set the number of tasks for a job. value can be a scalar integer > 0L, or integer 0L for matching the worker number, or NA for representing an infinite task number. DoparParam and BatchtoolsParam have their own approach to dividing a job among workers.

We define a job as a single call to a function such as bplapply, bpmapply etc. A task is the division of the X argument into chunks. When tasks == 0 (default), X is divided by the number of workers. This approach distributes X in (approximately) equal chunks.

A tasks value of > 0 dictates the total number of tasks. Values can range from 1 (all of X to a single worker) to the length of X (each element of X to a different worker); values greater than length(X) (e.g., .Machine$integer.max) are rounded to length(X).

When the length of X is less than the number of workers each element of X is sent to a worker and tasks is ignored. Another case where the tasks value is ignored is when using the bpiterate function; the number of tasks are defined by the number of data chunks returned by the ITER function.

bpstart(x): logical(1). Starts the back-end, if necessary.

bpstop(x): logical(1). Stops the back-end, if necessary and possible.

bpisup(x): logical(1). Tests whether the back-end is available for processing, returning a scalar logical value. bp* functions such as bplapply automatically start the back-end if necessary.

bpbackend(x), bpbackend(x) <- value: Gets or sets the parallel bpbackend. Not all back-ends can be retrieved; see methods("bpbackend").

bplog(x), bplog(x) <- value: Get or enable logging, if available. value must be a logical(1).

bpthreshold(x), bpthreshold(x) <- value: Get or set the logging threshold. value must be a character(1) string of one of the levels defined in the futile.logger package: "TRACE", "DEBUG", "INFO", "WARN", "ERROR", or "FATAL".

bplogdir(x), bplogdir(x) <- value: Get or set an optional directory for saving log files. The directory must already exist with read / write ability.

bpresultdir(x), bpresultdir(x) <- value: Get or set an optional directory for saving results as 'rda' files. The directory must already exist with read / write ability.

bptimeout(x), bptimeout(x) <- value: numeric(1) Time (in seconds) allowed for worker to complete a task. This value is passed to base::setTimeLimit() as both the cpu and elapsed arguments. If the computation exceeds timeout an error is thrown with message 'reached elapsed time limit'.

bpexportglobals(x), bpexportglobals(x) <- value: logical(1) Export base::options() from manager to workers? Default TRUE.

bpexportvariables(x), bpexportvariables(x) <- value: logical(1) Automatically export the variables which are defined in the global environment and used by the function from manager to workers. Default TRUE.

bpprogressbar(x), bpprogressbar(x) <- value: Get or set the value to enable text progress bar. value must be a logical(1).

`bpRNGseed(x)`, `bpRNGseed(x) <- value`: Get or set the seed for random number generaton. `value` must be a `numeric(1)` or `NULL`.

`bpjobname(x)`, `bpjobname(x) <- value`: Get or set the job name.

`bpforceGC(x)`, `bpforceGC(x) <- value`: Get or set whether 'garbage collection' should be invoked at the end of each call to `FUN`.

`bpfallback(x)`, `bpfallback(x) <- value`: Get or set whether the fallback `SerialParam` should be used (e.g., for efficiency when starting a cluster) when the current `BPPARAM` has not been started and the worker number is less than or equal to 1.

Error Handling: In the code below `BPPARAM` is a `BiocParallelParam` object.

`bpstopOnError(x)`, `bpstopOnError(x) <- value`: `logical()`. Controls if the job stops when an error is hit.
`stop.on.error` controls whether the job stops after an error is thrown. When `TRUE`, the output contains all successfully completed results up to and including the error. When `stop.on.error == TRUE` all computations stop once the error is hit. When `FALSE`, the job runs to completion and successful results are returned along with any error messages.

Methods

Evaluation: In the code below `BPPARAM` is a `BiocParallelParam` object. Full documentation for these functions are on separate man pages: see ?`bpmapply`, ?`bplapply`, ?`bpvec`, ?`bpiterate` and ?`bpaggregate`.

- `bpmapply(FUN, ..., MoreArgs=NULL, SIMPLIFY=TRUE, USE.NAMES=TRUE, BPPARAM=bpparam())`
- `bplapply(X, FUN, ..., BPPARAM=bpparam())`
- `bpvec(X, FUN, ..., AGGREGATE=c, BPPARAM=bpparam())`
- `bpiterate(ITER, FUN, ..., BPPARAM=bpparam())`
- `bpaggregate(x, data, FUN, ..., BPPARAM=bpparam())`

Other: In the code below `BPPARAM` is a `BiocParallelParam` object.

- `show(x)`

Author(s)

Martin Morgan and Valerie Obenchain.

See Also

- `SnowParam` for computing in distributed memory
- `MulticoreParam` for computing in shared memory
- `BatchtoolsParam` for computing with cluster schedulers
- `DoparParam` for computing with foreach
- `SerialParam` for non-parallel execution

Examples

```
getClass("BiocParallelParam")

## For examples see ?SnowParam, ?MulticoreParam, ?BatchtoolsParam
## and ?SerialParam.
```

bpaggregate *Apply a function on subsets of data frames*

Description

This is a parallel version of aggregate.

Usage

```
## S4 method for signature 'formula,BiocParallelParam'
bpaggregate(x, data, FUN, ...,
    BPREDO=list(), BPPARAM=bpparam(), BPOPTIONS = bpoptions())

## S4 method for signature 'data.frame,BiocParallelParam'
bpaggregate(x, by, FUN, ...,
    simplify=TRUE, BPREDO=list(), BPPARAM=bpparam(), BPOPTIONS = bpoptions())

## S4 method for signature 'matrix,BiocParallelParam'
bpaggregate(x, by, FUN, ...,
    simplify=TRUE, BPREDO=list(),
    BPPARAM=bpparam(), BPOPTIONS = bpoptions()
)

## S4 method for signature 'ANY,missing'
bpaggregate(x, ..., BPREDO=list(),
    BPPARAM=bpparam(), BPOPTIONS = bpoptions()
)
```

Arguments

x	A data.frame, matrix or a formula.
by	A list of factors by which x is split; applicable when x is data.frame or matrix.
data	A data.frame; applicable when x is a formula.
FUN	Function to apply.
...	Additional arguments for FUN.
simplify	If set to TRUE, the return values of FUN will be simplified using simplify2array.
BPPARAM	An optional BiocParallelParam instance determining the parallel back-end to be used during evaluation.
BPREDO	A list of output from bpaggregate with one or more failed elements. When a list is given in BPREDO, bpok is used to identify errors, tasks are rerun and inserted into the original results.
BPOPTIONS	Additional options to control the behavior of the parallel evaluation, see bpoptions.

Details

bpaggregate is a generic with methods for data.frame matrix and formula objects. x is divided into subsets according to factors in by. Data chunks are sent to the workers, FUN is applied and results are returned as a data.frame.

The function is similar in spirit to aggregate from the stats package but aggregate is not explicitly called. The bpaggregate formula method reformulates the call and dispatches to the data.frame method which in turn distributes data chunks to workers with bplapply.

Value

See aggregate.

Author(s)

Martin Morgan mailto:mtmorgan@fhcrc.org.

Examples

```
if (interactive() && require(Rsamtools) && require(GenomicAlignments)) {

  fl <- system.file("extdata", "ex1.bam", package="Rsamtools")
  param <- ScanBamParam(what = c("flag", "mapq"))
  gal <- readGAlignments(fl, param=param)

  ## Report the mean map quality by range cutoff:
  cutoff <- rep(0, length(gal))
  cutoff[start(gal) > 1000 & start(gal) < 1500] <- 1
  cutoff[start(gal) > 1500] <- 2
  bpaggregate(as.data.frame(mcols(gal)$mapq), list(cutoff = cutoff), mean)

}
```

---

bpiterate *Parallel iteration over an indeterminate number of data chunks*

Description

bpiterate iterates over an indeterminate number of data chunks (e.g., records in a file). Each chunk is processed by parallel workers in an asynchronous fashion; as each worker finishes it receives a new chunk. Data are traversed a single time.

When provided with a vector-like argument ITER = X, bpiterate uses bpiterateAlong to produce the sequence of elements X[[1]], X[[2]], etc.

Usage

```
bpiterate(
    ITER, FUN, ...,
    BPREDO = list(), BPPARAM=bpparam(), BPOPTIONS = bpoptions()
)

## S4 method for signature 'ANY,ANY,missing'
bpiterate(
    ITER, FUN, ...,
    BPREDO = list(), BPPARAM=bpparam(), BPOPTIONS = bpoptions())

## S4 method for signature 'ANY,ANY,BatchtoolsParam'
bpiterate(
    ITER, FUN, ..., REDUCE, init, reduce.in.order=FALSE,
    BPREDO = list(), BPPARAM=bpparam(), BPOPTIONS = bpoptions()
)

bpiterateAlong(X)
```

Arguments

X	An object (e.g., vector or list) with 'length()' and '[[' methods available.
ITER	A function with no arguments that returns an object to process, generally a chunk of data from a file. When no objects are left (i.e., end of file) it should return NULL and continue to return NULL regardless of the number of times it is invoked after reaching the end of file. This function is run on the master.
FUN	A function to process the object returned by ITER; run on parallel workers separate from the master. When BPPARAM is a MulticoreParam, FUN is 'decorated' with additional arguments and therefore must have ... in the signature.
BPPARAM	An optional BiocParallelParam instance determining the parallel back-end to be used during evaluation, or a list of BiocParallelParam instances, to be applied in sequence for nested calls to bpiterate.
REDUCE	Optional function that combines (reduces) output from FUN. As each worker returns, the data are combined with the REDUCE function. REDUCE takes 2 arguments; one is the current result and the other is the output of FUN from a worker that just finished.
init	Optional initial value for REDUCE; must be of the same type as the object returned from FUN. When supplied, reduce.in.order is set to TRUE.
reduce.in.order	Logical. When TRUE, REDUCE is applied to the results from the workers in the same order the tasks were sent out.
BPREDO	An output from bpiterate with one or more failed elements. This argument cannot be used with BatchtoolsParam
...	Arguments to other methods, and named arguments for FUN.
BPOPTIONS	Additional options to control the behavior of the parallel evaluation, see bpoptions.

Details

Supported for SnowParam, MulticoreParam and BatchtoolsParam.

bpiterate iterates through an unknown number of data chunks, dispatching chunks to parallel workers as they become available. In contrast, other bp*apply functions such as bplapply or bpmapply require the number of data chunks to be specified ahead of time. This quality makes bpiterate useful for iterating through files of unknown length.

ITER serves up chunks of data until the end of the file is reached at which point it returns NULL. Note that ITER should continue to return NULL reguardless of the number of times it is invoked after reaching the end of the file. FUN is applied to each object (data chunk) returned by ITER.

bpiterateAlong() provides an interator for a vector or other object with length() and [[methods defined. It is used in place of the first argument ITER=

Value

By default, a list the same length as the number of chunks in ITER(). When REDUCE is used, the return is consistent with application of the reduction. When errors occur, the errors will be attached to the result as an attribute errors

Author(s)

Valerie Obenchain mailto:vobencha@fhcrc.org.

See Also

- bpvec for parallel, vectorized calculations.
- bplapply for parallel, lapply-like calculations.
- BiocParallelParam for details of BPPARAM.
- BatchtoolsParam for details of BatchtoolsParam.

Examples

```
## A simple iterator
ITER <- bpiterateAlong(1:10)
result <- bpiterate(ITER, sqrt)
## alteernatively, result <- bpiterate(1:10, sqrt)
unlist(result)

## Not run:
if (require(Rsamtools) && require(RNAseqData.HNRNPC.bam.chr14) &&
    require(GenomicAlignments) && require(ShortRead)) {

  ## -------------------------------------------------------------------
  ## Iterate through a BAM file
  ## -------------------------------------------------------------------

  ## Select a single file and set 'yieldSize' in the BamFile object.
  fl <- RNAseqData.HNRNPC.bam.chr14_BAMFILES[[1]]
  bf <- BamFile(fl, yieldSize = 300000)
```

```
## bamIterator() is initialized with a BAM file and returns a function.
## The return function requires no arguments and iterates through the
## file returning data chunks the size of yieldSize.
bamIterator <- function(bf) {
    done <- FALSE
    if (!isOpen( bf))
 open(bf)

    function() {
 if (done)
     return(NULL)
 yld <- readGAlignments(bf)
 if (length(yld) == 0L) {
     close(bf)
     done <<- TRUE
     NULL
 } else yld
    }
}

## FUN counts reads in a region of interest.
roi <- GRanges("chr14", IRanges(seq(19e6, 107e6, by = 10e6), width = 10e6))
counter <- function(reads, roi, ...) {
    countOverlaps(query = roi, subject = reads)
}

## Initialize the iterator.
ITER <- bamIterator(bf)

## The number of chunks returned by ITER() determines the result length.
bpparam <- MulticoreParam(workers = 3)
## bpparam <- BatchtoolsParam(workers = 3), see ?BatchtoolsParam
bpiterate(ITER, counter, roi = roi, BPPARAM = bpparam)

## Re-initialize the iterator and combine on the fly with REDUCE:
ITER <- bamIterator(bf)
bpparam <- MulticoreParam(workers = 3)
bpiterate(ITER, counter, REDUCE = sum, roi = roi, BPPARAM = bpparam)

## ----------------------------------------------------------------------
## Iterate through a FASTA file
## ----------------------------------------------------------------------

## Set data chunk size with 'n' in the FastqStreamer object.
sp <- SolexaPath(system.file('extdata', package = 'ShortRead'))
fl <- file.path(analysisPath(sp), "s_1_sequence.txt")

## Create an iterator that returns data chunks the size of 'n'.
fastqIterator <- function(fqs) {
    done <- FALSE
    if (!isOpen(fqs))
 open(fqs)
```

```
    function() {
 if (done)
     return(NULL)
 yld <- yield(fqs)
 if (length(yld) == 0L) {
     close(fqs)
     done <<- TRUE
     NULL
 } else yld
    }
}

## The process function summarizes the number of times each sequence occurs.
summary <- function(reads, ...) {
     ShortRead::tables(reads, n = 0)$distribution
}

## Create a param.
bpparam <- SnowParam(workers = 2)

## Initialize the streamer and iterator.
fqs <- FastqStreamer(fl, n = 100)
ITER <- fastqIterator(fqs)
bpiterate(ITER, summary, BPPARAM = bpparam)

## Results from the workers are combined on the fly when REDUCE is used.
## Collapsing the data in this way can substantially reduce memory
## requirements.
fqs <- FastqStreamer(fl, n = 100)
ITER <- fastqIterator(fqs)
bpiterate(ITER, summary, REDUCE = merge, all = TRUE, BPPARAM = bpparam)

}

## End(Not run)
```

`bplapply` *Parallel lapply-like functionality*

Description

`bplapply` applies `FUN` to each element of `X`. Any type of object `X` is allowed, provided `length`, `[`, and `[[` methods are available. The return value is a `list` of length equal to `X`, as with `lapply`.

Usage

```
bplapply(X, FUN, ..., BPREDO = list(), BPPARAM=bpparam(), BPOPTIONS = bpoptions())
```

Arguments

X	Any object for which methods length, [, and [[are implemented.
FUN	The function to be applied to each element of X.
...	Additional arguments for FUN, as in lapply.
BPPARAM	An optional BiocParallelParam instance determining the parallel back-end to be used during evaluation, or a list of BiocParallelParam instances, to be applied in sequence for nested calls to **BiocParallel** functions.
BPREDO	A list of output from bplapply with one or more failed elements. When a list is given in BPREDO, bpok is used to identify errors, tasks are rerun and inserted into the original results.
BPOPTIONS	Additional options to control the behavior of the parallel evaluation, see bpoptions.

Details

See methods{bplapply} for additional methods, e.g., method?bplapply("MulticoreParam").

Value

See lapply.

Author(s)

Martin Morgan mailto:mtmorgan@fhcrc.org. Original code as attributed in mclapply.

See Also

- bpvec for parallel, vectorized calculations.
- BiocParallelParam for possible values of BPPARAM.

Examples

```
methods("bplapply")

## ten tasks (1:10) so ten calls to FUN default registered parallel
## back-end. Compare with bpvec.
fun <- function(v) {
    message("working") ## 10 tasks
    sqrt(v)
}
bplapply(1:10, fun)
```

bploop *Internal Functions for SNOW-style Parallel Evaluation*

Description

The functions documented on this page are primarily for use within **BiocParallel** to enable SNOW-style parallel evaluation, using communication between manager and worker nodes through sockets.

Usage

```
## S3 method for class 'lapply'
bploop(manager, X, FUN, ARGS, BPPARAM, BPOPTIONS = bpoptions(), BPREDO, ...)

## S3 method for class 'iterate'
bploop(manager, ITER, FUN, ARGS, BPPARAM, BPOPTIONS = bpoptions(),
       REDUCE, BPREDO, init, reduce.in.order, ...)
```

Arguments

manager	An object representing the manager node. For workers, this is the node to which the worker will communicate. For managers, this is the form of iteration – `lapply` or `iterate`.
X	A vector of jobs to be performed.
FUN	A function to apply to each job.
ARGS	A list of arguments to be passed to `FUN`.
BPPARAM	An instance of a `BiocParallelParam` class.
ITER	A function used to generate jobs. No more jobs are available when `ITER()` returns `NULL`.
REDUCE	(Optional) A function combining two values returned by `FUN` into a single value.
init	(Optional) Initial value for reduction.
reduce.in.order	(Optional) logical(1) indicating that reduction must occur in the order jobs are dispatched (`TRUE`) or that reduction can occur in the order jobs are completed (`FALSE`).
BPREDO	(Optional) A `list` of output from `bplapply` or `bpiterate` with one or more failed elements.
...	Additional arguments, ignored in all cases.
BPOPTIONS	Additional options to control the behavior of the parallel evaluation, see `bpoptions`.

Details

Workers enter a loop. They wait to receive a message (R list) from the `manager`. The message contains a `type` element, with evaluation as follows:

"EXEC" Execute the R code in the message, returning the result to the `manager`.

"DONE" Signal termination to the `manager`, terminate the worker.

Managers under `lapply` dispatch pre-determined jobs, `X`, to workers, collecting the results from and dispatching new jobs to the first available worker. The manager returns a list of results, in a one-to-one correspondence with the order of jobs supplied, when all jobs have been evaluated.

Managers under `iterate` dispatch an undetermined number of jobs to workers, collecting previous jobs from and dispatching new jobs to the first available worker. Dispatch continues until available jobs are exhausted. The return value is by default a list of results in a one-to-one correspondence with the order of jobs supplied. The return value is influenced by `REDUCE`, `init`, and `reduce.in.order`.

Author(s)

Valerie Obenchain, Martin Morgan. Derived from similar functionality in the **snow** and **parallel** packages.

Examples

```
## These functions are not meant to be called by the end user.
```

`bpmapply` *Parallel mapply-like functionality*

Description

`bpmapply` applies `FUN` to first elements of `...`, the second elements and so on. Any type of object in `...` is allowed, provided `length`, `[`, and `[[` methods are available. The return value is a `list` of length equal to the length of all objects provided, as with `mapply`.

Usage

```
bpmapply(FUN, ..., MoreArgs=NULL, SIMPLIFY=TRUE, USE.NAMES=TRUE,
    BPREDO=list(), BPPARAM=bpparam(), BPOPTIONS = bpoptions())

## S4 method for signature 'ANY,missing'
bpmapply(FUN, ..., MoreArgs=NULL, SIMPLIFY=TRUE,
    USE.NAMES=TRUE, BPREDO=list(), BPPARAM=bpparam(), BPOPTIONS = bpoptions())

## S4 method for signature 'ANY,BiocParallelParam'
bpmapply(FUN, ..., MoreArgs=NULL,
    SIMPLIFY=TRUE, USE.NAMES=TRUE, BPREDO=list(),
    BPPARAM=bpparam(), BPOPTIONS = bpoptions())
```

Arguments

FUN	The function to be applied to each element passed via ....
...	Objects for which methods length, [, and [[are implemented. All objects must have the same length or shorter objects will be replicated to have length equal to the longest.
MoreArgs	List of additional arguments to FUN.
SIMPLIFY	If TRUE the result will be simplified using simplify2array.
USE.NAMES	If TRUE the result will be named.
BPPARAM	An optional BiocParallelParam instance defining the parallel back-end to be used during evaluation.
BPREDO	A list of output from bpmapply with one or more failed elements. When a list is given in BPREDO, bpok is used to identify errors, tasks are rerun and inserted into the original results.
BPOPTIONS	Additional options to control the behavior of the parallel evaluation, see bpoptions.

Details

See methods{bpmapply} for additional methods, e.g., method?bpmapply("MulticoreParam").

Value

See mapply.

Author(s)

Michel Lang . Original code as attributed in mclapply.

See Also

- bpvec for parallel, vectorized calculations.
- BiocParallelParam for possible values of BPPARAM.

Examples

```
methods("bpmapply")

fun <- function(greet, who) {
    paste(Sys.getpid(), greet, who)
}
greet <- c("morning", "night")
who <- c("sun", "moon")

param <- bpparam()
original <- bpworkers(param)
bpworkers(param) <- 2
result <- bpmapply(fun, greet, who, BPPARAM = param)
cat(paste(result, collapse="\n"), "\n")
bpworkers(param) <- original
```

bpok *Resume computation with partial results*

Description

Identifies unsuccessful results returned from bplapply, bpmapply, bpvec, bpaggregate or bpvectorize.

Usage

```
bpok(x, type = bperrorTypes())

bperrorTypes()

bpresult(x)
```

Arguments

x	Results returned from a call to bp*apply.
type	A character(1) error type, from the vector returned by bperrorTypes() and described below

Details

bpok() returns a logical() vector: FALSE for any jobs that resulted in an error. x is the result list output by bplapply, bpmapply, bpvec, bpaggregate or bpvectorize.

bperrorTypes() returns a character() vector of possible error types generated during parallel evaluation. Types are:

- bperror: Any of the following errors. This is the default value for bpok().
- remote_error: An *R* error occurring while evaluating FUN(), e.g., taking the square root of a character vector, sqrt("One").
- unevaluated_error: When *Param(stop.on.error = TRUE) (default), a remote error halts evaluation of other tasks assigned to the same worker. The return value for these unevaluated elements is an error of type unevaluated_error.
- not_available_error: Only produced by DoparParam() when a remote error occurs during evaluation of an element of X – DoparParam() sets all values after the remote error to this class.
- worker_comm_error: An error occurring while trying to communicate with workers, e.g., when a worker quits unexpectedly. when this type of error occurs, the length of the result may differ from the length of the input X.

bpresult() when applied to an object with a class of one of the error types returns the list of tasks results.

Author(s)

Michel Lang, Martin Morgan, Valerie Obenchain, and Jiefei Wang

See Also

tryCatch

Examples

```
## ------------------------------------------------------------------------
## Catch errors:
## ------------------------------------------------------------------------

## By default 'stop.on.error' is TRUE in BiocParallelParam objects.  If
## 'stop.on.error' is TRUE an ill-fated bplapply() simply stops,
## displaying the error message.
param <- SnowParam(workers = 2, stop.on.error = TRUE)
result <- tryCatch({
    bplapply(list(1, "two", 3), sqrt, BPPARAM = param)
}, error=identity)
result
class(result)
bpresult(result)

## If 'stop.on.error' is FALSE then the computation continues. Errors
## are signalled but the full evaluation can be retrieved
param <- SnowParam(workers = 2, stop.on.error = FALSE)
X <- list(1, "two", 3)
result <- bptry(bplapply(X, sqrt, BPPARAM = param))
result

## Check for errors:
fail <- !bpok(result)
fail

## Access the traceback with attr():
tail(attr(result[[2]], "traceback"), 5)

## ------------------------------------------------------------------------
## Resume calculations:
## ------------------------------------------------------------------------

## The 'resume' mechanism is triggered by supplying a list of partial
## results as 'BPREDO'. Data elements that failed are rerun and merged
## with previous results.

## A call of sqrt() on the character "2" returns an error. Fix the input
## data by changing the character "2" to a numeric 2:
X_mod <- list(1, 2, 3)
bplapply(X_mod, sqrt, BPPARAM = param , BPREDO = result)
```

bpoptions *Additional options to parallel evaluation*

Description

This function is used to pass additional options to bplapply() and other functions function. One use case is to use the argument BPOPTIONS to temporarily change the parameter of BPPARAM (e.g. enabling the progressbar). A second use case is to change the behavior of the parallel evaluation (e.g. manually exporting some variables to the worker)

Usage

```
bpoptions(
    workers, tasks, jobname, log, logdir, threshold, resultdir,
    stop.on.error, timeout, exportglobals, exportvariables, progressbar,
    RNGseed, force.GC, fallback, exports, packages, ...
)
```

Arguments

workers	integer(1) or character() parameter for BPPARAM; see bpnworkers.
tasks	integer(1) parameter for BPPARAM; see bptasks.
jobname	character(1) parameter for BPPARAM; see bpjobname.
log	logical(1) parameter for BPPARAM; see bplog.
logdir	character(1) parameter for BPPARAM; see bplogdir.
threshold	A parameter for BPPARAM; see bpthreshold.
resultdir	character(1) parameter for BPPARAM; see bpresultdir.
stop.on.error	logical(1) parameter for BPPARAM; see bpstopOnError.
timeout	integer(1) parameter for BPPARAM; see bptimeout.
exportglobals	logical(1) parameter for BPPARAM; see bpexportglobals.
exportvariables	A parameter for BPPARAM; see bpexportvariables.
progressbar	logical(1) parameter for BPPARAM; see bpprogressbar.
RNGseed	integer(1) parameter for BPPARAM; see bpRNGseed.
force.GC	logical(1) parameter for BPPARAM; see bpforceGC.
fallback	logical(1) parameter for BPPARAM; see bpfallback.
exports	character() The names of the variables in the global environment which need to be exported to the global environment of the worker. This option works independently of the option exportvariables.
packages	character() The packages that needs to be attached by the worker prior to the evaluation of the task. This option works independently of the option exportvariables.
...	Additional arguments which may(or may not) work for some specific type of BPPARAM.

Value

A list of options

Author(s)

Jiefei Wang

See Also

BiocParallelParam, bplapply, bpiterate.

Examples

```
p <- SerialParam()
bplapply(1:5, function(x) Sys.sleep(1), BPPARAM = p,
       BPOPTIONS = bpoptions(progressbar = TRUE, tasks = 5L))
```

---

bpschedule *Schedule back-end Params*

Description

Use functions on this page to influence scheduling of parallel processing.

Usage

```
bpschedule(x)
```

Arguments

x	An instance of a BiocParallelParam class, e.g., MulticoreParam, SnowParam, DoparParam. x can be missing, in which case the default back-end (see register) is used.

Details

bpschedule returns a logical(1) indicating whether the parallel evaluation should occur at this point.

Value

bpschedule returns a scalar logical.

Author(s)

Martin Morgan mailto:mtmorgan@fhcrc.org.

See Also

BiocParallelParam for possible values of x.

Examples

```
bpschedule(SnowParam())                 # TRUE
bpschedule(MulticoreParam(2))           # FALSE on windows

p <- MulticoreParam()
bpschedule(p)                           # TRUE
bplapply(1:2, function(i, p) {
    bpschedule(p)                       # FALSE
}, p = p, BPPARAM=p)
```

`bptry` *Try expression evaluation, recovering from bperror signals*

Description

This function is meant to be used as a wrapper around `bplapply()` and friends, returning the evaluated expression rather than signalling an error.

Usage

```
bptry(expr, ..., bplist_error, bperror)
```

Arguments

`expr`	An R expression; see `tryCatch`.
`bplist_error`	A 'handler' function of a single argument, used to catch `bplist_error` conditions signalled by `expr`. A `bplist_error` condition is signalled when an element of `bplapply` and other iterations contain a evaluation that failed. When missing, the default retrieves the "result" attribute from the error, containing the partially evaluated results. Setting `bplist_error=identity` returns the evaluated condition. Setting `bplist_error=stop` passes the condition to other handlers, notably the handler provided by `bperror`.
`bperror`	A 'handler' function of a single argument, use to catch `bperror` conditions signalled by `expr`. A `bperror` is a base class to all errors signaled by **BiocParallel** code. When missing, the default returns the condition without signalling an error.
`...`	Additional named handlers passed to `tryCatch()`. These user-provided handlers are evaluated before default handlers `bplist_error`, `bperror`.

Value

The partially evaluated list of results.

Author(s)

Martin Morgan <martin.morgan@roswellpark.org>

See Also

bpok, tryCatch, bplapply.

Examples

```
param = registered()[[1]]
param
X = list(1, "2", 3)
bptry(bplapply(X, sqrt))                          # bplist_error handler
result <- bptry(bplapply(X, sqrt), bplist_error=identity)  # bperror handler
result
bpresult(result)
```

---

bpvalidate *Tools for developing functions for parallel execution in distributed memory*

Description

bpvalidate interrogates the function environment and search path to locate undefined symbols.

Usage

```
bpvalidate(fun, signal = c("warning", "error", "silent"))
```

Arguments

fun	The function to be checked. typeof(fun) must return either "closure" or "builtin".
signal	character(1) matching "warning", "error", "silent" or a function with signature (..., call.) to be invoked when reporting errors. Using "silent" suppresses output; "warning" and "error" emit warnings or errors when fun contains references to unknown variables or variables defined in the global environment (and hence not serialized to workers).

Details

bpvalidate tests if a function can be run in a distributed memory environment (e.g., SOCK clusters, Windows machines). bpvalidate looks in the environment of fun, in the NAMESPACE exports of libraries loaded in fun, and along the search path to identify any symbols outside the scope of fun.

bpvalidate can be used to check functions passed to the bp* family of functions in BiocParallel or other packages that support parallel evaluation on clusters such as snow, Rmpi, etc.

testing package functions The environment of a function defined inside a package is the NAMESPACE of the package. It is important to test these functions as they will be called from within the package, with the appropriate environment. Specifically, do not copy/paste the function into the workspace; once this is done the GlobalEnv becomes the function environment.

To test a package function, load the package then call the function by name (myfun) or explicitly (mypkg:::myfun) if not exported.

testing workspace functions The environment of a function defined in the workspace is the GlobalEnv. Because these functions do not have an associated package NAMESPACE, the functions and variables used in the body must be explicitly passed or defined. See examples.

Defining functions in the workspace is often done during development or testing. If the function is later moved inside a package, it can be rewritten in a more lightweight form by taking advantage of imported symbols in the package NAMESPACE.

NOTE: `bpvalidate` does not currently work on Generics.

Value

An object of class `BPValidate` summarizing symbols identified in the global environment or search path, or undefined in the enviornments the function was defined in. Details are only available via 'show()'.

Author(s)

Martin Morgan mailto:mtmorgan.bioc@gmail.com and Valerie Obenchain.

Examples

```
## ------------------------------------------------------------------------
## Interactive use
## ------------------------------------------------------------------------

fun <- function()
    .__UNKNOWN_SYMBOL__
bpvalidate(fun, "silent")

## ------------------------------------------------------------------------
## Testing package functions
## ------------------------------------------------------------------------

## Not run:
library(myPkg)

## Test exported functions by name or the double colon:
bpvalidate(myExportedFun)
bpvalidate(myPkg::myExportedFun)

## Non-exported functions are called with the triple colon:
bpvalidate(myPkg:::myInternalFun)

## End(Not run)

## ------------------------------------------------------------------------
## Testing workspace functions
## ------------------------------------------------------------------------

## Functions defined in the workspace have the .GlobalEnv as their
## environment. Often the symbols used inside the function body
## are not defined in .GlobalEnv and must be passed explicitly.
```

```
## Loading libraries:
## In 'fun1' countBam() is flagged as unknown:
fun1 <- function(fl, ...)
    countBam(fl)
v <- bpvalidate(fun1)

## countBam() is not defined in .GlobalEnv and must be passed as
## an argument or made available by loading the library.
fun2 <- function(fl, ...) {
    Rsamtools::countBam(fl)
}
v <- bpvalidate(fun2)

## Passing arguments:
## 'param' is defined in the workspace but not passed to 'fun3'.
## bpvalidate() flags 'param' as being found '.GlobalEnv' which means
## it is not defined in the function environment or inside the function.
library(Rsamtools)
param <- ScanBamParam(flag=scanBamFlag(isMinusStrand=FALSE))

fun3 <- function(fl, ...) {
  Rsamtools::countBam(fl, param=param)
}
v <- bpvalidate(fun3)

## 'param' is explicitly passed by adding it as a formal argument.
fun4 <- function(fl, ..., param) {
  Rsamtools::countBam(fl, param=param)
}
bpvalidate(fun4)

## The corresponding call to a bp* function includes 'param':
## Not run:
bplapply(files, fun4, param=param, BPPARAM=SnowParam(2))

## End(Not run)
```

---

bpvec *Parallel, vectorized evaluation*

Description

bpvec applies FUN to subsets of X. Any type of object X is allowed, provided length, and [are defined on X. FUN is a function such that length(FUN(X)) == length(X). The objects returned by FUN are concatenated by AGGREGATE (c() by default). The return value is FUN(X).

Usage

```
bpvec(X, FUN, ..., AGGREGATE=c, BPREDO=list(), BPPARAM=bpparam(), BPOPTIONS = bpoptions())
```

Arguments

X	Any object for which methods `length` and `[` are implemented.
FUN	A function to be applied to subsets of `X`. The relationship between `X` and `FUN(X)` is 1:1, so that `length(FUN(X, ...)) == length(X)`. The return value of separate calls to `FUN` are concatenated with `AGGREGATE`.
...	Additional arguments for `FUN`.
AGGREGATE	A function taking any number of arguments `...` called to reduce results (elements of the `...` argument of `AGGREGATE` from parallel jobs. The default, `c`, concatenates objects and is appropriate for vectors; `rbind` might be appropriate for data frames.
BPPARAM	An optional `BiocParallelParam` instance determining the parallel back-end to be used during evaluation, or a `list` of `BiocParallelParam` instances, to be applied in sequence for nested calls to **BiocParallel** functions.
BPREDO	A `list` of output from `bpvec` with one or more failed elements. When a list is given in `BPREDO`, `bpok` is used to identify errors, tasks are rerun and inserted into the original results.
BPOPTIONS	Additional options to control the behavior of the parallel evaluation, see `bpoptions`.

Details

This method creates a vector of indices for `X` that divide the elements as evenly as possible given the number of `bpworkers()` and `bptasks()` of `BPPARAM`. Indices and data are passed to `bplapply` for parallel evaluation.

The distinction between `bpvec` and `bplapply` is that `bplapply` applies `FUN` to each element of `X` separately whereas `bpvec` assumes the function is vectorized, e.g., `c(FUN(x[1]), FUN(x[2]))` is equivalent to `FUN(x[1:2])`. This approach can be more efficient than `bplapply` but requires the assumption that `FUN` takes a vector input and creates a vector output of the same length as the input which does not depend on partitioning of the vector. This behavior is consistent with `parallel:::pvec` and the `?pvec` man page should be consulted for further details.

Value

The result should be identical to `FUN(X, ...)` (assuming that `AGGREGATE` is set appropriately).

When evaluation of individual elements of `X` results in an error, the result is a `list` with the same geometry (i.e., `lengths()`) as the split applied to `X` to create chunks for parallel evaluation; one or more elements of the list contain a `bperror` element, indicting that the vectorized calculation failed for at least one of the index values in that chunk.

An error is also signaled when `FUN(X)` does not return an object of the same length as `X`; this condition is only detected when the number of elements in `X` is greater than the number of workers.

Author(s)

Martin Morgan mailto:mtmorgan@fhcrc.org.

See Also

bplapply for parallel lapply.

BiocParallelParam for possible values of BPPARAM.

pvec for background.

Examples

```
methods("bpvec")

## ten tasks (1:10), called with as many back-end elements are specified
## by BPPARAM.  Compare with bplapply
fun <- function(v) {
    message("working")
    sqrt(v)
}
system.time(result <- bpvec(1:10, fun))
result

## invalid FUN -- length(class(X)) is not equal to length(X)
bptry(bpvec(1:2, class, BPPARAM=SerialParam()))
```

---

bpvectorize *Transform vectorized functions into parallelized, vectorized function*

Description

This transforms a vectorized function into a parallel, vectorized function. Any function FUN can be used, provided its parallelized argument (by default, the first argument) has a length and [method defined, and the return value of FUN can be concatenated with c.

Usage

```
bpvectorize(FUN, ..., BPREDO=list(), BPPARAM=bpparam(), BPOPTIONS = bpoptions())

## S4 method for signature 'ANY,ANY'
bpvectorize(FUN, ..., BPREDO=list(), BPPARAM=bpparam(), BPOPTIONS = bpoptions())

## S4 method for signature 'ANY,missing'
bpvectorize(FUN, ..., BPREDO=list(),
    BPPARAM=bpparam(), BPOPTIONS = bpoptions())
```

Arguments

FUN	A function whose first argument has a length and can be subset [, and whose evaluation would benefit by splitting the argument into subsets, each one of which is independently transformed by FUN. The return value of FUN must support concatenation with c.

...	Additional arguments to parallization, unused.
BPPARAM	An optional BiocParallelParam instance determining the parallel back-end to be used during evaluation.
BPREDO	A list of output from bpvectorize with one or more failed elements. When a list is given in BPREDO, bpok is used to identify errors, tasks are rerun and inserted into the original results.
BPOPTIONS	Additional options to control the behavior of the parallel evaluation, see bpoptions.

Details

The result of bpvectorize is a function with signature ...; arguments to the returned function are the original arguments FUN. BPPARAM is used for parallel evaluation.

When BPPARAM is a class for which no method is defined (e.g., SerialParam), FUN(X) is used.

See methods{bpvectorize} for additional methods, if any.

Value

A function taking the same arguments as FUN, but evaluated using bpvec for parallel evaluation across available cores.

Author(s)

Ryan Thompson mailto:rct@thompsonclan.org

See Also

bpvec

Examples

```
psqrt <- bpvectorize(sqrt) ## default parallelization
psqrt(1:10)
```

DeveloperInterface *Developer interface*

Description

Functions documented on this page are meant for developers wishing to implement BPPARAM objects that extend the BiocParallelParam virtual class to support additional parallel back-ends.

Usage

```
## class extension

.prototype_update(prototype, ...)

## manager interface

.send_to(backend, node, value)
.recv_any(backend)
.send_all(backend, value)
.recv_all(backend)

## worker interface

.send(worker, value)
.recv(worker)
.close(worker)

## task manager interface(optional)
.manager(BPPARAM)
.manager_send(manager, value, ...)
.manager_recv(manager)
.manager_send_all(manager, value)
.manager_recv_all(manager)
.manager_capacity(manager)
.manager_flush(manager)
.manager_cleanup(manager)

## supporting implementations

.bpstart_impl(x)
.bpworker_impl(worker)
.bplapply_impl(
    X, FUN, ..., BPREDO = list(),
    BPPARAM = bpparam(), BPOPTIONS = bpoptions()
)
.bpiterate_impl(
    ITER, FUN, ..., REDUCE, init, reduce.in.order = FALSE, BPREDO = list(),
    BPPARAM = bpparam(), BPOPTIONS = bpoptions()
)
.bpstop_impl(x)


## extract the static or dynamic part from a task
.task_const(value)
.task_dynamic(value)
.task_remake(value, static_data = NULL)
```

```
## Register an option for BPPARAM
.registerOption(optionName, genericName)
```

Arguments

prototype	A named list of default values for reference class fields.
x	A BPPARAM instance.
backend	An object containing information about the cluster, returned by bpbackend(<BPPARAM>).
manager	An object returned by .manager()
worker	The object to which the worker communicates via .send and .recv. .close terminates the worker.
node	An integer value indicating the node in the backend to which values are to be sent or received.
value	Any R object, to be sent to or from workers.
X, ITER, FUN, REDUCE, init, reduce.in.order, BPREDO, BPPARAM	See bplapply and bpiterate.
...	For .prototype_update(), name-value pairs to initialize derived and base class fields. For .bplapply_impl(), .bpiterate_impl(), additional arguments to FUN(); see bplapply and bpiterate. For .manager_send(), this is a placeholder for the future development.
static_data	An object extracted from .task_const(value)
BPOPTIONS	Additional options to control the behavior of parallel evaluation, see bpoptions.
optionName	character(1), an option name for BPPARAM. The named options will be created by bpoptions
genericName	character(1), the name of the S4 generic function. This will be used to get or set the field in BPPARAM. The generic needs to support replacement function defined by setReplaceMethod.

Details

Start a BPPARM implementation by creating a reference class, e.g., extending the virtual class BiocParallelParam. Because of idiosyncracies in reference class field initialization, an instance of the class should be created by calling the generator returned by setRefClass() with a list of key-value pairs providing default parameteter arguments. The default values for the BiocParallelParam base class is provided in a list .BiocParallelParam_prototype, and the function .prototype_update() updates a prototype with new values, typically provided by the user. See the example below.

BPPARAM implementations need to implement bpstart() and bpstop() methods; they may also need to implement, bplapply() and bpiterate() methods. Each method usually performs implementation-specific functionality before calling the next (BiocParallelParam) method. To avoid the intricacies of multiple dispatch, the bodies of BiocParallelParam methods are available for direct use as exported symbols.

- bpstart,BiocParallelParam-method (.bpstart_impl()) initiates logging, random number generation, and registration of finalizers to ensure that started clusters are stopped.

- bpstop,BiocParallelParam-method (.bpstop_impl()) ensures appropriate clean-up of stopped clusters, including sending the DONE semaphore. bpstart() will usually arrange for workers to enter .bpworker_impl() to listen for and evaluate tasks.
- bplapply,ANY,BiocParallelParam-method and bpiterate,ANY,BiocParallelParam-method (.bplapply_impl(), .bpiterate_impl()) implement: serial evaluation when there is a single core or task available; BPREDO functionality, and parallel lapply-like or iterative calculation.

Invoke .bpstart_impl(), .bpstop_impl(), .bplapply_impl(), and .bpiterate_impl() after any BPPARAM-specific implementation details.

New implementations will also implement bpisup() and bpbackend() / bpbackend<-(); there are no default methods.

The *backends* (object returned by bpbackend()) of new BPPARAM implementations must support length() (number of nodes). In addition, the backends must support .send_to() and .recv_any() manager and .send(), .recv(), and .close() worker methods. Default .send_all() and .recv_all() methods are implemented as simple iterations along the length(cluster), invoking .send_to() or .recv_any() on each iteration.

The task manager is an optional interface for a backend that wants to control the task dispatching process. .manager_send() sends the task value to a worker, .manager_recv() returns a list with each element being a result received from a worker. .manager_capacity() instructs how many tasks values can be processed simultaneously by the cluster. .manager_flush() flushes all the cached tasks(if any) immediately. .manager_cleanup() performs cleanup after the job is finished. The default methods for .manager_flush() and .manager_cleanup() are no-op.

In some cases it might be worth-while to cache some objects in a task and reuse them in another task. This can reduce the bandwith requirement for sending the tasks out to the worker. .task_const() can be used to extract the objects from the task which are not going to change across all tasks. .task_dynamic() preserve only the dynamic components in a task. Given the static and dynamic task objects, the complete task can be recovered by .task_remake(). When there is no static data can be extracted(e.g. a task with no static component or a task which has been extracted by .task_dynamic()), .task_const() simply returns a NULL value. Calling .task_remake() is no-op if the task haven't been extracted by .task_dynamic() or the static data is NULL.

The function .registerOption allows the developer to register a generic function that can change the fields in BPPARAM. The developer does not need to register the fields that are already defined in BiocParallel. .registerOption should only be used to support new fields. For example, if the developer defines a BPPARAM which has a field worker.password, the developer may also define the getter / setter bpworkerPassword and bpworkerPassword<-. Then by calling .registerOption("worker.password", "bpworkerPassword"), the user can change the field in BPPARAM by passing an object of bpoptions(worker.password = "1234") in any apply function.

Value

The return value of .prototype_update() is a list with elements in prototype substituted with key-value pairs provided in ....

All send* and recv* functions are endomorphic, returning a cluster object.

The return value of .manager_recv() is a list with each element being a result received from a worker, .manager_capacity() is an integer. The return values of the other .manager_*() are not restricted

Examples

```
##
## Extend BiocParallelParam; `.A()` is not meant for the end user
##

.A <- setRefClass(
    "A",
    contains = "BiocParallelParam",
    fields = list(id = "character")
)

## Use a prototype for default values, including the prototype for
## inheritted fields

.A_prototype <- c(
    list(id = "default_id"),
    .BiocParallelParam_prototype
)

## Provide a constructor for the user

A <- function(...) {
    prototype <- .prototype_update(.A_prototype, ...)
    do.call(.A, prototype)
}

## Provide an R function for field access

bpid <- function(x)
    x$id

## Create and use an instance, overwriting default values

bpid(A())

a <- A(id = "my_id", threshold = "WARN")
bpid(a)
bpthreshold(a)
```

DoparParam-class *Enable parallel evaluation using registered dopar backend*

Description

This class is used to dispatch parallel operations to the dopar backend registered with the foreach package.

Usage

```
DoparParam(stop.on.error=TRUE,
           RNGseed = NULL)
```

Arguments

`stop.on.error` `logical(1)`

Stop all jobs as soon as one jobs fails (`stop.on.error == TRUE`) or wait for all jobs to terminate. Default is `TRUE`.

`RNGseed` `integer(1)` Seed for random number generation. The seed is used to set a new, independent random number stream for each element of `X`. The ith element recieves the same stream seed, regardless of use of `SerialParam()`, `SnowParam()`, or `MulticoreParam()`, and regardless of worker or task number. When `RNGseed = NULL`, a random seed is used.

Details

`DoparParam` can be used for shared or non-shared memory computing depending on what backend is loaded. The `doSNOW` package supports non-shared memory, `doParallel` supports both shared and non-shared. When not specified, the default number of workers in `DoparParam` is determined by `getDoParWorkers()`. See the `foreach` package vignette for details using the different backends:

http://cran.r-project.org/web/packages/foreach/vignettes/foreach.pdf

DoparParam constructor

Return a proxy object that dispatches parallel evaluation to the registered foreach parallel backend.

There are no options to the constructor. All configuration should be done through the normal interface to the foreach parallel backends.

Methods

The following generics are implemented and perform as documented on the corresponding help page (e.g., ?bpisup): bpworkers, bpnworkers, bpstart, bpstop, bpisup, bpbackend, bpbackend<-, bpvec.

Author(s)

Martin Morgan mailto:mtmorgan@fhcrc.org

See Also

`getClass("BiocParallelParam")` for additional parameter classes.

`register` for registering parameter classes for use in parallel evaluation.

`foreach-package` for the parallel backend infrastructure used by this param class.

Examples

```
## Not run:
# First register a parallel backend with foreach
library(doParallel)
registerDoParallel(2)

p <- DoparParam()
bplapply(1:10, sqrt, BPPARAM=p)
bpvec(1:10, sqrt, BPPARAM=p)

## set DoparParam() as the default for BiocParallel
## register(DoparParam(), default=TRUE)

## End(Not run)
```

`ipcmutex` *Inter-process locks and counters*

Description

Functions documented on this page enable locks and counters between processes on the *same* computer.

Use `ipcid()` to generate a unique mutex or counter identifier. A mutex or counter with the same `id`, including those in different processes, share the same state.

`ipcremove()` removes external state associated with mutex or counters created with `id`.

`ipclock()` blocks until the lock is obtained. `ipctrylock()` tries to obtain the lock, returning immediately if it is not available. `ipcunlock()` releases the lock. `ipclocked()` queries the lock to determine whether it is currently held.

`ipcyield()` returns the current counter, and increments the value for subsequent calls. `ipcvalue()` returns the current counter without incrementing. `ipcreset()` sets the counter to `n`, such that the next call to `ipcyield()` or `ipcvalue()` returns `n`.

Usage

```
## Utilities

ipcid(id)

ipcremove(id)

## Locks

ipclock(id)

ipctrylock(id)
```

```
ipcunlock(id)

ipclocked(id)

## Counters

ipcyield(id)

ipcvalue(id)

ipcreset(id, n = 1)
```

Arguments

id	character(1) identifier string for mutex or counter. `ipcid()` ensures that the identifier is universally unique.
n	integer(1) value from which `ipcyield()` will increment.

Value

Locks:

`ipclock()` creates a named lock, returning `TRUE` on success.

`trylock()` returns `TRUE` if the lock is obtained, `FALSE` otherwise.

`ipcunlock()` returns `TRUE` on success, `FALSE` (e.g., because there is nothing to unlock) otherwise.

`ipclocked()` returns `TRUE` when `id` is locked, and `FALSE` otherwise.

Counters:

`ipcyield()` returns an integer(1) value representing the next number in sequence. The first value returned is 1.

`ipcvalue()` returns the value to be returned by the next call to `ipcyield()`, without incrementing the counter. If the counter is no longer available, `ipcyield()` returns `NA`.

`ipcreset()` returns `n`, invisibly.

Utilities:

`ipcid()` returns a character(1) unique identifier, with `id` (if not missing) prepended.

`ipcremove()` returns (invisibly) `TRUE` if external resources were released or `FALSE` if not (e.g., because the resources has already been released).

Examples

```
ipcid()

## Locks

id <- ipcid()

ipclock(id)
```

```
ipctrylock(id)
ipcunlock(id)
ipctrylock(id)
ipclocked(id)

ipcremove(id)

id <- ipcid()
system.time({
    ## about 1s, .2s for each process instead of .2s if no lock
    result <- bplapply(1:2, function(i, id) {
        BiocParallel::ipclock(id)
        Sys.sleep(.2)
        time <- Sys.time()
        BiocParallel::ipcunlock(id)
        time
    }, id)
})
ipcremove(id)
diff(sort(unlist(result, use.names=FALSE)))

## Counters

id <- ipcid()

ipcyield(id)
ipcyield(id)

ipcvalue(id)
ipcyield(id)

ipcreset(id, 10)
ipcvalue(id)
ipcyield(id)

ipcremove(id)

id <- ipcid()
result <- bplapply(1:2, function(i, id) {
    BiocParallel::ipcyield(id)
}, id)
ipcremove(id)
sort(unlist(result, use.names=FALSE))
```

`MulticoreParam-class` *Enable multi-core parallel evaluation*

Description

This class is used to parameterize single computer multicore parallel evaluation on non-Windows computers. `multicoreWorkers()` chooses the number of workers.

Usage

```
## constructor
## -----------------------------------

MulticoreParam(workers = multicoreWorkers(), tasks = 0L,
               stop.on.error = TRUE,
               progressbar = FALSE, RNGseed = NULL,
               timeout = WORKER_TIMEOUT, exportglobals=TRUE,
               log = FALSE, threshold = "INFO", logdir = NA_character_,
               resultdir = NA_character_, jobname = "BPJOB",
               force.GC = FALSE, fallback = TRUE,
               manager.hostname = NA_character_, manager.port = NA_integer_,
               ...)

## detect workers
## -----------------------------------

multicoreWorkers()
```

Arguments

workers	integer(1) Number of workers. Defaults to the maximum of 1 or the number of cores determined by detectCores minus 2 unless environment variables R_PARALLELLY_AVAILABLECORES_FALLBACK or BIOCPARALLEL_WORKER_NUMBER are set otherwise.
tasks	integer(1). The number of tasks per job. value must be a scalar integer >= 0L. In this documentation a job is defined as a single call to a function, such as bplapply, bpmapply etc. A task is the division of the X argument into chunks. When tasks == 0 (default, except when progressbar = TRUE), X is divided as evenly as possible over the number of workers. A tasks value of > 0 specifies the exact number of tasks. Values can range from 1 (all of X to a single worker) to the length of X (each element of X to a different worker). When the length of X is less than the number of workers each element of X is sent to a worker and tasks is ignored. When the length of X is less than tasks, tasks is treated as length(X).
stop.on.error	logical(1) Enable stop on error.
progressbar	logical(1) Enable progress bar (based on plyr:::progress_text). Enabling the progress bar changes the *default* value of tasks to .Machine$integer.max, so that progress is reported for each element of X.
RNGseed	integer(1) Seed for random number generation. The seed is used to set a new, independent random number stream for each element of X. The ith element recieves the same stream seed, regardless of use of SerialParam(), SnowParam(), or MulticoreParam(), and regardless of worker or task number. When RNGseed = NULL, a random seed is used.

timeout	numeric(1) Time (in seconds) allowed for worker to complete a task. This value is passed to base::setTimeLimit() as both the cpu and elapsed arguments. If the computation exceeds timeout an error is thrown with message 'reached elapsed time limit'.
exportglobals	logical(1) Export base::options() from manager to workers? Default TRUE.
log	logical(1) Enable logging.
threshold	character(1) Logging threshold as defined in futile.logger.
logdir	character(1) Log files directory. When not provided, log messages are returned to stdout.
resultdir	character(1) Job results directory. When not provided, results are returned as an R object (list) to the workspace.
jobname	character(1) Job name that is prepended to log and result files. Default is "BPJOB".
force.GC	logical(1) Whether to invoke the garbage collector after each call to FUN. The value TRUE, explicitly call the garbage collection, can slow parallel computation, but is necessary when each call to FUN allocates a 'large' amount of memory. If FUN allocates little memory, then considerable performance improvements are gained by the default setting force.GC = FALSE.
fallback	logical(1) When TRUE, fall back to using SerialParam when MulticoreParam has not been started and the number of worker is no greater than 1.
manager.hostname	character(1) Host name of manager node. See 'Global Options', in SnowParam.
manager.port	integer(1) Port on manager with which workers communicate. See 'Global Options' in SnowParam.
...	Additional arguments passed to makeCluster

Details

MulticoreParam is used for shared memory computing. Under the hood the cluster is created with makeCluster(..., type ="FORK") from the parallel package.

See ?BIOCPARALLEL_WORKER_NUMBER to control the default and maximum number of workers.

A FORK transport starts workers with the mcfork function and communicates between master and workers using socket connections. mcfork builds on fork() and thus a Linux cluster is not supported. Because FORK clusters are Posix based they are not supported on Windows. When MulticoreParam is created/used in Windows it defaults to SerialParam which is the equivalent of using a single worker.

error handling: By default all computations are attempted and partial results are returned with any error messages.

- stop.on.error A logical. Stops all jobs as soon as one job fails or wait for all jobs to terminate. When FALSE, the return value is a list of successful results along with error messages as 'conditions'.
- The bpok(x) function returns a logical() vector that is FALSE for any jobs that threw an error. The input x is a list output from a bp*apply function such as bplapply or bpmapply.

logging: When log = TRUE the futile.logger package is loaded on the workers. All log messages written in the futile.logger format are captured by the logging mechanism and returned in real-time (i.e., as each task completes) instead of after all jobs have finished.

Messages sent to *stdout* and *stderr* are returned to the workspace by default. When log = TRUE these are diverted to the log output. Those familiar with the outfile argument to makeCluster can think of log = FALSE as equivalent to outfile = NULL; providing a logdir is the same as providing a name for outfile except that BiocParallel writes a log file for each task.

The log output includes additional statistics such as memory use and task runtime. Memory use is computed by calling gc(reset=TRUE) before code evaluation and gc() (no reseet) after. The output of the second gc() call is sent to the log file.

log and result files: Results and logs can be written to a file instead of returned to the workspace. Writing to files is done from the master as each task completes. Options can be set with the logdir and resultdir fields in the constructor or with the accessors, bplogdir and bpresultdir.

random number generation: For MulticoreParam, SnowParam, and SerialParam, random number generation is controlled through the RNGseed = argument. BiocParallel uses the L'Ecuyer-CMRG random number generator described in the parallel package to generate independent random number streams. One stream is associated with each element of X, and used to seed the random number stream for the application of FUN() to X[[i]]. Thus setting RNGseed = ensures reproducibility across MulticoreParam(), SnowParam(), and SerialParam(), regardless of worker or task number. The default value RNGseed = NULL means that each evaluation of bplapply proceeds independently.

For details of the L'Ecuyer generator, see ?clusterSetRNGStream.

Constructor

MulticoreParam(workers = multicoreWorkers(), tasks = 0L, stop.on.error = FALSE, tasks = 0L, progressbar =
Return an object representing a FORK cluster. The cluster is not created until bpstart is called. Named arguments in ... are passed to makeCluster.

Accessors: Logging and results

In the following code, x is a MulticoreParam object.

bpprogressbar(x), bpprogressbar(x) <- value: Get or set the value to enable text progress bar. value must be a logical(1).

bpjobname(x), bpjobname(x) <- value: Get or set the job name.

bpRNGseed(x), bpRNGseed(x) <- value: Get or set the seed for random number generaton. value must be a numeric(1) or NULL.

bplog(x), bplog(x) <- value: Get or set the value to enable logging. value must be a logical(1).

bpthreshold(x), bpthreshold(x) <- value: Get or set the logging threshold. value must be a character(1) string of one of the levels defined in the futile.logger package: "TRACE", "DEBUG", "INFO", "WARN", "ERROR", or "FATAL".

bplogdir(x), bplogdir(x) <- value: Get or set the directory for the log file. value must be a character(1) path, not a file name. The file is written out as LOGFILE.out. If no logdir is provided and bplog=TRUE log messages are sent to stdout.

`bpresultdir(x)`, `bpresultdir(x) <- value`: Get or set the directory for the result files. `value` must be a `character(1)` path, not a file name. Separate files are written for each job with the prefix JOB (e.g., JOB1, JOB2, etc.). When no `resultdir` is provided the results are returned to the session as `list`.

Accessors: Back-end control

In the code below `x` is a `MulticoreParam` object. See the ?`BiocParallelParam` man page for details on these accessors.

- `bpworkers(x)`
- `bpnworkers(x)`
- `bptasks(x)`, `bptasks(x) <- value`
- `bpstart(x)`
- `bpstop(x)`
- `bpisup(x)`
- `bpbackend(x)`, `bpbackend(x) <- value`

Accessors: Error Handling

In the code below `x` is a `MulticoreParam` object. See the ?`BiocParallelParam` man page for details on these accessors.

- `bpstopOnError(x)`, `bpstopOnError(x) <- value`

Methods: Evaluation

In the code below `BPPARAM` is a `MulticoreParam` object. Full documentation for these functions are on separate man pages: see ?`bpmapply`, ?`bplapply`, ?`bpvec`, ?`bpiterate` and ?`bpaggregate`.

- `bpmapply(FUN, ..., MoreArgs=NULL, SIMPLIFY=TRUE, USE.NAMES=TRUE, BPPARAM=bpparam())`
- `bplapply(X, FUN, ..., BPPARAM=bpparam())`
- `bpvec(X, FUN, ..., AGGREGATE=c, BPPARAM=bpparam())`
- `bpiterate(ITER, FUN, ..., BPPARAM=bpparam())`
- `bpaggregate(x, data, FUN, ..., BPPARAM=bpparam())`

Methods: Other

In the code below `x` is a `MulticoreParam` object.

`show(x)`: Displays the `MulticoreParam` object.

Global Options

See the 'Global Options' section of `SnowParam` for manager host name and port defaults.

Author(s)

Martin Morgan mailto:mtmorgan@fhcrc.org and Valerie Obenchain

See Also

- `register` for registering parameter classes for use in parallel evaluation.
- SnowParam for computing in distributed memory
- DoparParam for computing with foreach
- SerialParam for non-parallel evaluation

Examples

```
## ----------------------------------------------------------------------
## Job configuration:
## ----------------------------------------------------------------------

## MulticoreParam supports shared memory computing. The object fields
## control the division of tasks, error handling, logging and
## result format.
bpparam <- MulticoreParam()
bpparam

## By default the param is created with the maximum available workers
## determined by multicoreWorkers().
multicoreWorkers()

## Fields are modified with accessors of the same name:
bplog(bpparam) <- TRUE
dir.create(resultdir <- tempfile())
bpresultdir(bpparam) <- resultdir
bpparam

## ----------------------------------------------------------------------
## Logging:
## ----------------------------------------------------------------------

## When 'log == TRUE' the workers use a custom script (in BiocParallel)
## that enables logging and access to other job statistics. Log messages
## are returned as each job completes rather than waiting for all to finish.

## In 'fun', a value of 'x = 1' will throw a warning, 'x = 2' is ok
## and 'x = 3' throws an error. Because 'x = 1' sleeps, the warning
## should return after the error.

X <- 1:3
fun <- function(x) {
    if (x == 1) {
        Sys.sleep(2)
        sqrt(-x)                  ## warning
        x
    } else if (x == 2) {
        x                         ## ok
    } else if (x == 3) {
        sqrt("FOO")               ## error
    }
```

```
}

## By default logging is off. Turn it on with the bplog()<- setter
## or by specifying 'log = TRUE' in the constructor.
bpparam <- MulticoreParam(3, log = TRUE, stop.on.error = FALSE)
res <- tryCatch({
    bplapply(X, fun, BPPARAM=bpparam)
}, error=identity)
res

## When a 'logdir' location is given the messages are redirected to a file:
## Not run:
bplogdir(bpparam) <- tempdir()
bplapply(X, fun, BPPARAM = bpparam)
list.files(bplogdir(bpparam))

## End(Not run)

## ------------------------------------------------------------------------
## Managing results:
## ------------------------------------------------------------------------

## By default results are returned as a list. When 'resultdir' is given
## files are saved in the directory specified by job, e.g., 'TASK1.Rda',
## 'TASK2.Rda', etc.
## Not run:
dir.create(resultdir <- tempfile())
bpparam <- MulticoreParam(2, resultdir = resultdir, stop.on.error = FALSE)
bplapply(X, fun, BPPARAM = bpparam)
list.files(bpresultdir(bpparam))

## End(Not run)

## ------------------------------------------------------------------------
## Error handling:
## ------------------------------------------------------------------------

## When 'stop.on.error' is TRUE the job is terminated as soon as an
## error is hit. When FALSE, all computations are attempted and partial
## results are returned along with errors. In this example the number of
## 'tasks' is set to equal the length of 'X' so each element is run
## separately. (Default behavior is to divide 'X' evenly over workers.)

## All results along with error:
bpparam <- MulticoreParam(2, tasks = 4, stop.on.error = FALSE)
res <- bptry(bplapply(list(1, "two", 3, 4), sqrt, BPPARAM = bpparam))
res

## Calling bpok() on the result list returns TRUE for elements with no error.
bpok(res)

## ------------------------------------------------------------------------
## Random number generation:
```

```
## ------------------------------------------------------------------------

## Random number generation is controlled with the 'RNGseed' field.
## This seed is passed to parallel::clusterSetRNGStream
## which uses the L'Ecuyer-CMRG random number generator and distributes
## streams to members of the cluster.

bpparam <- MulticoreParam(3, RNGseed = 7739465)
bplapply(seq_len(bpnworkers(bpparam)), function(i) rnorm(1), BPPARAM = bpparam)
```

`register` *Maintain a global registry of available back-end Params*

Description

Use functions on this page to add to or query a registry of back-ends, including the default for use when no `BPPARAM` object is provided to functions.

Usage

```
register(BPPARAM, default=TRUE)
registered(bpparamClass)
bpparam(bpparamClass)
```

Arguments

`BPPARAM`	An instance of a `BiocParallelParam` class, e.g., `MulticoreParam`, `SnowParam`, `DoparParam`.
`default`	Make this the default `BiocParallelParam` for subsequent evaluations? If `FALSE`, the argument is placed at the lowest priority position.
`bpparamClass`	When present, the text name of the `BiocParallelParam` class (e.g., "MulticoreParam") to be retrieved from the registry. When absent, a list of all registered instances is returned.

Details

The registry is a list of back-ends with configuration parameters for parallel evaluation. The first list entry is the default and is used by `BiocParallel` functions when no `BPPARAM` argument is supplied.

At load time the registry is populated with default backends. On Windows these are `SnowParam` and `SerialParam` and on non-Windows `MulticoreParam`, `SnowParam` and `SerialParam`. When `snowWorkers()` or `multicoreWorkers` returns a single core, only `SerialParm` is registered.

The `BiocParallelParam` objects are constructed from global options of the corresponding name, or from the default constructor (e.g., `SnowParam()`) if no option is specified. The user can set customizations during start-up (e.g., in an `.Rprofile` file) with, for instance, `options(MulticoreParam=quote(MulticorePar`

The act of "registering" a back-end modifies the existing `BiocParallelParam` in the list; only one param of each type can be present in the registry. When `default=TRUE`, the newly registered param

is moved to the top of the list thereby making it the default. When `default=FALSE`, the param is modified 'in place' vs being moved to the top.

`bpparam()`, invoked with no arguments, returns the default `BiocParallelParam` instance from the registry. When called with the text name of a `bpparamClass`, the global options are consulted first, e.g., `options(MulticoreParam=MulticoreParam())` and then the value of `registered(bpparamClass)`.

Value

`register` returns, invisibly, a list of registered back-ends.

`registered` returns the back-end of type `bpparamClass` or, if `bpparamClass` is missing, a list of all registered back-ends.

`bpparam` returns the back-end of type `bpparamClass` or,

Author(s)

Martin Morgan mailto:mtmorgan@fhcrc.org.

See Also

`BiocParallelParam` for possible values of `BPPARAM`.

Examples

```
## ------------------------------------------------------------------------
## The registry
## ------------------------------------------------------------------------

## The default registry.
default <- registered()
default

## When default = TRUE the last param registered becomes the new default.
snowparam <- SnowParam(workers = 3, type = "SOCK")
register(snowparam, default = TRUE)
registered()

## Retrieve the default back-end,
bpparam()

## or a specific BiocParallelParam.
bpparam("SnowParam")

## restore original registry -- push the defaults in reverse order
for (param in rev(default))
    register(param)

## ------------------------------------------------------------------------
## Specifying a back-end for evaluation
## ------------------------------------------------------------------------

## The back-end of choice is given as the BPPARAM argument to
```

```
## the BiocParallel functions. None, one, or multiple back-ends can be
## used.

bplapply(1:6, sqrt, BPPARAM = MulticoreParam(3))

## When not specified, the default from the registry is used.
bplapply(1:6, sqrt)
```

SerialParam-class *Enable serial evaluation*

Description

This class is used to parameterize serial evaluation, primarily to facilitate easy transition from parallel to serial code.

Usage

```
SerialParam(
    stop.on.error = TRUE,
    progressbar = FALSE,
    RNGseed = NULL,
    timeout = WORKER_TIMEOUT,
    log = FALSE,
    threshold = "INFO",
    logdir = NA_character_,
    resultdir = NA_character_,
    jobname = "BPJOB",
    force.GC = FALSE
)
```

Arguments

stop.on.error	logical(1) Enable stop on error.
progressbar	logical(1) Enable progress bar (based on plyr:::progress_text).
RNGseed	integer(1) Seed for random number generation. The seed is used to set a new, independent random number stream for each element of X. The ith element recieves the same stream seed, regardless of use of SerialParam(), SnowParam(), or MulticoreParam(), and regardless of worker or task number. When RNGseed = NULL, a random seed is used.
timeout	numeric(1) Time (in seconds) allowed for worker to complete a task. This value is passed to base::setTimeLimit() as both the cpu and elapsed arguments. If the computation exceeds timeout an error is thrown with message 'reached elapsed time limit'.
log	logical(1) Enable logging.

threshold	character(1) Logging threshold as defined in futile.logger.
logdir	character(1) Log files directory. When not provided, log messages are returned to stdout.
resultdir	character(1) Job results directory. When not provided, results are returned as an R object (list) to the workspace.
jobname	character(1) Job name that is prepended to log and result files. Default is "BPJOB".
force.GC	logical(1) Whether to invoke the garbage collector after each call to FUN. The default (FALSE, do not explicitly call the garbage collection) rarely needs to be changed.

Details

SerialParam is used for serial computation on a single node. Using SerialParam in conjunction with bplapply differs from use of lapply because it provides features such as error handling, logging, and random number use consistent with SnowParam and MulticoreParam.

error handling: By default all computations are attempted and partial results are returned with any error messages.

- stop.on.error A logical. Stops all jobs as soon as one job fails or wait for all jobs to terminate. When FALSE, the return value is a list of successful results along with error messages as 'conditions'.
- The bpok(x) function returns a logical() vector that is FALSE for any jobs that threw an error. The input x is a list output from a bp*apply function such as bplapply or bpmapply.

logging: When log = TRUE the futile.logger package is loaded on the workers. All log messages written in the futile.logger format are captured by the logging mechanism and returned real-time (i.e., as each task completes) instead of after all jobs have finished.

Messages sent to *stdout* and *stderr* are returned to the workspace by default. When log = TRUE these are diverted to the log output. Those familiar with the outfile argument to makeCluster can think of log = FALSE as equivalent to outfile = NULL; providing a logdir is the same as providing a name for outfile except that BiocParallel writes a log file for each task.

The log output includes additional statistics such as memory use and task runtime. Memory use is computed by calling gc(reset=TRUE) before code evaluation and gc() (no reseet) after. The output of the second gc() call is sent to the log file.

log and result files: Results and logs can be written to a file instead of returned to the workspace. Writing to files is done from the master as each task completes. Options can be set with the logdir and resultdir fields in the constructor or with the accessors, bplogdir and bpresultdir.

random number generation: For MulticoreParam, SnowParam, and SerialParam, random number generation is controlled through the RNGseed = argument. BiocParallel uses the L'Ecuyer-CMRG random number generator described in the parallel package to generate independent random number streams. One stream is associated with each element of X, and used to seed the

random number stream for the application of FUN() to X[[i]]. Thus setting RNGseed = ensures reproducibility across MulticoreParam(), SnowParam(), and SerialParam(), regardless of worker or task number. The default value RNGseed = NULL means that each evaluation of bplapply proceeds independently.

For details of the L'Ecuyer generator, see ?clusterSetRNGStream.

Constructor

SerialParam(): Return an object to be used for serial evaluation of otherwise parallel functions such as bplapply, bpvec.

Methods

The following generics are implemented and perform as documented on the corresponding help page (e.g., ?bpworkers): bpworkers. bpisup, bpstart, bpstop, are implemented, but do not have any side-effects.

Author(s)

Martin Morgan mailto:mtmorgan@fhcrc.org

See Also

getClass("BiocParallelParam") for additional parameter classes.

register for registering parameter classes for use in parallel evaluation.

Examples

```
p <- SerialParam()
simplify2array(bplapply(1:10, sqrt, BPPARAM=p))
bpvec(1:10, sqrt, BPPARAM=p)
```

SnowParam-class *Enable simple network of workstations (SNOW)-style parallel evaluation*

Description

This class is used to parameterize simple network of workstations (SNOW) parallel evaluation on one or several physical computers. snowWorkers() chooses the number of workers.

Usage

```
## constructor
## -----------------------------------

SnowParam(workers = snowWorkers(type), type=c("SOCK", "MPI", "FORK"),
    tasks = 0L, stop.on.error = TRUE,
    progressbar = FALSE, RNGseed = NULL,
    timeout = WORKER_TIMEOUT, exportglobals = TRUE, exportvariables = TRUE,
    log = FALSE, threshold = "INFO", logdir = NA_character_,
    resultdir = NA_character_, jobname = "BPJOB",
    force.GC = FALSE, fallback = TRUE,
    manager.hostname = NA_character_, manager.port = NA_integer_,
    ...)

## coercion
## -----------------------------------

## as(SOCKcluster, SnowParam)
## as(spawnedMPIcluster,SnowParam)

## detect workers
## -----------------------------------

snowWorkers(type = c("SOCK", "MPI", "FORK"))
```

Arguments

workers	integer(1) Number of workers. Defaults to the maximum of 1 or the number of cores determined by detectCores minus 2 unless environment variables R_PARALLELLY_AVAILABLECORES_FALLBACK or BIOCPARALLEL_WORKER_NUMBER are set otherwise. For a SOCK cluster, workers can be a character() vector of host names.
type	character(1) Type of cluster to use. Possible values are SOCK (default) and MPI. Instead of type=FORK use MulticoreParam.
tasks	integer(1). The number of tasks per job. value must be a scalar integer >= 0L. In this documentation a job is defined as a single call to a function, such as bplapply, bpmapply etc. A task is the division of the X argument into chunks. When tasks == 0 (default), X is divided as evenly as possible over the number of workers. A tasks value of > 0 specifies the exact number of tasks. Values can range from 1 (all of X to a single worker) to the length of X (each element of X to a different worker). When the length of X is less than the number of workers each element of X is sent to a worker and tasks is ignored.
stop.on.error	logical(1) Enable stop on error.
progressbar	logical(1) Enable progress bar (based on plyr:::progress_text).

RNGseed — integer(1) Seed for random number generation. The seed is used to set a new, independent random number stream for each element of X. The ith element recieves the same stream seed, regardless of use of SerialParam(), SnowParam(), or MulticoreParam(), and regardless of worker or task number. When RNGseed = NULL, a random seed is used.

timeout — numeric(1) Time (in seconds) allowed for worker to complete a task. This value is passed to base::setTimeLimit() as both the cpu and elapsed arguments. If the computation exceeds timeout an error is thrown with message 'reached elapsed time limit'.

exportglobals — logical(1) Export base::options() from manager to workers? Default TRUE.

exportvariables — logical(1) Automatically export the variables which are defined in the global environment and used by the function from manager to workers. Default TRUE.

log — logical(1) Enable logging.

threshold — character(1) Logging threshold as defined in futile.logger.

logdir — character(1) Log files directory. When not provided, log messages are returned to stdout.

resultdir — character(1) Job results directory. When not provided, results are returned as an R object (list) to the workspace.

jobname — character(1) Job name that is prepended to log and result files. Default is "BPJOB".

force.GC — logical(1) Whether to invoke the garbage collector after each call to FUN. The default (FALSE, do not explicitly call the garbage collection) rarely needs to be changed.

fallback — logical(1) When TRUE, fall back to using SerialParam when SnowParam has not been started and the number of worker is no greater than 1.

manager.hostname — character(1) Host name of manager node. See 'Global Options', below.

manager.port — integer(1) Port on manager with which workers communicate. See 'Global Options', below.

... — Additional arguments passed to makeCluster

Details

SnowParam is used for distributed memory computing and supports 2 cluster types: 'SOCK' (default) and 'MPI'. The SnowParam builds on infrastructure in the snow and parallel packages and provides the additional features of error handling, logging and writing out results.

See ?BIOCPARALLEL_WORKER_NUMBER to control the default and maximum number of workers.

error handling: By default all computations are attempted and partial results are returned with any error messages.

- stop.on.error A logical. Stops all jobs as soon as one job fails or wait for all jobs to terminate. When FALSE, the return value is a list of successful results along with error messages as 'conditions'.

- The bpok(x) function returns a logical() vector that is FALSE for any jobs that threw an error. The input x is a list output from a bp*apply function such as bplapply or bpmapply.

logging: When log = TRUE the futile.logger package is loaded on the workers. All log messages written in the futile.logger format are captured by the logging mechanism and returned real-time (i.e., as each task completes) instead of after all jobs have finished.

Messages sent to *stdout* and *stderr* are returned to the workspace by default. When log = TRUE these are diverted to the log output. Those familiar with the outfile argument to makeCluster can think of log = FALSE as equivalent to outfile = NULL; providing a logdir is the same as providing a name for outfile except that BiocParallel writes a log file for each task.

The log output includes additional statistics such as memory use and task runtime. Memory use is computed by calling gc(reset=TRUE) before code evaluation and gc() (no reseet) after. The output of the second gc() call is sent to the log file.

log and result files: Results and logs can be written to a file instead of returned to the workspace. Writing to files is done from the master as each task completes. Options can be set with the logdir and resultdir fields in the constructor or with the accessors, bplogdir and bpresultdir.

random number generation: For MulticoreParam, SnowParam, and SerialParam, random number generation is controlled through the RNGseed = argument. BiocParallel uses the L'Ecuyer-CMRG random number generator described in the parallel package to generate independent random number streams. One stream is associated with each element of X, and used to seed the random number stream for the application of FUN() to X[[i]]. Thus setting RNGseed = ensures reproducibility across MulticoreParam(), SnowParam(), and SerialParam(), regardless of worker or task number. The default value RNGseed = NULL means that each evaluation of bplapply proceeds independently.

For details of the L'Ecuyer generator, see ?clusterSetRNGStream.

NOTE: The PSOCK cluster from the parallel package does not support cluster options scriptdir and useRscript. PSOCK is not supported because these options are needed to re-direct to an alternate worker script located in BiocParallel.

Constructor

SnowParam(workers = snowWorkers(), type=c("SOCK", "MPI"), tasks = 0L, stop.on.error = FALSE, progressbar
: Return an object representing a SNOW cluster. The cluster is not created until bpstart is called. Named arguments in ... are passed to makeCluster.

Accessors: Logging and results

In the following code, x is a SnowParam object.

bpprogressbar(x), bpprogressbar(x) <- value**:** Get or set the value to enable text progress bar. value must be a logical(1).

bpjobname(x), bpjobname(x) <- value**:** Get or set the job name.

bpRNGseed(x), bpRNGseed(x) <- value**:** Get or set the seed for random number generaton. value must be a numeric(1) or NULL.

bplog(x), bplog(x) <- value**:** Get or set the value to enable logging. value must be a logical(1).

`bpthreshold(x)`, `bpthreshold(x) <- value`: Get or set the logging threshold. `value` must be a `character(1)` string of one of the levels defined in the `futile.logger` package: "TRACE", "DEBUG", "INFO", "WARN", "ERROR", or "FATAL".

`bplogdir(x)`, `bplogdir(x) <- value`: Get or set the directory for the log file. `value` must be a `character(1)` path, not a file name. The file is written out as BPLOG.out. If no `logdir` is provided and `bplog=TRUE` log messages are sent to stdout.

`bpresultdir(x)`, `bpresultdir(x) <- value`: Get or set the directory for the result files. `value` must be a `character(1)` path, not a file name. Separate files are written for each job with the prefix TASK (e.g., TASK1, TASK2, etc.). When no `resultdir` is provided the results are returned to the session as `list`.

Accessors: Back-end control

In the code below `x` is a `SnowParam` object. See the ?`BiocParallelParam` man page for details on these accessors.

- `bpworkers(x)`, `bpworkers(x) <- value`, `bpnworkers(x)`
- `bptasks(x)`, `bptasks(x) <- value`
- `bpstart(x)`
- `bpstop(x)`
- `bpisup(x)`
- `bpbackend(x)`, `bpbackend(x) <- value`

Accessors: Error Handling

In the code below `x` is a `SnowParam` object. See the ?`BiocParallelParam` man page for details on these accessors.

- `bpstopOnError(x)`, `bpstopOnError(x) <- value`

Methods: Evaluation

In the code below `BPPARAM` is a `SnowParam` object. Full documentation for these functions are on separate man pages: see ?`bpmapply`, ?`bplapply`, ?`bpvec`, ?`bpiterate` and ?`bpaggregate`.

- `bpmapply(FUN, ..., MoreArgs=NULL, SIMPLIFY=TRUE, USE.NAMES=TRUE, BPPARAM=bpparam())`
- `bplapply(X, FUN, ..., BPPARAM=bpparam())`
- `bpvec(X, FUN, ..., AGGREGATE=c, BPPARAM=bpparam())`
- `bpiterate(ITER, FUN, ..., BPPARAM=bpparam())`
- `bpaggregate(x, data, FUN, ..., BPPARAM=bpparam())`

Methods: Other

In the code below `x` is a `SnowParam` object.

`show(x)`: Displays the `SnowParam` object.

`bpok(x)`: Returns a `logical()` vector: FALSE for any jobs that resulted in an error. `x` is the result list output by a `BiocParallel` function such as `bplapply` or `bpmapply`.

Coercion

`as(from, "SnowParam")`: Creates a `SnowParam` object from a `SOCKcluster` or `spawnedMPIcluster` object. Instances created in this way cannot be started or stopped.

Global Options

The environment variable `BIOCPARALLEL_WORKER_NUMBER` and the the global option `mc.cores` influences the number of workers determined by `snowWorkers()` (described above) or `multicoreWorkers()` (see `multicoreWorkers`).

Workers communicate to the master through socket connections. Socket connections require a hostname and port. These are determined by arguments `manager.hostname` and `manager.port`; default values are influenced by global options.

The default manager hostname is "localhost" when the number of workers are specified as a `numeric(1)`, and `Sys.info()[["nodename"]]` otherwise. The hostname can be over-ridden by the envirnoment variable `MASTER`, or the global option `bphost` (e.g., `options(bphost=Sys.info()[["nodename"]])`.

The default port is chosen as a random value between 11000 and 11999. The port may be over-ridden by the environment variable `R_PARALLEL_PORT` or `PORT`, and by the option `ports`, e.g., `options(ports=12345L)`.

Author(s)

Martin Morgan and Valerie Obenchain.

See Also

- `register` for registering parameter classes for use in parallel evaluation.
- `MulticoreParam` for computing in shared memory
- `DoparParam` for computing with foreach
- `SerialParam` for non-parallel evaluation

Examples

```
## ----------------------------------------------------------------------
## Job configuration:
## ----------------------------------------------------------------------

## SnowParam supports distributed memory computing. The object fields
## control the division of tasks, error handling, logging and result
## format.
bpparam <- SnowParam()
bpparam

## Fields are modified with accessors of the same name:
bplog(bpparam) <- TRUE
dir.create(resultdir <- tempfile())
bpresultdir(bpparam) <- resultdir
bpparam
```

```
## ---------------------------------------------------------------------
## Logging:
## ---------------------------------------------------------------------

## When 'log == TRUE' the workers use a custom script (in BiocParallel)
## that enables logging and access to other job statistics. Log messages
## are returned as each job completes rather than waiting for all to
## finish.

## In 'fun', a value of 'x = 1' will throw a warning, 'x = 2' is ok
## and 'x = 3' throws an error. Because 'x = 1' sleeps, the warning
## should return after the error.

X <- 1:3
fun <- function(x) {
    if (x == 1) {
        Sys.sleep(2)
        log(-x)                     ## warning
    } else if (x == 2) {
        x                           ## ok
    } else if (x == 3) {
        sqrt("FOO")                 ## error
    }
}

## By default logging is off. Turn it on with the bplog()<- setter
## or by specifying 'log = TRUE' in the constructor.
bpparam <- SnowParam(3, log = TRUE, stop.on.error = FALSE)
tryCatch({
    bplapply(X, fun, BPPARAM = bpparam)
}, error=identity)

## When a 'logdir' location is given the messages are redirected to a
## file:
## Not run:
dir.create(logdir <- tempfile())
bplogdir(bpparam) <- logdir
bplapply(X, fun, BPPARAM = bpparam)
list.files(bplogdir(bpparam))

## End(Not run)

## ---------------------------------------------------------------------
## Managing results:
## ---------------------------------------------------------------------

## By default results are returned as a list. When 'resultdir' is given
## files are saved in the directory specified by job, e.g., 'TASK1.Rda',
## 'TASK2.Rda', etc.
## Not run:
dir.create(resultdir <- tempfile())
bpparam <- SnowParam(2, resultdir = resultdir)
bplapply(X, fun, BPPARAM = bpparam)
```

```
list.files(bpresultdir(bpparam))

## End(Not run)

## ------------------------------------------------------------------------
## Error handling:
## ------------------------------------------------------------------------

## When 'stop.on.error' is TRUE the process returns as soon as an error
## is thrown.

## When 'stop.on.error' is FALSE all computations are attempted. Partial
## results are returned along with errors. Use bptry() to see the
## partial results
bpparam <- SnowParam(2, stop.on.error = FALSE)
res <- bptry(bplapply(list(1, "two", 3, 4), sqrt, BPPARAM = bpparam))
res

## Calling bpok() on the result list returns TRUE for elements with no
## error.
bpok(res)

## ------------------------------------------------------------------------
## Random number generation:
## ------------------------------------------------------------------------

## Random number generation is controlled with the 'RNGseed' field.
## This seed is passed to parallel::clusterSetRNGStream
## which uses the L'Ecuyer-CMRG random number generator and distributes
## streams for each job

bpparam <- SnowParam(3, RNGseed = 7739465)
bplapply(seq_len(bpnworkers(bpparam)), function(i) rnorm(1),
         BPPARAM = bpparam)
```

`workers` *Environment control of worker number*

Description

Environment variables, global options, and aspects of the computing environment controlling default and maximum worker number.

Details

By default, BiocParallel `Param` objects use almost all (`parallel::detectCores() - 2`) available cores as workers. Several variables can determine alternative default number of workers. Elements earlier in the description below override elements later in the description.

`_R_CHECK_LIMIT_CORES_`: Environment variable defined in base R, described in the 'R Internals' manual (`RShowDoc("R-ints")`). If defined and not equal to `"false"` or `"FALSE"`, default to 2 workers.

`IS_BIOC_BUILD_MACHINE`: Environment variable used by the Bioconductor build system; when defined, default to 4 workers.

`getOption("mc.cores")`: Global R option (initialized from the environment variable `MC_CORES`) with non-negative integer number of workers, also recognized by the base R 'parallel' package.

`BIOCPARALLEL_WORKER_MAX`: Environment variable, non-negative integer number of workers. Use this to set both the default and maximum worker number to a single value.

`BIOCPARALLEL_WORKER_NUMBER`: Environment variable, non-negative integer number of workers. Use this to set a default worker number without specifying `BIOCPARALLEL_WORKER_MAX`, or to set a default number of workers less than the maximum number.

`R_PARALLELLY_AVAILABLECORES_FALLBACK`: Environment variable, non-negative integer number of workers, also recognized by the 'parallelly' family of packages.

A subset of environment variables and other aspects of the computing environment also *enforce* limits on worker number. Usually, a request for more than the maximum number of workers results in a warning message and creation of a 'Param' object with the maximum rather than requested number of workers.

`_R_CHECK_LIMIT_CORES_`: Environment variable defined in base R. `"warn"` limits the number of workers to 2, with a warning; `"false"`, or `"FALSE"` does not limit worker number; any other value generates an error.

`IS_BIOC_BUILD_MACHINE`: Environment variable used by the Bioconductor build system. When set, limit the number of workers to 4.

`BIOCPARALLEL_WORKER_MAX`: Environment variable, non-negative integer.

Number of available connections: R has an internal limit (126) on the number of connections open at any time. 'SnowParam()' and 'MulticoreParam()' use 1 connection per worker, and so are limited by the number of available connections.

Examples

```
## set up example
original_worker_max <- Sys.getenv("BIOCPARALLEL_WORKER_MAX", NA_integer_)
original_worker_n <- Sys.getenv("BIOCPARALLEL_WORKER_NUMBER", NA_integer_)
Sys.setenv(BIOCPARALLEL_WORKER_MAX = 4)
Sys.setenv(BIOCPARALLEL_WORKER_NUMBER = 2)

bpnworkers(SnowParam())  # 2
bpnworkers(SnowParam(4)) # OK
bpnworkers(SnowParam(5)) # warning; set to 4

## clean up
Sys.unsetenv("BIOCPARALLEL_WORKER_MAX")
if (!is.na(original_worker_max))
    Sys.setenv(BIOCPARALLEL_WORKER_MAX = original_worker_max)
Sys.unsetenv("BIOCPARALLEL_WORKER_NUMBER")
```

```
    if (!is.na(original_worker_n))
        Sys.setenv(BIOCPARALLEL_WORKER_NUMBER = original_worker_n)
```

Index