

# Package ‘BiocParallel’

May 23, 2015

**Type** Package

**Title** Bioconductor facilities for parallel evaluation

**Version** 1.2.2

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

**biocViews** Infrastructure

**License** GPL-2 | GPL-3

**Depends** methods

**Imports** futile.logger, parallel, snow

**Suggests** BiocGenerics, tools, foreach, BatchJobs, BBmisc, doParallel, Rmpi, GenomicRanges, RNAseqData.HNRNPC.bam.chr14, Rsamtools, GenomicAlignments, ShortRead, codetools, RUnit, BiocStyle, knitr

**Collate.unix** AllGenerics.R BiocParallelParam-class.R ErrorHandling.R  
bpbackend-methods.R bpsup-methods.R bplapply-methods.R bpmapply-methods.R  
bpiterate-methods.R bpschedule-methods.R bpstart-methods.R bpstop-methods.R  
bpvec-methods.R bpvectorize-methods.R bpworkers-methods.R  
bpaggregate-methods.R bpvalidate.R SnowParam-class.R MulticoreParam-class.R  
register.R SerialParam-class.R DoparParam-class.R SnowParam-utils.R  
BatchJobsParam-class.R utilities.R unix/mclapply.R unix/pvec.R  
unix/bpiterate.R unix/zzz.R

**Collate.windows** AllGenerics.R BiocParallelParam-class.R ErrorHandling.R  
bpbackend-methods.R bpsup-methods.R bplapply-methods.R bpmapply-methods.R  
bpiterate-methods.R bpschedule-methods.R bpstart-methods.R bpstop-methods.R  
bpvec-methods.R bpvectorize-methods.R bpworkers-methods.R  
bpaggregate-methods.R bpvalidate.R SnowParam-class.R MulticoreParam-class.R  
register.R SerialParam-class.R DoparParam-class.R SnowParam-utils.R  
BatchJobsParam-class.R utilities.R windows/zzz.R

**VignetteBuilder** knitr

## R topics documented:

BiocParallel-package . . . . . 2

BatchJobsParam-class . . . . .	3
BiocParallelParam-class . . . . .	4
bpaggregate . . . . .	7
bpiterate . . . . .	8
bpapply . . . . .	11
bpmapply . . . . .	13
bpresume . . . . .	15
bpschedule . . . . .	17
bpvalidate . . . . .	18
bpvec . . . . .	20
bpvectorize . . . . .	22
DoparParam-class . . . . .	23
MulticoreParam-class . . . . .	24
register . . . . .	28
SerialParam-class . . . . .	30
SnowParam-class . . . . .	31
<b>Index</b>	<b>37</b>

---

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)

Author: Bioconductor Package Maintainer [cre], Martin Morgan [aut], Michel Lang [aut], Ryan Thompson [aut]

Maintainer: Bioconductor Package Maintainer <maintainer@bioconductor.org>

---

BatchJobsParam-class    *Enable parallelization on batch systems*


---

## Description

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

## Usage

```
BatchJobsParam(workers, catch.errors = TRUE, cleanup = TRUE,
  work.dir = getwd(), stop.on.error = FALSE, seed = NULL,
  resources = NULL, conffile = NULL, cluster.functions = NULL,
  progressbar = TRUE, ...)
```

## Arguments

workers	integer(1) Number of workers to divide tasks (e.g., elements in the first argument of <code>bplapply</code> ) between. On Multicore and SSH backends, this defaults to all available nodes. On managed (e.g., slurm, SGE) clusters workers defaults to NA, meaning that the number of workers equals the number of tasks. See argument <code>n.chunks</code> in <a href="#">chunk</a> and <a href="#">submitJobs</a> for more information.
catch.errors	logical(1) Flag to determine in apply-like functions (see e.g. <a href="#">bplapply</a> ) whether to quit with an error as soon as one application fails or encapsulation of function calls in <a href="#">try</a> blocks which triggers a resume mechanism (see <a href="#">bpresume</a> ). Defaults to TRUE.
cleanup	logical(1) BatchJobs creates temporary directories in the <code>work.dir</code> . If <code>cleanup</code> is set to TRUE (default), the directories are removed from the file systems automatically. Set this to FALSE whenever it might become necessary to utilize any special functionality provided by BatchJobs. To retrieve the registry, call <a href="#">loadRegistry</a> on the temporary directory.
work.dir	character(1) Directory to store temporary files. Note that this must be shared across computational nodes if you use a distributed computing backend. Default is the current working directory of R, see <a href="#">getwd</a> .
stop.on.error	logical(1) Stop all jobs as soon as one jobs fails ( <code>stop.on.error == TRUE</code> ) or wait for all jobs to terminate. Default is FALSE.
seed	integer(1L) Set an initial seed for the RNG. See <a href="#">makeRegistry</a> for more information. Default is NULL where a random seed is chosen upon initialization.
resources	list() List of job specific resources passed to <a href="#">submitJobs</a> . Default is NULL where the resources defined in the configuration are used.
conffile	character(1) URI to a custom BatchJobs configuration file used for execution. Default is NULL which relies on BatchJobs to handle configuration files.
cluster.functions	ClusterFunctions Specify a specific cluster backend using one of the constructors provided by BatchJobs, see <a href="#">ClusterFunctions</a> . Default is NULL where the default cluster functions defined in the configuration are used.

progressbar	logical(1) Suppress the progress bar used in BatchJobs and be less verbose. Default is FALSE.
...	Addition arguments, currently not handled.

### BatchJobsParam 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](#), [bpstart](#), [bpstop](#), [bpisup](#), [bpbackend](#), [bpbackend<-](#)

### Author(s)

Michel Lang, <mailto:michellang@gmail.com>

### See Also

`getClass("BiocParallelParam")` for additional parameter classes.  
[register](#) for registering parameter classes for use in parallel evaluation.

### Examples

```
p <- BatchJobsParam(progressbar=FALSE)
bplapply(1:10, sqrt, BPPARAM=p)

## Not run:
## see vignette for additional explanation
funs <- makeClusterFunctionsSLURM("~/slurm.tpl")
param <- BatchJobsParam(4, cluster.functions=funs)
register(param)
bplapply(1:10, function(i) sqrt)

## End(Not run)
```

---

BiocParallelParam-class

*BiocParallelParam objects*

---

### Description

The `BiocParallelParam` virtual class stores configuration parameters for parallel execution. Concrete subclasses include `SnowParam`, `MulticoreParam`, `BatchJobsParam`, 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  
 BatchJobsParam: 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()`
- `BatchJobsParam()`
- `DoparParam()`
- `SerialParam()`

## Accessors

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

`bpworkers(x, ...)`: `integer(1)` or `character()`. Gets the number or names of the back-end workers.

`bptasks(x, ...)`, `bptasks(x) <- value`: `integer(1)`. Get or set the number of tasks for a job. `value` must be a scalar integer  $\geq 0$ L. This argument applies to `SnowParam` and `MulticoreParam` only; `DoparParam` and `BatchJobsParam` have their own approach to dividing a job among workers.

In this documentation a job is defined as a single call to a function, `bpapply`, `bpmap` 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).

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.

`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 `bpapply` 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 `showMethods("backend")`.

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

`bpcatchErrors(x, ...)`, `bpcatchErrors(x) <- value`: Get or set the flag to determine if apply-like functions quit with an error as soon as one application fails. See `?SnowParam` and `?BatchJobsParam` for details applicable to each class.

`bpstopOnError(x, ...)`, `bpstopOnError(x) <- value`: Get or set the flag to determine if all jobs should be stopped as soon as one fails or if all should be attempted.

## 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, BPRESUME=getOpt
bplapply(X, FUN, ..., BPRESUME = getOption("BiocParallel.BPRESUME", FALSE),
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
- [BatchJobsParam](#) for computing with cluster schedulers
- [DoparParam](#) for computing with foreach
- [SerialParam](#) for non-parallel execution

## Examples

```
getClass("BiocParallelParam")

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

---

bpaggregate	<i>Apply a function on subsets of data frames</i>
-------------	---

---

## Description

This is a parallel version of [aggregate](#).

## Usage

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

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

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

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

## 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 <a href="#">simplify2array</a> .
BPPARAM	An optional <a href="#">BiocParallelParam</a> instance determining the parallel back-end to be used during evaluation.

## 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 (all(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.

**Usage**

```
bpiterate(ITER, FUN, ..., BPPARAM=bpparam())

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

## S4 method for signature 'ANY,ANY,BiocParallelParam'
bpiterate(ITER, FUN, ..., BPPARAM=bpparam())
```

**Arguments**

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 <a href="#">BiocParallelParam</a> instance determining the parallel back-end to be used during evaluation, or a list of <a href="#">BiocParallelParam</a> instances, to be applied in sequence for nested calls to bpiterate. Currently only MulticoreParam is supported for bpiterate.
...	Arguments to other methods, specifically named arguments for FUN, or REDUCE or init. <ul style="list-style-type: none"> <li>• 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. Currently not supported on Windows.</li> <li>• init: Optional initial value for REDUCE; must be of the same type as the object returned from FUN. When supplied, <code>reduce.in.order</code> is set to TRUE.</li> <li>• <code>reduce.in.order</code>: Logical. When TRUE, REDUCE is applied to the results from the workers in the same order the tasks were sent out.</li> </ul>

## Details

Currently only MulticoreParam and SerialParam are supported.

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 codebplapply 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 regardless 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.

## Value

A list of output type specified by FUN. When REDUCE is supplied the list is of length 1.

## Author(s)

Valerie Obenchain <mailto:vobencha@fhcrc.org>.

The multi-core implementation is a modification of the sclapply in HTSeqGeni by Gregorie Pau.

## See Also

- [bpvec](#) for parallel, vectorized calculations.
- [bplapply](#) for parallel, lapply-like calculations.
- [BiocParallelParam](#) for details of BPPARAM.

## Examples

```

if (all(require(Rsamtools) &&
         require(RNAseqData.HNRNPC.bam.chr14) &&
         require(GenomicAlignments) &&
         require(ShortRead) &&
         .Platform$OS.type != "windows")) {

  ## -----
  ## Iterating 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
    }
  }

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

  ## Create a FUN that 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)
  }

  ## Create a MulticoreParam and call bpiterate().
  bpparam <- MulticoreParam(workers = 2)
  res <- bpiterate(ITER, counter, BPPARAM = bpparam, roi = roi)

  ## The result length is the same as the number of data chunks.
  length(res)
  colSums(do.call(rbind, res))
}

```

```

## -----
## Iterating 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")
fqs <- FastqStreamer(fl, n = 100)

## 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
  }
}

## Initialize the iterator.
ITER <- fastqIterator(fqs)

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

bpparam <- MulticoreParam(workers = 2)
bpiterate(ITER, summary, BPPARAM = bpparam)

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

}

```

## 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, ...,
         BPRESUME = getOption("BiocParallel.BPRESUME", FALSE),
         BPPARAM=bpparam())

## S4 method for signature 'ANY,ANY'
bplapply(X, FUN, ...,
         BPRESUME = getOption("BiocParallel.BPRESUME", FALSE),
         BPPARAM=bpparam())

## S4 method for signature 'ANY,missing'
bplapply(X, FUN, ...,
         BPRESUME = getOption("BiocParallel.BPRESUME", FALSE),
         BPPARAM=bpparam())

## S4 method for signature 'ANY,BiocParallelParam'
bplapply(X, FUN, ...,
         BPRESUME = getOption("BiocParallel.BPRESUME", FALSE),
         BPPARAM=bpparam())
```

## 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 <a href="#">lapply</a> .
BPRESUME	Flag to determine if a previous partially successful run should be resumed. See <a href="#">bpresume</a> for details. Supported for BatchJobsParam and DoparParam only.
BPPARAM	An optional <a href="#">BiocParallelParam</a> instance determining the parallel back-end to be used during evaluation, or a list of <a href="#">BiocParallelParam</a> instances, to be applied in sequence for nested calls to bplapply.

## Details

See `showMethods{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**

```
showMethods("bplapply")

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

---

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,
  BPRESUME=getOption("BiocParallel.BPRESUME", FALSE), BPPARAM=bpparam())

## S4 method for signature 'ANY,ANY'
bpmapply(FUN, ..., MoreArgs=NULL, SIMPLIFY=TRUE, USE.NAMES=TRUE,
  BPRESUME=getOption("BiocParallel.BPRESUME", FALSE), BPPARAM=bpparam())

## S4 method for signature 'ANY,missing'
bpmapply(FUN, ..., MoreArgs=NULL, SIMPLIFY=TRUE, USE.NAMES=TRUE,
  BPRESUME=getOption("BiocParallel.BPRESUME", FALSE), BPPARAM=bpparam())

## S4 method for signature 'ANY,BiocParallelParam'
bpmapply(FUN, ..., MoreArgs=NULL, SIMPLIFY=TRUE, USE.NAMES=TRUE,
  BPRESUME=getOption("BiocParallel.BPRESUME", FALSE), BPPARAM=bpparam())
```

## Arguments

<code>FUN</code>	The function to be applied to each element passed via <code>...</code> .
<code>...</code>	Objects for which methods <code>length</code> , <code>[</code> , and <code>[[</code> are implemented. All objects must have the same length or shorter objects will be replicated to have length equal to the longest.
<code>MoreArgs</code>	List of additional arguments to <code>FUN</code> .
<code>SIMPLIFY</code>	If <code>TRUE</code> the result will be simplified using <a href="#">simplify2array</a> .
<code>USE.NAMES</code>	If <code>TRUE</code> the result will be named.
<code>BPRESUME</code>	Flag to determine if a previous partially successful run should be resumed. See <a href="#">bpresume</a> for details. Supported for <code>BatchJobsParam</code> and <code>DoparParam</code> only.
<code>BPPARAM</code>	An optional <a href="#">BiocParallelParam</a> instance defining the parallel back-end to be used during evaluation.

## Details

See `showMethods{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

```
showMethods("bpmapply")

## ten tasks (1:10) so ten calls to FUN default registered parallel
## back-end. Compare with bpvec.
result <- bpmapply(function(greet, who) {
  paste(Sys.getpid(), greet, who)
}, c("morning", "night"), c("sun", "moon"))
cat(paste(result, collapse="\n"), "\n")
```

bpresume

*Resume computation with partial results***Description**

Resume partial successful calls to [bplapply](#) or [bpmapply](#)

**Usage**

```
bplastererror()
bpresume(expr)
```

**Arguments**

expr                    expression A expression which calls either [bplapply](#) or [bpmapply](#).

**Details**

- **bpresume** Implemented for [bplapply](#) and [bpmapply](#) only. Supported for `BatchJobsParam` and `DoparParam` only.

The resume mechanism is triggered if the argument `catch.errors` of the [BiocParallelParam](#) class is set to `TRUE`. The [bplapply](#) and [bpmapply](#) methods store the current state of computation marking which jobs were successful and which returned an error.

There are two approaches to re-computing jobs that returned errors. The first is to set `BPRESUME=TRUE` in the call to [bplapply](#) or [bpmapply](#). Alternatively, if the call to [bplapply](#) and [bpmapply](#) is inside a function and not directly accessible to the user, the last call can be embedded in `bpresume()`. Wrapping in `bpresume` sets an option accordingly to enable the resume feature down in the call stack. In both cases, only the computations with errors are re-attempted. New results are merged with the previous and returned to the user.

- **bplastererror** Returns a `LastError` object containing the partial results and errors to investigate.

Note that nested calls of the apply functions can cause troubles in some scenarios.

**Author(s)**

Martin Morgan <mailto:mtmorgan@fhcrc.org>

**Examples**

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

## By default 'catch.errors' is TRUE in BiocParallelParam objects.
SnowParam(workers = 2)

## If 'catch.errors' is FALSE an ill-fated bplapply() simply stops
```

```

## displaying the error message.
snowp <- SnowParam(workers = 2, catch.errors = FALSE)
## Not run:
> res <- bplapply(list(1, "two", 3), sqrt, BPPARAM = snowp)
Error in checkForRemoteErrors(val) :
  one node produced an error: non-numeric argument to mathematical function

## End(Not run)

## When 'catch.errors' is TRUE the same call provides traceback
## information (truncated here) and suggests use of bplastererror() and
## bpresume().
snowp <- SnowParam(workers = 2)
## Not run:
> res <- bplapply(list(1, "two", 3), sqrt, BPPARAM = snowp)
Error: 1 errors; first error:
  Error in FUN(...): non-numeric argument to mathematical function

For more information, use bplastererror(). To resume calculation, re-call
the function and set the argument 'BPRESUME' to TRUE or wrap the
previous call in bpresume().

First traceback:
19: parallel:::slaveRSOCK()
18: slaveLoop(makeSOCKmaster(master, port, timeout, useXDR))
...

## End(Not run)

## bplastererror() reports the number of successful results and error message.
## Not run:
> bplastererror()
2 / 3 partial results stored. First 1 error messages:
[2]: Error in FUN(...): non-numeric argument to mathematical function

## End(Not run)

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

## The 'resume' mechanism attempts to re-run the data element that failed.
## In our example the character "two" list element will never succeed. In
## the runs below we replace the character with a numeric and compute the
## result.

## There are two approaches for resuming a suspended calculation, one
## is to wrap the original call in bpresume().
## Not run:
> bpresume(res <- bplapply(list(1, 2, 3), sqrt, BPPARAM = snowp))
Resuming previous calculation...
> res
[[1]]

```



```
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

## End(Not run)

## Another approach is to set 'BPRESUME = TRUE' in bplapply().
## Not run:
res <- bplapply(list(1, 2, 3), sqrt, BPPARAM = snowp, BPRESUME = TRUE)

## End(Not run)
```

---

**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., <a href="#">MulticoreParam</a> , <a href="#">SnowParam</a> , <a href="#">DoparParam</a> .
	x can be missing, in which case the default back-end (see <a href="#">register</a> ) is used.
...	Additional arguments, perhaps used by methods.

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

---

bpvalidate	<i>Tools for developing functions for parallel execution in distributed memory</i>
------------	--

---

## Description

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

## Usage

```
bpvalidate(fun)
```

## Arguments

fun                      The function to be checked.

## 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, BatchJobs, 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**

A list of length 2 with named elements ‘inPath’ and ‘unknown’.

- inPath A named list of symbols and where they were found. These symbols were found on the search path instead of the function environment and should probably be imported in the NAMESPACE or otherwise defined in the package.
- unknown A vector of symbols not found in the function environment or the search path.

**Author(s)**

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

**Examples**

```
## -----
## 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(f1, ...)
  countBam(f1)
bpvalidate(fun1)

## countBam() is not defined in .GlobalEnv and must be passed as
## an argument or made available by loading the library.
fun2 <- function(f1, ...) {
  library(Rsamtools)
  countBam(f1)
```

```

}
bpvalidate(fun2)

## Passing arguments:
## 'param' is defined in the workspace but not passed to 'fun3'.
## bpvalidate() flags 'param' as being found 'inPath' 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, ...) {
  library(Rsamtools)
  countBam(fl, param=param)
}
bpvalidate(fun3)

## 'param' is explicitly passed by adding it as a formal argument.
fun4 <- function(fl, ..., param) {
  library(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))

```

---

bpvec

*Parallel, vectorized evaluation*


---

## Description

bpvec applies FUN to subsets of X. Any type of object X is allowed, provided length, [, and c methods are available. The return value is a vector of length equal to X, as with FUN(X).

## Usage

```

bpvec(X, FUN, ..., AGGREGATE=c, BPPARAM=bpparam())

## S4 method for signature 'ANY,ANY'
bpvec(X, FUN, ..., AGGREGATE=c, BPPARAM=bpparam())

## S4 method for signature 'ANY,BiocParallelParam'
bpvec(X, FUN, ..., AGGREGATE=c, BPPARAM=bpparam())

## S4 method for signature 'ANY,missing'
bpvec(X, FUN, ..., AGGREGATE=c, BPPARAM=bpparam())

```

## Arguments

X	Any object for which methods <code>length</code> , <code>[</code> , and <code>c</code> are implemented.
FUN	The function to be applied to subsets of X.
...	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, <code>c</code> , concatenates objects and is appropriate for vectors; <code>rbind</code> might be appropriate for data frames.
BPPARAM	A optional <a href="#">BiocParallelParam</a> instance determining the parallel back-end to be used during evaluation.

## Details

When BPPARAM is a `MulticoreParam` this method dispatches to the `pvec` function from the `parallel` package.

For all other `BiocParallelParams`, this method creates a vector of indices for X that divide the elements as evenly as possible given the number of workers. Indices and data are passed to `bplapply` for parallel evaluation. `SnowParam` and `MulticoreParam` offer further control over the division of X through the `tasks` argument. See `?bptasks`.

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).

## Author(s)

Martin Morgan <mailto:mtmorgan@fhcrc.org>. Original code as attributed in [pvec](#).

## See Also

[bplapply](#) for parallel lapply.

[BiocParallelParam](#) for possible values of BPPARAM.

[pvec](#) for background.

## Examples

```
showMethods("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

```

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, ..., BPPARAM=bpparam())

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

## S4 method for signature 'ANY,missing'
bpvectorize(FUN, ..., BPPARAM=bpparam())

```

**Arguments**

FUN	A function whose first argument has a length and can be subset <code>[]</code> , 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 <code>c</code> .
...	Additional arguments to parallization, unused.
BPPARAM	An optional <a href="#">BiocParallelParam</a> instance determining the parallel back-end to be used during evaluation.

**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 `showMethods{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)
```

---

DoparParam-class	<i>Enable parallel evaluation using registered dopar backend</i>
------------------	--

---

**Description**

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

**Usage**

```
DoparParam(catch.errors = TRUE)
```

**Arguments**

`catch.errors` `logical(1)` Flag to determine in apply-like functions (see e.g. [bplapply](#)) whether to quit with an error as soon as one application fails or encapsulation of function calls in [try](#) blocks which triggers a resume mechanism (see [bpresume](#)). Defaults to TRUE.

**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](#), [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**

```
# First register a parallel backend with foreach
library(doParallel)
cl <- makeCluster(2)
registerDoParallel(cl)

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

stopCluster(cl)
## Not run:
register(DoparParam(), default=TRUE)

## End(Not run)
```

---

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 based on operating system (Windows only supports 1 core), global user preference (options(mc.cores=...)), or the minimum of 8 and the number of detected cores (detectCores()).

**Usage**

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

MulticoreParam(workers = multicoreWorkers(), tasks = 0L,
               catch.errors = TRUE, stop.on.error = FALSE,
               log = FALSE, threshold = "INFO", logdir = character(),
               resultdir = character(), setSeed = TRUE,
               recursive = TRUE, cleanup = TRUE, cleanupSignal = tools::SIGTERM,
               verbose = FALSE, ...)

## detect workers
```



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

```
multicoreWorkers()
```

## Arguments

workers	integer(1) Number of workers. Defaults to all cores available as determined by detectCores.
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, 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).  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.
catch.errors	logical(1) Enable the catching of errors and warnings.
stop.on.error	logical(1) Enable stop on error.
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.
setSeed	logical(1) as described in parallel::mcpipeline argument mc.set.seed.
recursive	logical(1) indicating whether recursive calls are evaluated in parallel; see parallel::mclapply argument mc.allow.recursive.
cleanup	logical(1) indicating whether forked children will be terminated before bplapply returns, as for parallel::mclapply argument cleanup. If TRUE, then the signal sent to the child is cleanupSignal.
cleanupSignal	integer(1) the signal sent to forked processes when cleanup=TRUE.
verbose	logical(1) when TRUE echo stdout of forked processes. This is the complement of parallel::mclapply's argument mc.silent.
...	Additional arguments passed to <a href="#">makeCluster</a>

## Details

MulticoreParam is used for shared memory computing. Under the hood the cluster is created with SnowParam(..., type="FORK"). See ?SnowParam for a description of error handling, logging and writing out results.

**Constructor**

```
MulticoreParam(workers = multicoreWorkers(), tasks = 0L, catch.errors = TRUE, s
```

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.

```
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, ...)
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.

```
bpcatchErrors(x, ...), bpcatchErrors(x) <- value
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.

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.

**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
- [BatchJobsParam](#) for computing with cluster schedulers
- [DoparParam](#) for computing with foreach
- [SerialParam](#) for non-parallel evaluation

**Examples**

```
multicoreWorkers()
p <- MulticoreParam()
bplapply(1:10, sqrt, BPPARAM=p)
bpvec(1:10, sqrt, BPPARAM=p)

## Not run:
register(MulticoreParam(), default=TRUE)

## End(Not run)
```

---

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., <a href="#">MulticoreParam</a> , <a href="#">SnowParam</a> , <a href="#">DoparParam</a> .
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. Defaults are SnowParam and SerialParam on Windows and MulticoreParam, SnowParam and SerialParam on non-Windows. Additional backends can be added to the registry and existing entries can be modified.

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.
registered()

## 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")

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

## The back-end of choice is specified in the BPPARAM argument to
## the BiocParallel functions. None, one, or multiple back-ends can be
## provided.

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

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

---

SerialParam-class	<i>Enable serial evaluation</i>
-------------------	---------------------------------

---

## Description

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

## Usage

```
SerialParam()
```

## SerialParam constructor

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)

## Not run:
register(SerialParam(), default=TRUE)

## End(Not run)
```

---

SnowParam-class	<i>Enable simple network of workstations (SNOW)-style parallel evaluation</i>
-----------------	---

---

## 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 based on global user preference (`options(mc.cores=...)`), or the minimum of 8 and the number of detected cores (`detectCores()`).

## Usage

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

SnowParam(workers = snowWorkers(), type=c("SOCK", "MPI", "FORK"),
           tasks = 0L, catch.errors=TRUE, stop.on.error = FALSE,
           log = FALSE, threshold = "INFO", logdir = character(),
           resultdir = character(), ...)

## coercion
## -----

## S4 method for signature 'SOCKcluster,SnowParam'
coerce(from, to)
## S4 method for signature 'spawnedMPIcluster,SnowParam'
coerce(from, to)

## detect workers
## -----

snowWorkers()
```

## Arguments

<code>workers</code>	integer(1) Number of workers. Defaults to all cores available as determined by <code>detectCores</code> . For a SOCK cluster workers can be a <code>character()</code> vector of host names.
<code>type</code>	character(1) Type of cluster to use. Possible values are SOCK (default) and MPI. Instead of <code>type=FORK</code> use <code>MulticoreParam</code> .
<code>tasks</code>	integer(1). The number of tasks per job. value must be a scalar integer $\geq 0L$ .  In this documentation a job is defined as a single call to a function, <code>bplapply</code> , <code>bpmapply</code> 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).
	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.
catch.errors	logical(1) Enable the catching of errors and warnings.
stop.on.error	logical(1) Enable stop on error.
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.
...	Additional arguments passed to <a href="#">makeCluster</a>
from	A SOCKcluster or spawnedMPIcluster instance created with makeCluster in the parallel or snow package. Applicable to coerce methods only.
to	character(1) "SnowParam". Applicable to coerce methods only.

## 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.

**error handling:** Two flags control error handling: catch.errors and stop.on.error.

The catch.errors flag determines whether apply-like functions quit with an error as soon as one task fails. When FALSE, this is the default behavior seen in the parallel and snow packages; an error is returned with no results. When TRUE, errors are returned for tasks that failed as well as results for those that completed successfully.

stop.on.error offers an additional level of control and is only applicable when catch.errors == TRUE.

When this argument is TRUE the job terminates upon the first error is encountered but the results also contain the successfully completed tasks. catch.errors must be TRUE when stop.on.error is TRUE. When FALSE, the job runs to completion and all successfully completed tasks are returned along with any error messages.

**logging:** Logging is implemented on the workers with the futile.logger package. When bplg(BPPARAM) == TRUE cluster workers use a script in the BiocParallel package instead of snow. This modified script captures warning and error messages and collects additional task statistics such as gc output and node name.

Any futile.logger messages the user provides in their function will be captured by the logging mechanism. Messages are returned real-time (i.e., as each task completes) instead of after all jobs have finished.

**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. These options are controlled with logdir and resultdir.



NOTE: The PSOCK cluster from the parallel package does not support cluster options scriptdir and useRscript. Because these options are needed to use an alternate worker script, PSOCK is not supported in SnowParam.

### Constructor

```
SnowParam(workers = snowWorkers(), type=c("SOCK", "MPI"), catch.errors = TRUE, stop)
Return an object representing a SNOW cluster. The cluster is not created until bptest is called. Named arguments in ... are passed to makeCluster.
```

### Accessors: Logging and results

In the following code, x is a SnowParam object.

```
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 SnowParam object. See the ?BiocParallelParam man page for details on these accessors.

```
bpworkers(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.

```
bpcatchErrors(x, ...), bpcatchErrors(x) <- value
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.

**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
- [BatchJobsParam](#) for computing with cluster schedulers
- [DoparParam](#) for computing with foreach
- [SerialParam](#) for non-parallel evaluation

**Examples**

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

## SnowParam supports distributed memory computing. Fields control the
## division of tasks, error handling, logging and how results are returned.
bpparam <- SnowParam()
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)
    if (TRUE & c(TRUE, TRUE)) ## warning
      x
  } else if (x == 2) {
    x ## ok
  } else if (x == 3) {
    sqrt("FOO") ## error
  }
}

## By default logging is off; turn it on with bplog():
bpparam <- SnowParam(3)
bplog(bpparam) <- TRUE
bplapply(X, fun, BPPARAM=bpparam)

## When a 'logdir' location is given the messages are redirected to a file:
bplogdir(bpparam) <- tempdir()
bplapply(X, fun, BPPARAM=bpparam)
list.files(bplogdir(bpparam))

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

## Results are written to a file by providing a directory path in
## 'resultdir'.
bpparam <- SnowParam(2, type="SOCK", resultdir = tempdir())
bplapply(X, fun, BPPARAM=bpparam)

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

## When 'stop.on.error' is TRUE, the method returns as soon as an error
## is thrown.
bpparam <- SnowParam(2, type="SOCK", stop.on.error=TRUE)
res <- bplapply(list(1, "two", 3, 4), sqrt, BPPARAM=bpparam)
res

```

```
## bpok() returns TRUE for the elements that did not return an error.
bpok(res)

## When 'stop.on.error' is FALSE, all results are computed.
bpstopOnError(bpparam) <- FALSE
res <- bplapply(list(1, "two", 3, 4), sqrt, BPPARAM=bpparam)
bpok(res)
```

# Index

## \*Topic **classes**

- BiocParallelParam-class, 4
- DoparParam-class, 23
- MulticoreParam-class, 24
- SerialParam-class, 30
- SnowParam-class, 31

## \*Topic **interface**

- bpvectorize, 22

## \*Topic **manip**

- bpiterate, 8
- bplapply, 11
- bpmapply, 13
- bpschedule, 17
- bpvalidate, 18
- bpvec, 20
- register, 28

## \*Topic **methods**

- BiocParallelParam-class, 4
- bpiterate, 8
- MulticoreParam-class, 24
- SnowParam-class, 31

## \*Topic **package**

- BiocParallel-package, 2

aggregate, 7

BatchJobsParam, 6, 27, 34

BatchJobsParam (BatchJobsParam-class), 3

BatchJobsParam-class, 3

BiocParallel (BiocParallel-package), 2

BiocParallel-package, 2

BiocParallelParam, 7, 9, 12–15, 17, 21, 22, 28, 29

BiocParallelParam

(BiocParallelParam-class), 4

BiocParallelParam-class, 4

bpaggregate, 7

bpaggregate, ANY, missing-method  
(bpaggregate), 7

bpaggregate, data.frame, BiocParallelParam-method  
(bpaggregate), 7

bpaggregate, formula, BiocParallelParam-method  
(bpaggregate), 7

bpaggregate, matrix, BiocParallelParam-method  
(bpaggregate), 7

bpbackend, 4, 23

bpbackend (BiocParallelParam-class), 4

bpbackend, BatchJobsParam-method  
(BatchJobsParam-class), 3

bpbackend, DoparParam-method  
(DoparParam-class), 23

bpbackend, missing-method  
(BiocParallelParam-class), 4

bpbackend, SnowParam-method  
(SnowParam-class), 31

bpbackend<- (BiocParallelParam-class), 4

bpbackend<-, BatchJobsParam  
(BatchJobsParam-class), 3

bpbackend<-, DoparParam, SOCKcluster-method  
(DoparParam-class), 23

bpbackend<-, missing, ANY-method  
(BiocParallelParam-class), 4

bpbackend<-, SnowParam, cluster-method  
(SnowParam-class), 31

bpcatchErrors

(BiocParallelParam-class), 4

bpcatchErrors, BiocParallelParam-method  
(BiocParallelParam-class), 4

bpcatchErrors<-  
(BiocParallelParam-class), 4

bpcatchErrors<-, BiocParallelParam, logical-method  
(BiocParallelParam-class), 4

bpisup, 4, 23, 30

bpisup (BiocParallelParam-class), 4

bpisup, ANY-method  
(BiocParallelParam-class), 4

bpisup, BatchJobsParam-method  
(BatchJobsParam-class), 3

bpisup, DoparParam-method  
     (DoparParam-class), 23  
 bpisup, missing-method  
     (BiocParallelParam-class), 4  
 bpisup, MulticoreParam-method  
     (MulticoreParam-class), 24  
 bpisup, SerialParam-method  
     (SerialParam-class), 30  
 bpisup, SnowParam-method  
     (SnowParam-class), 31  
 bpiterate, 8  
 bpiterate, ANY, ANY, BatchJobsParam-method  
     (bpiterate), 8  
 bpiterate, ANY, ANY, BiocParallelParam-method  
     (bpiterate), 8  
 bpiterate, ANY, ANY, DoparParam-method  
     (bpiterate), 8  
 bpiterate, ANY, ANY, missing-method  
     (bpiterate), 8  
 bpiterate, ANY, ANY, MulticoreParam-method  
     (bpiterate), 8  
 bpiterate, ANY, ANY, SerialParam-method  
     (bpiterate), 8  
 bpiterate, ANY, ANY, SnowParam-method  
     (bpiterate), 8  
 bplapply, 3, 9, 11, 15, 21, 23, 30  
 bplapply, ANY, ANY-method (bplapply), 11  
 bplapply, ANY, BatchJobsParam-method  
     (bplapply), 11  
 bplapply, ANY, BiocParallelParam-method  
     (bplapply), 11  
 bplapply, ANY, DoparParam-method  
     (bplapply), 11  
 bplapply, ANY, list-method (bplapply), 11  
 bplapply, ANY, missing-method (bplapply),  
     11  
 bplapply, ANY, MulticoreParam-method  
     (bplapply), 11  
 bplapply, ANY, SerialParam-method  
     (bplapply), 11  
 bplapply, ANY, SnowParam-method  
     (bplapply), 11  
 bplastererror (bpresume), 15  
 bplog (SnowParam-class), 31  
 bplog, SnowParam-method  
     (SnowParam-class), 31  
 bplog<- (SnowParam-class), 31  
 bplog<-, SnowParam, logical-method  
     (SnowParam-class), 31  
 bplogdir (SnowParam-class), 31  
 bplogdir, SnowParam-method  
     (SnowParam-class), 31  
 bplogdir<- (SnowParam-class), 31  
 bplogdir<-, SnowParam, character-method  
     (SnowParam-class), 31  
 bpmapply, 13, 15  
 bpmapply, ANY, ANY-method (bpmapply), 13  
 bpmapply, ANY, BatchJobsParam-method  
     (bpmapply), 13  
 bpmapply, ANY, BiocParallelParam-method  
     (bpmapply), 13  
 bpmapply, ANY, DoparParam-method  
     (bpmapply), 13  
 bpmapply, ANY, missing-method (bpmapply),  
     13  
 bpmapply, ANY, MulticoreParam-method  
     (bpmapply), 13  
 bpmapply, ANY, SerialParam-method  
     (bpmapply), 13  
 bpmapply, ANY, SnowParam-method  
     (bpmapply), 13  
 bpok (SnowParam-class), 31  
 bpparam (register), 28  
 bpresultdir (SnowParam-class), 31  
 bpresultdir, SnowParam-method  
     (SnowParam-class), 31  
 bpresultdir<- (SnowParam-class), 31  
 bpresultdir<-, SnowParam, character-method  
     (SnowParam-class), 31  
 bpresume, 3, 12, 14, 15, 23  
 bprunMPIslave (SnowParam-class), 31  
 bpschedule, 17  
 bpschedule, ANY-method (bpschedule), 17  
 bpschedule, BatchJobsParam-method  
     (BatchJobsParam-class), 3  
 bpschedule, missing-method (bpschedule),  
     17  
 bpschedule, MulticoreParam-method  
     (MulticoreParam-class), 24  
 bpslaveLoop (SnowParam-class), 31  
 bpstart, 4, 23, 30  
 bpstart (BiocParallelParam-class), 4  
 bpstart, ANY-method  
     (BiocParallelParam-class), 4  
 bpstart, BatchJobsParam-method  
     (BatchJobsParam-class), 3

- bpstart,DoparParam-method  
(DoparParam-class), 23
- bpstart,missing-method  
(BiocParallelParam-class), 4
- bpstart,SnowParam-method  
(SnowParam-class), 31
- bpstop, 4, 23, 30
- bpstop (BiocParallelParam-class), 4
- bpstop,ANY-method  
(BiocParallelParam-class), 4
- bpstop,BatchJobsParam-method  
(BatchJobsParam-class), 3
- bpstop,DoparParam-method  
(DoparParam-class), 23
- bpstop,missing-method  
(BiocParallelParam-class), 4
- bpstop,SnowParam-method  
(SnowParam-class), 31
- bpstopOnError  
(BiocParallelParam-class), 4
- bpstopOnError,BiocParallelParam-method  
(BiocParallelParam-class), 4
- bpstopOnError<-  
(BiocParallelParam-class), 4
- bpstopOnError<-,BiocParallelParam,logical-method  
(BiocParallelParam-class), 4
- bptasks (BiocParallelParam-class), 4
- bptasks,BiocParallelParam-method  
(BiocParallelParam-class), 4
- bptasks<- (BiocParallelParam-class), 4
- bptasks<-,BiocParallelParam,numeric-method  
(BiocParallelParam-class), 4
- bpthreshold (SnowParam-class), 31
- bpthreshold,SnowParam-method  
(SnowParam-class), 31
- bpthreshold<- (SnowParam-class), 31
- bpthreshold<-,SnowParam,character-method  
(SnowParam-class), 31
- bpvalidate, 18
- bpvec, 9, 13, 14, 20, 22, 23, 30
- bpvec,ANY,ANY-method (bpvec), 20
- bpvec,ANY,BiocParallelParam-method  
(bpvec), 20
- bpvec,ANY,missing-method (bpvec), 20
- bpvec,ANY,MulticoreParam-method  
(MulticoreParam-class), 24
- bpvectorize, 22
- bpvectorize,ANY,ANY-method  
(bpvectorize), 22
- bpvectorize,ANY,missing-method  
(bpvectorize), 22
- bpworkers, 4, 23, 30
- bpworkers (BiocParallelParam-class), 4
- bpworkers,BatchJobsParam-method  
(BatchJobsParam-class), 3
- bpworkers,BiocParallelParam-method  
(BiocParallelParam-class), 4
- bpworkers,DoparParam-method  
(DoparParam-class), 23
- bpworkers,missing-method  
(BiocParallelParam-class), 4
- bpworkers,SerialParam-method  
(SerialParam-class), 30
- bpworkers,SnowParam-method  
(SnowParam-class), 31
- chunk, 3
- ClusterFunctions, 3
- coerce,SOCKcluster,DoparParam-method  
(DoparParam-class), 23
- coerce,SOCKcluster,SnowParam-method  
(SnowParam-class), 31
- coerce,spawnedMPIcluster,SnowParam-method  
(SnowParam-class), 31
- DoparParam, 6, 17, 27, 28, 34
- DoparParam (DoparParam-class), 23
- DoparParam-class, 23
- getwd, 3
- lapply, 12
- loadRegistry, 3
- makeCluster, 25, 32
- makeRegistry, 3
- mapply, 13, 14
- mclapply, 13, 14
- MulticoreParam, 6, 17, 28, 34
- MulticoreParam (MulticoreParam-class), 24
- MulticoreParam-class, 24
- multicoreWorkers  
(MulticoreParam-class), 24
- parallel, 2
- pvec, 21

register, [17](#), [28](#)  
registered (register), [28](#)  
  
SerialParam, [6](#), [22](#), [27](#), [34](#)  
SerialParam (SerialParam-class), [30](#)  
SerialParam-class, [30](#)  
show, BatchJobsParam-method  
    (BatchJobsParam-class), [3](#)  
show, BiocParallel-method  
    (BiocParallelParam-class), [4](#)  
show, DoparParam-method  
    (DoparParam-class), [23](#)  
show, MulticoreParam-method  
    (MulticoreParam-class), [24](#)  
show, SnowParam-method  
    (SnowParam-class), [31](#)  
simplify2array, [7](#), [14](#)  
SnowParam, [6](#), [17](#), [27](#), [28](#)  
SnowParam (SnowParam-class), [31](#)  
SnowParam-class, [31](#)  
snowWorkers (SnowParam-class), [31](#)  
submitJobs, [3](#)  
  
try, [3](#), [23](#)