Package 'BiocParallel'

October 25, 2015

Type Package

Title Bioconductor facilities for parallel evaluation

Version 1.4.0

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 bpisup-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 progress.R utilities.R unix/pvec.R unix/mclapply.R unix/zzz.R

Collate.windows AllGenerics.R BiocParallelParam-class.R ErrorHandling.R bpbackend-methods.R bpisup-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 progress.R utilities.R windows/zzz.R

VignetteBuilder knitr

NeedsCompilation no

Author Bioconductor Package Maintainer [cre], Martin Morgan [aut],

Valerie Obenchain [aut],
Michel Lang [aut],
Ryan Thompson [aut]

Maintainer Bioconductor Package Maintainer <maintainer@bioconductor.org>

R topics documented:

`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], Valerie Obenchain [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, jobname = "BPJOB", ...)
```

Arguments

workers	integer(1)Number of workers to divide tasks (e.g., elements in the first argument of bplapply) 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 n.chunks in chunk and submitJobs for more information.
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.
cleanup	logical(1)BatchJobs creates temporary directories in the work.dir. If cleanup 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 loadRegistry 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 ist the current working directory of R, see getwd.
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 FALSE.
seed	integer(1L)Set an initial seed for the RNG. See makeRegistry for more information. Default is NULL where a random seed is chosen upon initialization.
resources	list()List of job specific resources passed to submitJobs. 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	ClusterFunctionsSpecify a specific cluster backend using on of the constructors provided by BatchJobs, see ClusterFunctions. 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.

jobname	character(1)Job name that is prepended to the output log and result files. Default is "BPJOB".
...	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.tmpl")
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, ...), 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.

bptasks(x, ...), bptasks(x) <- value: integer(1). Get or set the number of tasks for a job. value must be a scalar integer >= 0L. This argument applies to SnowParam and MulticoreParam only; DoparParam and BatchJobsParam 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).

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 showMethods("backend").

`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.

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

`bpcatchErrors(x, ...)`, `bpCatchErrors(x) <- value`: `logical()`. Controls if errors are caught and returned with completed results.
`catch.errors` determines whether errors are caught and returned with other results. When `TRUE`, all computations are attempted and output contains both errors and successfully completed results. When `FALSE`, the job is terminated as soon as the first error is hit and only the error message is returned (no results); this is the default behavior of the parent packages, e.g., `parallel`, `snow`, `foreach`.

`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. Unlike `catch.errors == TRUE`, 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=bppa
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
- `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 *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())

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

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

## S4 method for signature 'ANY,missing'
bpaggregate(x, ..., BPREDO=list(), 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 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.

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 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.

... — Arguments to other methods, specifically named arguments for FUN, or REDUCE or init.

- 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.

Details

Supported for SnowParam and MulticorParam.

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.

Value

A list the same length as the number of chunks in ITER(). When REDUCE is used list length is 1.

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.

Examples

```
## Not run:
if (all(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)
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())

## S4 method for signature 'ANY,missing'
bplapply(X, FUN, ...,
    BPREDO = list(), BPPARAM=bpparam())

## S4 method for signature 'ANY,BiocParallelParam'
bplapply(X, FUN, ...,
    BPREDO = list(), 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 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 bplapply.
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.

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.
fun <- function(v) {
    message("working") ## 10 tasks
    sqrt(v)
}
bplapply(1:10, fun)
```

---

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())

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

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

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.

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")

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

param <- bpparam()
bpworkers(param) <- 2
## Not run:
result <- bpmapply(fun, greet, who, BPPARAM = param)
cat(paste(result, collapse="\n"), "\n")

## End(Not run)
```

---

bpok *Resume computation with partial results*

Description

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

bpresume and bplaterror have been deprecated.

Usage

```
bpok(x)

## Deprected:
bpresume(expr)
bplasterror()
```

Arguments

x	Results returned from a call to bp*lapply.
expr	A expression to be re-evaluated. If the original error was due to input error, X should be modified. If hardware limitations or failure caused the error this expression may be the same as the original.

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.
- bpresume THIS FUNCTION IS DEPRECATED. The resume mechanism allows computations with errors to be re-attempted and is triggered when the argument catch.errors is TRUE.

  Unsuccessful results returned from bp*lapply can be identified with bpok. Failure may have been due to faulty input or hardware error. Incomplete portions of the job can be reattempted with bpresume. New results are merged with the previous and returned to the user.

- bplasterror THIS FUNCTION IS DEPRECATED. Use `attr` on the output of `bp*apply` to see traceback. See examples.

Author(s)

Michel Lang, Martin Morgan and Valerie Obenchain

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

## End(Not run)

## When 'catch.errors' is TRUE partial results are returned with
## the error.
param <- SnowParam(workers = 2)
X <- list(1, "two", 3)
res <- bplapply(X, sqrt, BPPARAM = param)
res

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

## Access the traceback with attr():
tail(attr(res[[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.
param <- SnowParam(workers = 2)
X <- list(1, "two", 3)
res <- bplapply(X, sqrt, BPPARAM = param)
res

## Fix the input data by changing the character "2" to a numeric 2:
X_mod <- list(1, 2, 3)
```

```
## Repeat the original call to bplapply() with the partial results as 'BPREDO':
bplapply(X_mod, sqrt, BPPARAM = param , BPREDO = res)
```

---

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

`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)
```

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:vobencha@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(fl, ...)
    countBam(fl)
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, ...) {
    library(Rsamtools)
    countBam(fl)
}
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, BPREDO=list(), BPPARAM=bpparam())

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

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

Arguments

X	Any object for which methods length, [, and c 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, c, concatenates objects and is appropriate for vectors; rbind might be appropriate for data frames.
BPPARAM	A optional BiocParallelParam instance determining the parallel back-end to be used during evaluation.
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.

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, ..., BPREDO=list(), BPPARAM=bpparam())

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

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

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.

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 *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(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.

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, 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)

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,
               progressbar = FALSE, RNGseed = NULL, timeout = Inf,
               log = FALSE, threshold = "INFO", logdir = NA_character_,
               resultdir = NA_character_, jobname = "BPJOB", ...)

## 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, 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.

`catch.errors` DEPRECATED. `logical(1)` Enable the catching of errors and warnings.

`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. When not `NULL`, this value is passed to parallel::clusterSetRNGStream to generate random number streams on each worker.

`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".

`...` 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. If not specified, the default number of workers is determined by `multicoreWorkers()`, which is parallel::detectCores() - 2. Machines with 3 or less cores are assigned a single worker.

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: The `catch.errors` field has been deprecated.

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

- `catch.errors` (DEPRECATED) determines whether errors are caught and returned with other results. When `TRUE`, all computations are attempted and output contains both errors and successfully completed results. When `FALSE`, the job is terminated as soon as the first error is hit and only the error message is returned (no results); this is the default behavior of the parent packages, e.g., `parallel`, `snow`, `foreach`.

- 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 FALSE, all computations are attempted and and successful results are returned along with any error messages.
- 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. There are many ways to track memory use - this particular approach was taken because it is consistent with how the BatchJobs package reports memory on the workers.

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: MulticoreParam and SnowParam use the random number generation support from the parallel package. These params are snow-derived clusters so the arguments for multicore-derived functions such as mc.set.seed and mc.reset.stream do not apply.

Random number generation is controlled through the param argument, RNGseed which is passed to parallel::clusterSetRNGStream. clusterSetRNGStream uses the L'Ecuyer-CMRG random number generator and distributes streams to the members of a cluster. If RNGseed is not NULL it serves as the seed to the streams, otherwise the streams are set from the current seed of the master process after selecting the L'Ecuyer generator. See ?clusterSetRNGStream for more details.

Constructor

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

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.

bpprogress(x, ...), bpprogress(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).

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.

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

```
## ----------------------------------------------------------------------
## 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
bpresultdir(bpparam) <- "/myResults/"
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 the bplog()<- setter
## or by specifying 'log = TRUE' in the constructor.
```

```
bpparam <- MulticoreParam(3, log = TRUE)
bplapply(X, fun, BPPARAM=bpparam)

## 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:
bpparam <- MulticoreParam(2, resultdir = tempdir())
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 FALSE, all computations are attempted and partial
## results are retured along with errors. To more clearly demonstrate
## the difference in output, 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.)

## Results prior to and including the error:
bpparam <- MulticoreParam(2, tasks = 4, stop.on.error = TRUE)
bplapply(list(1, "two", 3, 4), sqrt, BPPARAM = bpparam)

## All results along with error:
bpparam <- MulticoreParam(2, tasks = 4, stop.on.error = FALSE)
res <- 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(bpworkers(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(Multicore`

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 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(catch.errors = FALSE, log = FALSE, threshold = "INFO")
```

Arguments

catch.errors	logical(1) Enable the catching of errors and warnings.
log	logical(1) Enable logging.
threshold	character(1) Logging threshold as defined in futile.logger.

Constructor

SerialParam(catch.errors = FALSE, log = FALSE, threshold = "INFO"):

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 *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 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,
          progressbar = FALSE, RNGseed = NULL, timeout = Inf,
          log = FALSE, threshold = "INFO", logdir = NA_character_,
          resultdir = NA_character_, jobname = "BPJOB", ...)

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

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

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

snowWorkers()
```

Arguments

`workers`	`integer(1)` Number of workers. Defaults to all cores available as determined by `detectCores`. 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.
catch.errors	DEPRECATED. logical(1) Enable the catching of errors and warnings.
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. When not NULL, this value is passed to parallel::clusterSetRNGStream to generate random number streams on each worker.
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".
...	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. When not specified, the default number of workers is determined by snowWorkers() which is parallel::detectCores() - 2. Machines with 3 or less cores are assigned a single worker.

error handling: The catch.errors field has been deprecated.

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

- catch.errors (DEPRECATED) determines whether errors are caught and returned with other results. When TRUE, all computations are attempted and output contains both errors and successfully completed results. When FALSE, the job is terminated as soon as the first error is hit and only the error message is returned (no results); this is the default behavior of the parent packages, e.g., parallel, snow, foreach.
- 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 FALSE, all computations are attempted and and successful results are returned along with any error messages.
- 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. There are many ways to track memory use - this particular approach was taken because it is consistent with how the BatchJobs package reports memory on the workers.

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: `MulticoreParam` and `SnowParam` use the random number generation support from the parallel package. These params are snow-derived clusters so the arguments for multicore-derived functions such as `mc.set.seed` and `mc.reset.stream` do not apply.

Random number generation is controlled through the param argument, `RNGseed` which is passed to parallel::clusterSetRNGStream. `clusterSetRNGStream` uses the L'Ecuyer-CMRG random number generator and distributes streams to the members of a cluster. If `RNGseed` is not NULL it serves as the seed to the streams, otherwise the streams are set from the current seed of the master process after selecting the L'Ecuyer generator. See ?`clusterSetRNGStream` for more details.

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, catch.erro`

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)`.

`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`

`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. 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
bpresultdir(bpparam) <- "/myResults/"
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 the bplog()<- setter
## or by specifying 'log = TRUE' in the constructor.
bpparam <- SnowParam(3, log = TRUE)
bplapply(X, fun, BPPARAM = bpparam)
```

```
## 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:
bpparam <- SnowParam(2, resultdir = tempdir())
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.
bpparam <- SnowParam(2, stop.on.error = TRUE)
bplapply(list(1, "two", 3, 4), sqrt, BPPARAM = bpparam)

## When 'stop.on.error' is FALSE all computations are attempted. Partial
## results are returned along with errors.
bpparam <- SnowParam(2, stop.on.error = FALSE)
res <- 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 <- SnowParam(3, RNGseed = 7739465)
bplapply(seq_len(bpworkers(bpparam)), function(i) rnorm(1), BPPARAM = bpparam)
```

Index