Package 'parallel'

September 14, 2016

Version 3.3.1

Priority base

Title Support for Parallel computation in R

Author R Core Team

Maintainer R Core Team <R-core@r-project.org>

Description Support for parallel computation, including by forking (taken from package multicore), by sockets (taken from package snow) and random-number generation.

License Part of R 3.3.1

Imports tools

Suggests methods

Enhances snow, nws, Rmpi

R topics documented:

parallel-package . . . 2
clusterApply . . . 2
detectCores . . . 5
makeCluster . . . 7
mcaffinity . . . 9
mcchildren . . . 10
mcfork . . . 12
mclapply . . . 14
mcparallel . . . 16
pvec . . . 19
RNGstreams . . . 21
splitIndices . . . 22

Index **24**

parallel-package *Support for Parallel Computation*

Description

並列計算に対するサポートで，乱数生成を含む.

Details

このパッケージは発展途中である：最初のバージョンは R 2.14.0 でリリースされた.

‘"L'Ecuyer-CMRG"’ RNG を用いた多重 RNG ストリームのサポートがある：nextRNGStream を見よ.

これはパッケージ **multicore** (幾つかの低レベル関数は改名され移出されない) と **snow** (ソケットクラスター専用であるが， **snow** で生成された MPI と NWS クラスターもサポートされている)に含まれるものに由来し，かなりの程度同値である．このパッケージは又 makeForkCluster を提供する.

移出関数の完全なリストを得るには library(help = "parallel") を使う.

Author(s)

Brian Ripley, Luke Tierney そして Simon Urbanek

保守管理者: R Core Team <R-core@r-project.org>

See Also

並列計算はワーカープロセスの始動を含む：パッケージ **tools** 中の関数 psnice と pskill はそうしたプロセスを管理する方法を提供する.

clusterApply *Apply Operations using Clusters*

Description

これらの関数はクラスターを使った計算を並列化する幾つかの方法を提供する.

Usage

```
clusterCall(cl = NULL, fun, ...)
clusterApply(cl = NULL, x, fun, ...)
clusterApplyLB(cl = NULL, x, fun, ...)
clusterEvalQ(cl = NULL, expr)
clusterExport(cl = NULL, varlist, envir = .GlobalEnv)
clusterMap(cl = NULL, fun, ..., MoreArgs = NULL, RECYCLE = TRUE,
           SIMPLIFY = FALSE, USE.NAMES = TRUE,
           .scheduling = c("static", "dynamic"))
clusterSplit(cl = NULL, seq)

parLapply(cl = NULL, X, fun, ...)
```

```
parSapply(cl = NULL, X, FUN, ..., simplify = TRUE,
          USE.NAMES = TRUE)
parApply(cl = NULL, X, MARGIN, FUN, ...)
parRapply(cl = NULL, x, FUN, ...)
parCapply(cl = NULL, x, FUN, ...)

parLapplyLB(cl = NULL, X, fun, ...)
parSapplyLB(cl = NULL, X, FUN, ..., simplify = TRUE,
            USE.NAMES = TRUE)
```

Arguments

cl	クラスターオブジェクトで，このパッケージかパッケージ **snow** で作られる．もし NULL なら登録された既定のクラスターを使う．
fun, FUN	関数か関数の名前の文字列．
expr	評価する表現式．
seq	分割するベクトル．
varlist	移出するオブジェクトの名前の文字ベクトル．
envir	変数を移出する環境．
x	clusterApply と clusterApplyLB に対してがベクトルで， parRapply と parCapply に対しては行列．
...	fun 又は FUN に渡される追加引数：前の引数への部分マッチングに注意する．
MoreArgs	fun に対する追加引数．
RECYCLE	論理値；もし真なら短い引数はリサイクルされる．
X	parLapply と parSapply に対してはベクトル(原子的かリスト)， parApply に対しては配列．
MARGIN	使用する次元を指定するベクトル．
simplify, USE.NAMES	論理値；sapply を見よ．
SIMPLIFY	論理値；mapply を見よ．
.scheduling	タスクを静的にノードに配置すべきか，それとも動的な負荷均等化を使うべきか?

Details

clusterCall は関数 fun を各ノードに同じ引数 ... で呼び出す．

clusterEvalQ は各クラスターノードで文字通りの表現式を評価する．これは evalq の並列バージョンで， clusterCall を起動するための簡便化関数である．

clusterApply は fun を最初のノードで引数 seq[[1]] と ... で呼び出し，二番目のノードで seq[[2]] と ... で呼び出す，云々．必要ならノードをリサイクルする．

clusterApplyLB は clusterApply の負荷均等バージョンである．もし seq の長さ p がノード数 n より大きくなければ，ジョブは p 個のノードに送られる．さもなければ，最初の n 個のジョブは順に n 個のノードに置かれる．最初のジョブが完了すると次のジョブが空いたノードに置かれる；これが全てのジョブが完了するまで続けられる．clusterApplyLB の使用は clusterApply の使用よりもより良くクラスターを利用するが，増加するやり取りがパフォーマンスを低下させる可能性がある．更に特定のジョブを実行するノードは予め定まっていない．

clusterMap は clusterApply の多重引数バージョンであり， mapply と Map に類似する．もし RECYCLE が真なら，短い引数はリサイクルされる (そしてどれもが長さゼロでないか全てがゼロでなければならない)；さもなければ，結果の長さは最も短い引数の長さになる．もし結果がノード数よりも大きければノードはリサイクルされる． (mapply は常に RECYCLE = TRUE を使用し，引数 SIMPLIFY = TRUE を持つ． Map は常に RECYCLE = TRUE を使う．)

clusterExport は varlist 中の名前が付いた変数のマスター R プロセス上での値を各ノードの大局的環境 ('作業空間'ともいう)中の同名の変数に付値する．変数がそこから移出されるマスターに関する環境の既定値は大局的環境である．

clusterSplit は seq を各クラスター用に連続する小片に分割し，ノード数に等しい長さのリストとして結果を返す．現在小片は長さがほぼ等しくなるように選ばれる：計算はマスター上でなされる．

parLapply, parSapply そして parApply lapply, sapply そして apply の並列バージョンである． parLapplyLB と parSapplyLB は負荷均等バージョンで， FUN の X の異なった要素への適用が非常に時間差があり，関数がランダムでないか再現可能な結果が不要の時に使われることを意図している．

parRapply と parCapply は行列 x の行と列に関して apply 関数を並列に適用する；これらは parApply よりも少々より効率的であるが結果の事後処理が少ない．

Value

clusterCall, clusterEvalQ そして clusterSplit に対してはノード毎に一つの要素を持つリスト．

clusterApply と clusterApplyLB に対しては seq と同じ長さのリスト．

clusterMap は mapply に従う．

clusterExport は何も返さない．

parLapply は長さ X のリストを返す．

parSapply と parApply はそれぞれ sapply と apply に従う．

parRapply と parCapply は常にベクトルを返す．もし FUN が常にスカラを返せば，これは列数か行数の長さを持つ：さもなければそれは返り値を連結したものになる．

ワーカーのどれかにエラーが発生すればマスターの側にエラーが起きる．

Note

これらはパッケージ **snow** 中のそれらとほとんど同じである．

二つの例外：parLapply は引数has argument X lapply との一貫性のため引数 x ではなく X を持つ，そして parSapply は sapply とマッチするように更新されてきている．

Author(s)

Luke Tierney と R Core.

パッケージ **snow** から導入された．

Examples

```
## 適当なクラスターサイズを選ぶためにオプション cl.cores を使う．
cl <- makeCluster(getOption("cl.cores", 2))

clusterApply(cl, 1:2, get("+"), 3)
```

```
xx <- 1
clusterExport(cl, "xx")
clusterCall(cl, function(y) xx + y, 2)

## mapply の例のように clusterMap を使う
clusterMap(cl, function(x, y) seq_len(x) + y,
          c(a =  1, b = 2, c = 3), c(A = 10, B = 0, C = -10))


parSapply(cl, 1:20, get("+"), 3)

## ブートストラップの例，これは様々な方法で行える：
clusterEvalQ(cl, {
  ## ワーカーを設定．clusterExport() を使うことも出来た
  library(boot)
  cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
  cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
  NULL
})
res <- clusterEvalQ(cl, boot(cd4, corr, R = 100,
                    sim = "parametric", ran.gen = cd4.rg, mle = cd4.mle))
library(boot)
cd4.boot <- do.call(c, res)
boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
        conf = 0.9, h = atanh, hinv = tanh)
stopCluster(cl)

## または
library(boot)
run1 <- function(...) {
   library(boot)
   cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
   cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
   boot(cd4, corr, R = 500, sim = "parametric",
        ran.gen = cd4.rg, mle = cd4.mle)
}
cl <- makeCluster(mc <- getOption("cl.cores", 2))
## これを再生可能にする
clusterSetRNGStream(cl, 123)
cd4.boot <- do.call(c, parLapply(cl, seq_len(mc), run1))
boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
        conf = 0.9, h = atanh, hinv = tanh)
stopCluster(cl)
```

`detectCores` *Detect the Number of CPU Cores*

Description

現在のホストの CPU コア数を検出しようと試みる．

Usage

```
detectCores(all.tests = FALSE, logical = TRUE)
```

Arguments

all.tests	論理値：もし真なら全ての既知の検査を適用する.
logical	論理値：もし可能なら物理的な CPU/コア数 (もし FALSE ならば)，又は論理的な CPU 数(もし TRUE ならば). を使用する．現在これは Linux, OS X, Sparc Solaris そして Windows だけで有効である.

Details

これは利用可能な CPU コアの数を検出しようと試みる.

それを行う Linux, OS X, FreeBSD, OpenBSD, Solaris, Irix そして Windows 用のメソッドがある．他の Unix 風システムでは detectCores(TRUE) が適用できるかもしれない.

R 3.3.0 以前では既定は Windows を除いて logical = FALSE であったが，logical = TRUE は Sparc Solaris と Windows でだけ有効であった(それらに対しては既定であった).

Value

整数，もし回答が未知なら NA.

これが何を表すかは正確には OS 依存である：既定では可能な場所ではそれは論理的 CPU (つまりハイパースレッド)であり物理的なコアやパッケージではない.

OS X では更に ‘現在のパワー管理モードで利用可能’と ‘ブートすれば利用可能’の違いがあり，この関数は最初を返す.

Sparc Solaris では logical = FALSE は物理的コア数を返し， logical = TRUE は利用可能なハードウェアスレッド数を返す． (ある Sparc CPU は CPU 毎に複数のコアを持ち，他はコアごとに複数のスレッドを持ち，両方を持つものもある．) 例えば CRAN のチェックサーバー中の UltraSparc T2 CPU は 8 コアを持つ単一の CPU で，各コアは 8 個のハードウェアスレッドを持つ．従って detectCores(logical = FALSE) は 8 を返し，detectCores(logical = TRUE) は 64 を返す.

仮想マシーンが使われているときは，結果はその特定の VM に対して利用可能な(又は潜在的に利用可能な) CPU 数を表していると期待できる.

Note

これは mclapply の mc.cores 引数に対してや makeCluster 中のコア数の指定に直接使うのには向いていない.

先ずそれは NA を返すかもしれないし，次にそれは許されるコア数を与えず，三つには Sparc Solaris とある種の Windows ボックスでは全ての論理的 CPU を一度に全て使おうとするのは合理的ではないからである.

Author(s)

Simon Urbanek と Brian Ripley

Examples

```
detectCores()
detectCores(logical = FALSE)
```

makeCluster *Create a Parallel Socket Cluster*

Description

並列に動作する R のコピーのセットを作り，ソケットを通じて通信する．

Usage

```
makeCluster(spec, type, ...)
makePSOCKcluster(names, ...)
makeForkCluster(nnodes = getOption("mc.cores", 2L), ...)

stopCluster(cl = NULL)

setDefaultCluster(cl = NULL)
```

Arguments

spec	クラスタのタイプに適当な指定．
names	R のワーカーのコピーをその上で実行するホスト名の文字列ベクトルか，正整数 (その場合その数のコピーが '**ローカルホスト**' の上で実行される)．
nnodes	フォークされるノードの数．
type	サポートされるタイプの一つ：'詳細'を見よ．
...	ワーカーを生み出す関数に渡されるオプション，'詳細'を見よ．
cl	クラス "cluster" のオブジェクト．

Details

makeCluster はサポートされるタイプの一つのクラスターを作る． creates a cluster of one of the supported types. 既定のタイプ "PSOCK" は makePSOCKcluster を呼び出す．タイプ "FORK" は makeForkCluster を呼びだす．他のタイプはパッケージ **snow** に渡される．

makePSOCKcluster はパッケージ **snow** 中の makeSOCKcluster の拡張版である． これは Rscript を指定されたホストの上でソケットから評価すべき表現式を聴き取り，結果を(シリアル化されたオブジェクトとして)返すワーカープロセスを設定する．

makeForkCluster は Windows では単なる名残である．Unix 風プラットフォームではこれはフォークによりワーカープロセスを作る．

ワーカーは最もしばしば同じホスト上でマスターとして稼働する．その時はオプションを設定する必要はない．

幾つかのオプションがサポートされている(主に makePSOCKcluster 用)：

master ワーカーに知られているようなマスターのホスト名．これはマスターに知られているものと同じではないかもしれず，プライベートなサブネットではこれを数値 IP アドレスとして指定する必要があるかもしれない．例えば OS X はあるマシーンを恐らくそれだけに知られている名前の 'somename.local' として検出する．

port ソケットコネクションのポート番号で，既定値は環境変数 R_PARALLEL_PORT から取り，それから無作為に範囲 11000:11999 から選ぶ．

timeout そのポートに対する時間切れの秒数．既定値は 30 日(そして POSIX 標準は最大31日までのサポートだけを要求する).

outfile ワーカーからの stdout と stderr コネクション出力をどこにダイレクトするか．"" はリダイレクションをしないことを指示する (それはワーカーがローカルマシン上にある時だけ役に立つ)．既定では '/dev/null' へ(Windows では 'nul:')．他の可能性はワーカーのホストのファイルパスである．ファイルは追加モードで開かれ，全てのワーカーのログが同じファイルに書かれる．

homogeneous 論理値．全てのホストは同一の設定で稼働し，従って Rscript はそれぞれ同じパスを使って始動されるか? さもなければ Rscript はワーカーの既定パス上になければならない．

rscript ワーカー上の Rscript へのパスで， homogeneous が真の時だけ使われる．既定ではマスター上のフルパス．

rscript_args '--no-environ' のような Rscript への追加引数の文字列ベクトル．

renice ワーカープロセスに対して設定される'優先度'，例えば低優先度なら 15． OS 依存：詳細は psnice を見よ．

rshcmd 他のホスト上のプロセスを指導するために実行される命令．既定値は ssh.

user 別のホストと通信する際に使われるユーザ名．

manual 論理値．もし真ならワーカーは手動で実行されなければならない．

methods 論理値．もし真(既定)ならばワーカーはパッケージ **methods** をロードする：それをロードしないとクラスターの起動 CPU 時間を約 30% 節約する．

useXDR 論理値．もし真なら(既定値)シリアル化は XDR を使う：大量のデータが移動され全てのノードがリトルエンディアンならば，これを偽に設定すると通信はかなり速くなる．

関数 makeForkCluster はソケットクラスターをフォークで作る(従って Windows では利用できない)．それはオプション port, timeout そして outfile をサポートし，常に useXDR = FALSE を使う．

ワーカーを呼び出し stopCluster でシャットダウンするのは良い作法である：しかしながらワーカーはそれらが命令を聴くソケットが利用不可能になるとそれ自体で終了し，もしマスターの R セッションが完了(またはそれが死亡)すればそうなるべきである．

関数 setDefaultCluster は現在のセッションに対して一つのクラスターを既定として登録する． setDefaultCluster(NULL) の使用はそのクラスターを停止するので，登録クラスターを取り除く．

Value

クラス c("SOCKcluster", "cluster") のオブジェクト．

Author(s)

Luke Tierney と R Core.

パッケージ **snow** に由来する．

mcaffinity *Get or Set CPU Affinity Mask of the Current Process*

Description

mcaffinity は現在のプロセスの CPU アフィニティマスクを回収したり設定したりする，つまりプロセスがその上で実行を許される CPU のセットである．(ここで CPU とは CPU になり得る論理 CPU，コア又はハイパースレッドユニットを意味する．)

Usage

```
mcaffinity(affinity = NULL)
```

Arguments

affinity	このプロセスをロックする CPU の指定 (数値ベクトル) 又はもし変更が要求されなければ NULL.

Details

mcaffinity は現在のプロセスの CPU アフィニティマスクを得たり(affinity = NULL)又は設定するのに使える． CPU アフィニティマスクはこのプロセスがその上で実行を許される整数の CPU 識別子(1 から始まる)のリストである．全てのシステムが CPU アフィニティマスクへのユーザのアクセスを許すわけではなく，サポートが全くされない場合は mcaffinity() は NULL を返す．ある種のシステムはマスク中にある CPU の数だけを考慮するかもしれない.

典型的には，論理 CPU の数よりも大きなセットを指定することは適正であり (しかし最大でも OS が扱えるだけの多さに限られる)，そしてシステムは実際に存在するセットに復帰する.

Value

もし CPU アフィニティがシステムによりサポートされていなければ NULL を，又はこのプロセスに対してアクティブな CPU アフィニティマスク中の CPU セットの整数ベクトル (これは affinity とは異なるかもしれない).

Author(s)

Simon Urbanek.

See Also

mcparallel

mcchildren *Low-level Functions for Management of Forked Processes*

Description

これらはフォークを使ったアプローチに対する低水準のサポート関数である.

これらは Windows では利用できず，名前空間から移出されない.

Usage

```
children(select)
readChild(child)
readChildren(timeout = 0)
selectChildren(children = NULL, timeout = 0)
sendChildStdin(child, what)
sendMaster(what)

mckill(process, signal = 2L)
```

Arguments

select	もし省略されると，全てのアクティブな子プロセスが返される．さもなければ select はプロセスのリストであるべきで，アクティブなリストからのそれらだけが返される.
child	子プロセス(クラス "childProcess" のオブジェクト) 又はプロセス ID (pid). ‘詳細’も見よ.
timeout	応答を諦める前に待つ時間切れ時間(秒単位，小数が可能).
children	子プロセスのリスト，単一の子プロセス，またはプロセス ID のベクトル，又は NULL．もし NULL ならば現在知られている子プロセス全てが提供される.
what	sendChildStdin に対して： 文字列かバイト型ベクトル．最初のケースでは要素は新行文字を使い一つにまとめられる． (しかし末尾には新行文字が加えられない!) sendMaster に対して： マスタープロセスに送るデータ．もし what がバイト型ベクトルでなければ，それはバイト型ベクトルにシリアル化される．空のバイト型ベクトルを送らないこと – それは内部的使用のため予約されている.
process	プロセス(クラス process のオブジェクト)又はプロセス ID (pid).
signal	整数：送るシグナル．値 2 (SIGINT), 9 (SIGKILL) そして 15 (SIGTERM) はかなりの程度に可搬的であるが，最大の可搬性のためには tools::SIGTERM 等を使う.

Details

children は現在アクティブな子プロセスを返す.

readChild は与えられた子プロセスからのデータを読み取る.

selectChildren は子プロセスに利用可能なデータがあるかどうかをチェックする.

readChildren は全ての子プロセスが利用可能なデータを持つかどうかチェックし，利用可能なデータを持つ最初の子プロセスから読み取る．

sendChildStdin は文字列(又はデータ)を一つ又は複数の子プロセスの入力に送る．もしマスターセッションが対話的ならば，それはまたマスタープロセスの標準出力にエコーされる (無効にされていない限り)．この関数はベクトル互換であり，従って child をリスト又はプロセス ID のベクトルとして指定できる．

sendMaster はデータを子プロセスからマスタープロセスに送る．

mckill はシグナルを一つの子プロセスに送る：これはパッケージ **tools** 中の pskill と同値である．

Value

children は(もしかすると空の)プロセス ID であるクラス "process" のオブジェクトのリストを返す．

readChild と readChildren はもしデータが利用可能ならば "pid" 属性を持つバイト型ベクトル，もし子プロセスが終了していればプロセス ID を持つ長さ 1 の整数ベクトル，又は子プロセスが最早存在しなければ NULL を返す (readChildren に対して子プロセスが全く無い)．

selectChildren はもし時間切れならば TRUE を，もしエラーが起きていれば FALSE (つまりもしマスタープロセスが中断されていれば)，又はデータが利用可能な子プロセスのプロセス ID の整数ベクトル，もしくは子プロセスが存在しなければ NULL を返す．

sendChildStdin は TRUE 値のベクトルを返す (child の各メンバーに対して一つ)，又はエラーを出す．

sendMaster は TRUE を返すかエラーを出す．

mckill は TRUE を返す．

警告

これはエキスパート使用専用の非常に低水準の API である．

Author(s)

Simon Urbanek と R Core.

Derived from the **multicore** パッケージから導入された．

See Also

mcfork, sendMaster, mcparallel

Examples

```
## Not run:
p  <- mcparallel(scan(n = 1, quiet = TRUE))
sendChildStdin(p, "17.4\n")
mccollect(p)[[1]]

## End(Not run)
```

mcfork *Fork a Copy of the Current R Process*

Description

これらは低水準の関数であり， Windows では利用できず，名前空間から移出されない．

mcfork は新しい子プロセスを現在の R プロセスのコピーとして作る．

mcexit は現在の子プロセスを閉じ，必要に応じてマスタープロセスに通知する．

Usage

```
mcfork(estranged = FALSE)

mcexit(exit.code = 0L, send = NULL)
```

Arguments

estranged	論理値，もし TRUE ならば新しいプロセスは親プロセスと一切結びつきを持たず，子プロセスのリストに示されず退出時にも抹消されない．
exit.code	プロセスの退出コード．慣習から 0L は正常な退出を，1L はエラーを意味する．
send	もし NULL でなければ退出前にこのデータを送る (sendMaster の使用と同値)．

Details

mcfork 関数は fork システムコールへのインタフェイスを提供する．更にそれはマスターと子プロセス間にパイプを設定し，それにより子プロセスからマスターにデータを送ったり (sendMaster を見よ)，子プロセスの 'stdin' をマスタープロセスが所持する別のパイプに再マップ (sendChildStdin を見よ)することが出来るようになる．.

もし fork システムコールに不慣れなら，この関数を直接使ってはならない．なぜならそれは関係する R プロセス間に非常に複雑な相互作用をもたらすからである．

要約すれば fork は現在のプロセスのコピー(子プロセス) を生み出し，それはマスター(親)プロセスと並列に動作可能になる．フォークの時点で双方のプロセスは作業空間，大局的オプション，ロード済みパッケージ等正確に同じ状態を共有する．

フォークは現代のオペレーティング・システムでは相対的に安上がりで，使用メモリーの実際のコピーは作られず，代わりに双方のプロセスは同じメモリーを共有し，変更された部分だけがコピーされる．並列な作業環境を設定する必要がないため，これは mcfork を並列プロセスに対する理想的なツールにする．データとコードは最初から自動的に共有される．

mcexit は子プロセス中で実行されるべきである．これは send をマスタープロセスに送り (NULL で無い限り)，そして子プロセスをシャットダウンする．子プロセスは未移出の関数 parallel:::rmChild により実行されるように，それにシグナル SIGUSR1 を送ることでもシャットダウン出来る．

Value

mcfork はクラス "childProcess" のオブジェクトをマスタープロセスに返し，子プロセスにクラス "masterProcess" のオブジェクトを返す：二つのクラスはクラス "process" を継承する．もし estranged が TRUE に設定されると子プロセスはクラス "estrangedProcess" になりマスタープロセスと交信できないし子プロセスのリストにも示されない．これらはリストであり，pid (他のプロセスの id) と，プロセス間パイプの現在のプロセス中の最後に対する二つのファイル記述子数 fd を成分に持つ．

mcexit は決して返らない．

GUI/埋め込み環境

mcfork とそれに依存する高水準関数 (例えば mcparallel, mclapply そして pvec) を GUI や埋め込み環境で使うことは 全く勧められない．なぜならそれは複数のプロセスが同じ GUI を共有することになり混乱しやすい(そして恐らくクラッシュする) からである．子プロセスは決してスクリーンを使うグラフィックスデバイスを使うべきではない． OS X の R.app ではこれを使用可能にするある種の処置が取られてきているが，サードパーティのフロントエンドのユーザはそれらのドキュメントを吟味すべきである．

これは又フォーク前に作られた他のコネクション(例えば X サーバー)と例えばグラフィックスデバイスで開かれたファイルにも当てはまる．

Tclはイベントループを実行するため，これらの目的にとっては **tcltk** は GUI と考えられることを注意する．そうしたイベントループは子プロセス中では禁じられているが， Tk グラフィカルコネクションに関しては依然問題があり得る．

警告

これはエキスパート使用専用の非常に低レベルの API である．

Author(s)

Simon Urbanek と R Core.

Derived from the パッケージ **multicore** 由来.

See Also

mcparallel, sendMaster

Examples

```
## これは例として実行すると動作するが，貼り付けすると動作しない.
p <- parallel:::mcfork()
if (inherits(p, "masterProcess")) {
    cat("I'm a child! ", Sys.getpid(), "\n")
    parallel:::mcexit(,"I was a child")
}
cat("I'm the master\n")
unserialize(parallel:::readChildren(1.5))
```

mclapply *Parallel Versions of* lapply *and* mapply *using Forking*

Description

mclapply は lapply の並列バージョンであり， X と同じ長さのリストを返す．その各要素は FUN を対応する codeX の要素に適用した結果である．

これはフォークを使っており，従って mc.cores = 1 で無い限り Windows では使えない．

mcmapply は mapply の並列バージョンであり，そして mcMap は Map に対応する．

Usage

```
mclapply(X, FUN, ...,
         mc.preschedule = TRUE, mc.set.seed = TRUE,
         mc.silent = FALSE, mc.cores = getOption("mc.cores", 2L),
         mc.cleanup = TRUE, mc.allow.recursive = TRUE)

mcmapply(FUN, ...,
         MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE,
         mc.preschedule = TRUE, mc.set.seed = TRUE,
         mc.silent = FALSE, mc.cores = getOption("mc.cores", 2L),
         mc.cleanup = TRUE)

mcMap(f, ...)
```

Arguments

X	ベクトル(原子的又はリスト)又は表現式ベクトル．他のオブジェクト(クラス付きオブジェクトを含む)は as.list で変換される．
FUN	(mclapply) X の各要素，又は (mcmapply) 並列に ... へ適用される関数．
f	... に並列に適用される関数．
...	mclapply に対しては FUN へのオプションの引数． mcmapply と mcMap に対してはベクトル又はリスト入力： mapply を見よ．
MoreArgs, SIMPLIFY, USE.NAMES	mapply を見よ．
mc.preschedule	もし TRUE に設定すると計算は最初(最大)コア数だけの多さのジョブに分割されそれからジョブが開始する．各ジョブは可能性として一つ以上の値をカバーしている．もし FALSE に設定されると X の各値に対して一つのジョブがフォークされる．短い計算や X 中に大量の値がある場合は前者が好ましい．後者は完了時間に大きな変動があり mc.cores に比較してそれほど大量の X の値が無いようなジョブに適している．
mc.set.seed	mcparallel を見よ．
mc.silent	もし TRUE に設定すると全てのフォークされた並列プロセスに対して‘stdout’上の全ての出力は抑制される (‘stderr’ は影響されない)．
mc.cores	使用するコア数，つまり最大幾つの子プロセスを同時に実行するか．オプションはもし設定されていれば環境変数 MC_CORES から初期化される．少なくとも一つのコアがなければならず，並列化のためには少なくとも二つのコアがいる．

mc.cleanup もし TRUE に設定するとこの関数でフォークされた全ての子プロセスはこの関数が復帰する前に(SIGTERM を送ることにより)消滅する．普通の状況では mclapply は子プロセスが結果を配送するのを待つので，このオプションは普通 mclapply が中断された時だけ効果を持つ．もし FALSE に設定されると子プロセスは収集されるが強制的に停止されることはない．特別なケースとしてこの引数は SIGTERM の代わりに子プロセスを消滅するのに使われる幾つかのシグナルに設定することが出来る．

mc.allow.recursive 真でない限り，子プロセス内での mclapply の使用は子プロセスを使い再びフォークしない．

Details

mclapply は provided mc.cores > 1 であれば lapply の並列化バージョンである：mc.cores == 1 に対してはそれは単に lapply を呼び出す．

既定 (mc.preschedule = TRUE) では入力 X はコア数と同じだけの部分に分割され (現在値はコアに順番に割り当てられる，つまり最初の値がコア 1，二番目がコア 2, . . . ，(core + 1)番目がコア 1 等)，そしてそれから一つのプロセスが各コアに対してフォークされ，結果が回収される．

事前のスケジューリング無しでは，個別のジョブが X の各値に対してフォークされる．mc.cores 以上のジョブが同時に実行されないことを保証するために，その数だけフォークされると次のフォークの前にマスタープロセスは子プロセスが完了するまで待つ．

実行の並列化特性のために乱数は lapply を使用した時にそうなるようには (乱数系列の中で)シーケンシャルではない．それらは各フォークされたプロセスに対してはシーケンシャルであるが，全体としての全てのジョブに対してはそうではない．結果を mc.preschedule = TRUE で再現可能にする方法については mcparallel やパッケージのビニエットを見よ．

ノート：ファイル記述子 (とプロセス)の数は普通オペレーティングシステムにより制限されているので， 100 位以上(ulimit -n 等とするか OS のドキュメントを見よ) のコアを使おうとすると，開かれたファイル記述子の許可限界を増やしておかない限り，問題が起きるかもしれない (フォークはエラー "unable to create a pipe" で失敗する)．

Value

mclapply に対しては X と同じ長さで X で名前が付けられたリスト．

mcmapply に対してはベクトルか配列： mapply を見よ．

mcMap に対してはリスト．

フォークで作られたプロセスはその作業を try(..., silent = TRUE) の内部で実行するので，もしエラーが起きるとそれらは返り値中のクラス "try-error" のオブジェクトとして保存され警告がでる．ジョブは典型的に X の一つ以上の値を含むので "try-error" オブジェクトは失敗に含まれる全ての値について返される．それら全てが失敗しなかったとしてでもある．

警告

これらの関数を GUI や埋め込み環境で使うことは 全く勧められない．なぜならそれは複数のプロセスが同じ GUI を共有することになり混乱しやすい(そして恐らくクラッシュする) からである．子プロセスは決してスクリーンを使うグラフィックスデバイスを使うべきではない．

OS X の R.app ではこれを使用可能にするある種の処置が取られてきているが，サードパーティのフロントエンドのユーザはそれらのドキュメントを吟味すべきである．

Tclはイベントループを実行するため，これらの目的にとっては **tcltk** は GUI と考えられることを注意する．そうしたイベントループは子プロセス中では禁じられているが， Tk グラフィカルコネクションに関しては依然問題があり得る．

Author(s)

Simon Urbanek と R Core.

パッケージ **multicore** に由来する．

See Also

mcparallel, pvec, parLapply, clusterMap.

sapply のような結果には simplify2array.

Examples

```
simplify2array(mclapply(rep(4, 5), rnorm))
# 全ての値に同じ乱数を使う
set.seed(1)
simplify2array(mclapply(rep(4, 5), rnorm, mc.preschedule = FALSE,
                mc.set.seed = FALSE))

## これを clusterCallContrast の例と比較せよ
library(boot)
cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
mc <- getOption("mc.cores", 2)
run1 <- function(...) boot(cd4, corr, R = 500, sim = "parametric",
                           ran.gen = cd4.rg, mle = cd4.mle)
## これを再現可能にするには：
set.seed(123, "L'Ecuyer")
res <- mclapply(seq_len(mc), run1)
cd4.boot <- do.call(c, res)
boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
        conf = 0.9, h = atanh, hinv = tanh)
```

---

mcparallel *Evaluate an* R *Expression Asynchronously in a Separate Process*

Description

これらの関数はフォークに基づいており，従って Windows では利用できない．

mcparallel は与えられた表現式を評価する並列 R プロセスを開始する．

mccollect は一つもしくはそれ以上の並列プロセスから結果を回収する．

Usage

```
mcparallel(expr, name, mc.set.seed = TRUE, silent = FALSE,
           mc.affinity = NULL, mc.interactive = FALSE,
   detached = FALSE)

mccollect(jobs, wait = TRUE, timeout = 0, intermediate = FALSE)
```

Arguments

expr	評価する表現式(コード中ではスクリーン使用のデバイスや GUI 要素を使わない.
name	ジョブに関連付けられるオプションの名前 (長さ 1 の文字列).
mc.set.seed	論理値：'乱数'節を見よ.
silent	もし TRUE に設定すると標準出力上の全ての出力は抑制される(標準エラーは影響されない).
mc.affinity	子プロセスを制限する CPU を指定する(1-based)数値ベクトル. もしくは NULL ならば CPU のアフィニティーを変更しない.
mc.interactive	論理値, もし TRUE か FALSE ならば子プロセスはそれぞれ対話的もしくは非対話的とされる. もし NA なら子プロセスは親プロセスから対話フラグを受け継ぐ.
detached	論理値, もし TRUE ならジョブは現在のセッションから取り除かれ, 如何なる結果も配送し返せない - これは副作用だけのためのコードに対して使われる.
jobs	結果を収集する対象のジョブのリスト(又は単一のジョブ). 別法として jobs はまたプロセス ID の整数ベクトルでも良い. もし省略されると collect は現在存在する全ての子プロセスを待つ.
wait	もし FALSE に設定すると, これは今から timeout 秒経過以内に利用可能な任意の結果をチェックする. さもなければそれは全ての指定されたジョブが終了するのを待つ.
timeout	ジョブの結果のチェックの時間切れ(秒単位). wait が FALSE の時だけ適用される.
intermediate	FALSE 又は collect が結果を待つ間に呼び出される関数. この関数はこれまで受け取った結果のリストである一つのパラメータと共に呼び出される.

Details

mcparallel は expr 表現式を現在の R プロセスに対して並列に評価する. 全てが並列プロセスと現在のプロセスの間で読み込み専用(又は実際には copy-on-write で)で共有される, つまり表現式の副作用はいずれも主プロセスに影響しない. 並列実行の結果は mccollect 関数を用いて収集することが出来る.

mccollect 関数は並列ジョブ(又は実際は任意の子プロセス)から任意の利用可能な結果を収集する. もし wait が TRUE ならば collect は各ジョブに対する最後に報告された結果を含むリストを返す前に全ての指定されたジョブが終了するのを待つ. もし wait が FALSE ならば mccollect はその時に利用可能な任意の値をチェックするだけで, ジョブが終了するのを待たない. もし jobs が指定されると, そこにリストされていないジョブは影響されないし働きかけも受けない.

ノート：もし expr が sendMaster のような低水準の多コア関数を使えば一つのジョブは結果を何度も提供するので, それらを正確に解釈するのはユーザの責任になる.

mccollect はその後にはジョブが最早利用できないその結果を送ってしまった終了済みジョブに対しては NULL を返す.

mc.affinity パラメータは子プロセスを特定の CPU に制限するのに使うことが出来る. この特性の利用可能性と程度はシステム依存である (例えばある種のシステムは CPU の数だけを考慮し，それを全く無視するものもある).

Value

mcparallel はクラス "parallelJob" のオブジェクトを返す，これは "childProcess" を継承する (mcfork に対するヘルプの'値'節を見よ). もし引数 name が提供されているとこれは追加の成分 name を持つ.

mccollect はリスト中で利用可能な任意の結果を返す. 結果は指定されたジョブと同じ順序を持つ. もし複数のジョブがありジョブが名前を持てばそれが結果の名前に使われる. さもなければそのプロセス ID が使われる. もし指定された子プロセスがどれも稼働していなければそれは NULL を返す.

乱数

もし mc.set.seed = FALSE ならば子プロセスは現在の R セッションと同じ初期乱数生成器 (RNG) 状態を持つ. もし RNG (又は .Random.seed が保存された作業スペースから復元されると) が使われていると子プロセスは現在のセッションと同じ箇所から乱数抽出を開始する. もし RNG がまだ使われていなければ，子プロセスはそれが RNG を最初に使うときに時刻とプロセス ID に基づいて乱数種を設定する：これは現在のセッションや他の子プロセスと異なった乱数ストリームを与えることをかなりの程度に保証する.

mc.set.seed = TRUE の時の挙動は RNGkind("L'Ecuyer-CMRG") が選択されている時だけ異なる. その時は子プロセスがフォークされる度にそれに次のストリームが与えられる (nextRNGStream を見よ). 従ってもしその生成器を選び，乱数種を設定しそして mc.reset.stream を mcparallel の最初の使用の前に呼びだせば，シミュレーションの結果は同じ作業が最初，二番目，...のフォークされたプロセスに与えられる限り再生性を持つ.

Note

Package パッケージ **multicore** はまた関数 collect と parallel を移出する. これらの名前は容易にマスクされる (例えばパッケージ **lattice** は同様に関数 parallel を持つ) のでこのパッケージでは提供されない.

Author(s)

Simon Urbanek と R Core.

パッケージ **multicore** に由来する(しかし異なった RNG ストリームを持つ).

See Also

pvec, mclapply

Examples

```
p <- mcparallel(1:10)
q <- mcparallel(1:20)
# 両方のジョブが終了するのを待ち，それから全ての結果を集める
res <- mccollect(list(p, q))
```

```
p <- mcparallel(1:10)
mccollect(p, wait = FALSE, 10) # 結果を回収する(それは速いので)
mccollect(p, wait = FALSE)     # ジョブに終了シグナルを送る
mccollect(p, wait = FALSE)     # そうしたジョブは最早無い


# 素朴な並列 lapply は mcparallel だけを使い作れる：
jobs <- lapply(1:10, function(x) mcparallel(rnorm(x), name = x))
mccollect(jobs)
```

---

pvec *Parallelize a Vector Map Function using Forking*

Description

pvec は関数のベクトル要素への実行をベクトルを分割し各部分を一つのコアに任せることで並列実行する．関数はベクトル化マップでなければならない，つまりそれはベクトル入力を取りベクトルの分割とは無関係に入力と正確に同じ長さの出力を作る．

これはフォークに基づいており，従って mc.cores = 1 でない限り Windows では利用できない．

Usage

```
pvec(v, FUN, ..., mc.set.seed = TRUE, mc.silent = FALSE,
     mc.cores = getOption("mc.cores", 2L), mc.cleanup = TRUE)
```

Arguments

v	操作対象のベクトル．
FUN	ベクトルの各部分に呼び出される関数．
...	ベクトルの後に FUN に渡される追加の引数．
mc.set.seed	mcparallel を見よ．
mc.silent	もし TRUE に設定すると ‘stdout’ への全ての出力は全てのフォークされた並列プロセスに対して抑制される(‘stderr’ は影響を受けない)．
mc.cores	使用するコア数，つまり最大幾つまでの子プロセスを同時に実行するか．少なくとも一つでなければならない．このオプションはもし設定されていれば環境変数 MC_CORES から初期化される．
mc.cleanup	mclapply 中のこの引数の説明を見よ．

Details

pvec は FUN(x, ...) を並列化する，ここで FUN は x と同じ長さのベクトルを返す． FUN はまたピュアでなければならない(つまり副作用を持たない)． なぜなら副作用は並列プロセスからは収集できないからである．ベクトルはほぼ同じサイズのの副ベクトルに分割され，それぞれに FUN が実行される．原理的には必ずしもマップでない関数を使うことは可能であるが，分割は理論的には任意であるため解釈はケース依存である (そうした場合警告が与えられる)．

pvec と mclapply の主要な違いは， mclapply は FUN を各要素に個別に適用する一方 pvec は c(FUN(x[1]), FUN(x[2])) が FUN(x[1:2]) に等しいことを仮定するのでコア(またはもし少なければ要素)があるだけ多くの各々がベクトルの部分集合を処理する FUN の呼び出

しに分割されることである．これは pvec を mclapply より効率的にするが FUN に関する上記の仮定を必要とする．

もし mc.cores == 1 ならこれは現在のプロセスで FUN(v, ...) を評価する．

Value

計算の結果 – 成功の場合それは v と同じ長さになる．もしエラーが起きるか関数がマップでなければ，結果は短かったり長かったりし，警告が出る．

Note

並列化の特性から，エラーは文字列として返され抹消された子プロセスは単に非存在データとして現れるので，エラー処理は通常の規則に従わない．従って正しいサイズかどうか確証するため結果の長さをチェックするのはユーザの責任である． pvec はもしそうしたことが起きればそうした出力が意図的なものかどうか分からないので警告を出す．

これを GUI フロントエンドと共に使うことが勧められないことについては mcfork を見よ．

Author(s)

Simon Urbanek と R Core.

パッケージ **multicore** 由来．

See Also

mcparallel, mclapply, parLapply, clusterMap.

Examples

```
x <- pvec(1:1000, sqrt)
stopifnot(all(x == sqrt(1:1000)))


# 比較的遅い操作として大きなデータセット中のデータ文字列を
# Unix 時間に変換する
# それで先ずランダムな日付を最初に作ろう
# (コアを二つ使う小さなテスト： options("mc.cores") を設定し
# 良い大きなスケールのテストのため N を増やす．)
N <- 1e5
dates <- sprintf('%04d-%02d-%02d', as.integer(2000+rnorm(N)),
                 as.integer(runif(N, 1, 12)), as.integer(runif(N, 1, 28)))

system.time(a <- as.POSIXct(dates))

# しかし書式を指定するほうが速い
system.time(a <- as.POSIXct(dates, format = "%Y-%m-%d"))

# pvec の方が速いはずだが，システムのオーバーヘッドは高くなり得る
system.time(b <- pvec(dates, as.POSIXct, format = "%Y-%m-%d"))
stopifnot(all(a == b))

# これに mclapply を使うと相当遅くなる．なぜなら各値は
# as.POSIXct() を lapply(dates, as.POSIXct) として
# 個別に呼び出す必要があるから
system.time(c <- unlist(mclapply(dates, as.POSIXct,  format = "%Y-%m-%d")))
```

```
stopifnot(all(a == c))
```

RNGstreams *Implementation of Pierre L'Ecuyer's RngStreams*

Description

これは Pierre L'Ecuyer の 'RngStreams' の擬似乱数の多重ストリームの R への再実装である.

Usage

```
nextRNGStream(seed)
nextRNGSubStream(seed)

clusterSetRNGStream(cl = NULL, iseed)
mc.reset.stream()
```

Arguments

seed	'"L'Ecuyer-CMRG"' RNG が使われているときは .Random.seed が与えるような長さ 7 の整数ベクトル. 個々の値については RNG を見よ.
cl	このパッケージかパッケージ **snow** からのクラスター, 又は(もし NULL なら)登録されたクラスター.
iseed	set.seed へ提供される整数, もしくは再現可能な乱数種を設定しないのなら NULL.

Details

'RngStream' インタフェイスは(可能性として)擬似乱数の多重ストリームで動作する：各作業が別個の RNG ストリームを付値できるのでこれは並列計算の作業では特に有用である.

これは基礎の生成器として L'Ecuyer (1999) の RNGkind("L'Ecuyer-CMRG") を使う. これは6個の(符号付き)整数のベクトルを乱数種として持ち, 周期は約 2^{191} である. 各'ストリーム'は長さ 2^{127} の周期で, それを長さ 2^{76} の周期を持つ 'サブストリーム'に分割する.

L'Ecuyer *et al* (2002) のアイデアは並列計算の各々に対して別個のストリームを使い (乱数発生器は決して同期しないことが保証される), 必要なら並列計算自体がサブストリームを使うというものである. オリジナルのインタフェイスは最初のストリームのオリジナルな乱数種, 現在のストリームのオリジナルな乱数種, そして現在の乱数種を保存する：これを R に移植することは可能であるが, 関連する .Random.seed の値を保存して作業するほうが簡単である：例を見よ.

clusterSetRNGStream は "L'Ecuyer-CMRG" RNG を選択し, それからクラスターのメンバーにストリームを配分する. オプションでストリームの乱数種を set.seed(iseed) を使い設定する (さもなければ L'Ecuyer 生成器を選択した後に, それらはマスタープロセスの現在の乱数種から設定される.)

L'Ecuyer の乱数発生器と乱数種を設定した後の mc.reset.stream() の呼び出しは mcparallel(mc.set.seed = からの実行を再生可能にする. これは mclapply と pvec により内部的になされる. (それはマスタープロセスの乱数種を設定しないので, これらの関数のシリアル復帰バージョンに影響しないことを注意する.)

Value

nextRNGStream と nextRNGSubStream に対しては .Random.seed に付値可能な値.

Note

L'Ecuyer の C コードへのインタフェイスは CRAN パッケージ **rlecuyer** と **rstream** にある.

Author(s)

Brian Ripley

References

L'Ecuyer, P. (1999) Good parameters and implementations for combined multiple recursive random number generators. *Operations Research* **47**, 159–164.

L'Ecuyer, P., Simard, R., Chen, E. J. and Kelton, W. D. (2002) An object-oriented random-number package with many long streams and substreams. *Operations Research* **50** 1073–5.

See Also

R の組み込み乱数生成器のより完全な詳細は RNG.

パッケージ **parallel** のビニエット.

Examples

```
RNGkind("L'Ecuyer-CMRG")
set.seed(123)
(s <- .Random.seed)
## 乱数を含むある作業.
nextRNGStream(s)
nextRNGSubStream(s)
```

---

splitIndices *Divide Tasks for Distribution in a Cluster*

Description

これは一つのクラスター中のノードに作業を割り当てるために， 1:nx をほぼ等しい長さ ncl のリストに分割する.

これは主に内部的使用であるが，ある種のパッケージの作者はそれが役に立つことを見出してきた.

Usage

```
splitIndices(nx, ncl)
```

Arguments

nx	作業の数.
ncl	クラスターノードの数.

Value

長さ ncl のリストで，各要素は整数ベクトル.

Examples

```
splitIndices(20, 3)
```

Index

∗Topic **distribution**
RNGstreams, 21
∗Topic **interface**
mcaffinity, 9
mcchildren, 10
mcfork, 12
mclapply, 14
mcparallel, 16
pvec, 19
∗Topic **package**
parallel-package, 2
∗Topic **sysdata**
RNGstreams, 21
∗Topic **utility**
splitIndices, 22

apply, *4*
as.list, *14*

children *(*mcchildren*)*, 10
clusterApply, 2
clusterApplyLB *(*clusterApply*)*, 2
clusterCall *(*clusterApply*)*, 2
clusterEvalQ *(*clusterApply*)*, 2
clusterExport *(*clusterApply*)*, 2
clusterMap, *16*, *20*
clusterMap *(*clusterApply*)*, 2
clusterSetRNGStream *(*RNGstreams*)*, 21
clusterSplit *(*clusterApply*)*, 2

detectCores, 5

evalq, *3*

lapply, *4*, *14, 15*

makeCluster, 7
makeForkCluster, *2*
makeForkCluster *(*makeCluster*)*, 7
makePSOCKcluster *(*makeCluster*)*, 7
Map, *4*, *14*
mapply, *3, 4*, *14, 15*
mc.reset.stream, *18*
mc.reset.stream *(*RNGstreams*)*, 21
mcaffinity, 9
mcchildren, 10
mccollect *(*mcparallel*)*, 16
mcexit *(*mcfork*)*, 12
mcfork, *11*, 12, *18*, *20*
mckill *(*mcchildren*)*, 10
mclapply, 14, *18–21*
mcMap *(*mclapply*)*, 14
mcmapply *(*mclapply*)*, 14
mcparallel, *9*, *11*, *13–16*, 16, *19–21*

nextRNGStream, *2*, *18*
nextRNGStream *(*RNGstreams*)*, 21
nextRNGSubStream *(*RNGstreams*)*, 21

parallel *(*parallel-package*)*, 2
parallel-package, 2
parApply *(*clusterApply*)*, 2
parCapply *(*clusterApply*)*, 2
parLapply, *16*, *20*
parLapply *(*clusterApply*)*, 2
parLapplyLB *(*clusterApply*)*, 2
parRapply *(*clusterApply*)*, 2
parSapply *(*clusterApply*)*, 2
parSapplyLB *(*clusterApply*)*, 2
pskill, *2*, *11*
psnice, *2*, *8*
pvec, *16*, *18*, 19, *21*

R_PARALLEL_PORT *(*makeCluster*)*, 7
readChild *(*mcchildren*)*, 10
readChildren *(*mcchildren*)*, 10
RNG, *21, 22*
RNGkind, *18*
RNGstreams, 21

sapply, *3, 4*, *16*
selectChildren *(*mcchildren*)*, 10
sendChildStdin, *12*
sendChildStdin *(*mcchildren*)*, 10
sendMaster, *11–13*, *17*
sendMaster *(*mcchildren*)*, 10
set.seed, *21*
setDefaultCluster *(*makeCluster*)*, 7
SIGTERM, *10*

simplify2array, *16*
splitIndices, 22
stderr, *8*
stdout, *8*
stopCluster, *8*
stopCluster *(*makeCluster*)*, 7