Package 'parallel'

R-core

November 7, 2014

1 Introduction

Package **parallel** was first included in R 2.14.0. It builds on the work done for CRAN packages **multicore** (**?**) and **snow** (**?**) and provides drop-in replacements for most of the functionality of those packages, with integrated handling of random-number generation.

Parallelism can be done in computation at many different levels: this package is principally concerned with 'coarse-grained parallelization'. At the lowest level, modern CPUs can do several basic operations simultaneously (e.g. integer and floating-point arithmetic), and several implementations of external BLAS libraries use multiple threads to do parts of basic vector/matrix operations in parallel. Several contributed R packages use multiple threads at C level *via* OpenMP or pthreads.

This package handles running much larger chunks of computations in parallel. A typical example is to evaluate the same R function on many different sets of data: often simulated data as in bootstrap computations (or with 'data' being the random-number stream). The crucial point is that these chunks of computation are unrelated and do not need to communicate in any way. It is often the case that the chunks take approximately the same length of time. The basic computational model is

(a) Start up M 'worker' processes, and do any initialization needed on the workers.

(b) Send any data required for each task to the workers.

(c) Split the task into M roughly equally-sized chunks, and send the chunks (including the R code needed) to the workers.

(d) Wait for all the workers to complete their tasks, and ask them for their results.

(e) Repeat steps (b–d) for any further tasks.

(f) Shut down the worker processes.

Amongst the initializations which may be needed are to load packages and initialize the random-number stream.

There are implementations of this model in the functions `mclapply` and `parLapply` as near-drop-in replacements for `lapply`.

A slightly different model is to split the task into $M_1 > M$ chunks, send the first M chunks to the workers, then repeatedly wait for any worker to complete and send it the next remaining task: see the section on 'load balancing'.

In principle the workers could be implemented by threads[1] or lightweight processes, but in the current implementation they are full processes. They can be created in one of three ways:

[1] only 'in principle' since the R interpreter is not thread-safe.

1. *Via* `system("Rscript")` or similar to launch a new process on the current machine or a similar machine with an identical R installation. This then needs a way to communicate between master and worker processes, which is usually done *via* sockets.

   This should be available on all R platforms, although it is conceivable that zealous security measures could block the inter-process communication *via* sockets. Users of Windows and OS X may expect pop-up dialog boxes from the firewall asking if an R process should accept incoming connections.

   Following **snow**, a pool of worker processes listening *via* sockets for commands from the master is called a 'cluster' of nodes.

2. *Via* forking. *Fork* is a concept[2] from POSIX operating systems, and should be available on all R platforms except Windows. This creates a new R process by taking a complete copy of the master process, including the workspace and state of the random-number stream. However, the copy will (in any reasonable OS) share memory pages with the master until modified so forking is very fast.

   The use of forking was pioneered by package **multicore**.

   Note that as it does share the complete process, it also shares any GUI elements, for example an R console and on-screen devices. This can cause havoc.[3]

   There needs to be a way to communicate between master and worker. Once again there are several possibilities since master and workers share memory. In **multicore** the initial fork sends an R expression to be evaluated to the worker, and the master process opens a pipe for reading that is used by the worker to return the results. Both that and creating a cluster of nodes communicating *via* sockets are supported in package **parallel**.

3. Using OS-level facilities to set up a means to send tasks to other members of a group of machines. There are several ways to do that, and for example package **snow** can make use of MPI ('message passing interface') using R package **Rmpi**. Communication overheads can dominate computation times in this approach, so it is most often used on tightly-coupled networks of computers with high-speed interconnects.

   CRAN packages following this approach include **GridR** (using Condor or Globus) and **Rsge** (using SGE, currently called 'Oracle Grid Engine').

   It will not be considered further in this vignette, but those parts of **parallel** which provide **snow**-like functions will accept **snow** clusters including MPI clusters.

The landscape of parallel computing has changed with the advent of shared-memory computers with multiple (and often many) CPU cores. Until the late 2000's parallel computing was mainly done on clusters of large numbers of single- or dual-CPU computers: nowadays even laptops have two or four cores, and servers with 8, 32 or more cores are commonplace. It is such hardware that package **parallel** is designed to exploit. It can also be used with several computers running the same version of R connected by (reasonable-speed) ethernet: the computers need not be running the same OS.

Note that all these methods of communication use `serialize`/`unserialize` to send R objects between processes. This has limits (typically hundreds of millions of elements) which a well-designed parallelized algorithm should not approach.

[2] http://en.wikipedia.org/wiki/Fork_(operating_system)

[3] Some precautions are taken on Mac OS X: for example the event loops for `R.app` and the `quartz` device are inhibited in the child. This information is available at C level in the `Rboolean` variable `R_isForkedChild`.

2 Numbers of CPUs/cores

In setting up parallel computations it can be helpful to have some idea of the number of CPUs or cores available, but this is a rather slippery concept. Nowadays almost all physical CPUs contain two or more cores that run more-or-less independently (they may share parts of the cache memory, and they do share access to RAM). However, on some processors these cores may themselves be able to run multiple tasks simultaneously, and some OSes (e.g. Windows) have the concept of *logical* CPUs which may exceed the number of cores.

Note that all a program can possibly determine is the total number of CPUs and/or cores available. This is not necessarily the same as the number of CPUs available *to the current user* which may well be restricted by system policies on multi-user systems. Nor does it give much idea of a reasonable number of CPUs to use for the current task: the user may be running many R processes simultaneously, and those processes may themselves be using multiple threads through a multi-threaded BLAS, compiled code using OpenMP or other low-level forms of parallelism. We have even seen instances of **multicore**'s `mclapply` being called recursively,[4] generating $2n+n^2$ processes on a machine estimated to have $n = 16$ cores.

But in so far as it is a useful guideline, function `detectCores()` tries to determine the number of CPU cores in the machine on which R is running: it has ways to do so on all known current R platforms. What exactly it measures is OS-specific: we try where possible to report the number of physical cores available.

On Windows the default is to report the number of logical CPUs. On modern hardware (e.g. Intel *Core i7*) the latter may not be unreasonable as hyper-threading does give a significant extra throughput. What `detectCores(logical = FALSE)` reports is OS-version-dependent: on recent versions of Windows it reports the number of physical cores but on older versions it might report the number of physical CPU packages.

3 Analogues of apply functions

By far the most common direct applications of packages **multicore** and **snow** have been to provide parallelized replacements of `lapply`, `sapply`, `apply` and related functions.

As analogues of `lapply` there are

```
parLapply(cl, x, FUN, ...)
mclapply(X, FUN, ..., mc.cores)
```

where `mclapply` is not available[5] on Windows and has further arguments discussed on its help page. They differ slightly in philosophy: `mclapply` sets up a pool of `mc.cores` workers just for this computation, whereas `parLapply` uses a less ephemeral pool specified by the object `cl` created by a call to `makeCluster` (which *inter alia* specifies the size of the pool). So the workflow is

```
cl <- makeCluster(<size of pool>)
# one or more parLapply calls
stopCluster(cl)
```

For matrices there are the rarely used `parApply` and `parCapply` functions, and the more commonly used `parRapply`, a parallel row `apply` for a matrix.

[4] `parallel::mclapply` detects this and runs nested calls serially.
[5] except as a stub which simply calls `lapply`.

4 SNOW Clusters

The package contains a slightly revised copy of much of **snow**, and the functions it contains can also be used with clusters created by **snow** (provided the package is on the search path).

Two functions are provided to create SNOW clusters, `makePSOCKcluster` (a streamlined version of `snow::makeSOCKcluster`) and (except on Windows) `makeForkCluster`. They differ only in the way they spawn worker processes: `makePSOCKcluster` uses `Rscript` to launch further copies of R (on the same host or optionally elsewhere) whereas `makeForkCluster` forks the workers on the current host (which thus inherit the environment of the current session).

These functions would normally be called *via* `makeCluster`.

Both `stdout()` and `stderr()` of the workers are redirected, by default being discarded but they can be logged using the `outfile` option. Note that the previous sentence refers to the *connections* of those names, not the C-level file handles. Thus properly written R packages using `Rprintf` will have their output redirected, but not direct C-level output.

A default cluster can be registered by a call to `setDefaultCluster()`: this is then used whenever one of the higher-level functions such as `parApply` is called without an explicit cluster. A little care is needed when repeatedly re-using a pool of workers, as their workspaces will accumulate objects from past usage, and packages may get added to the search path.

If clusters are to be created on a host other than the current machine ('`localhost`'), `makeCluster` may need to be be given more information in the shape of extra arguments.

- If the worker machines are not set up in exactly the same way as the master (for example if they are of a different architecture), use `homogeneous = FALSE` and perhaps set `rscript` to the full path to `Rscript` on the workers.
- The worker machines need to know how to communicate with the master: normally this can be done using the hostname found by `Sys.info()`, but on private networks this need not be the case and `master` may need to be supplied as a name or IP address, e.g. `master = "192.168.1.111"`.
- By default `ssh` is used to launch R on the workers. If it is known by some other name, use e.g. `rshcmd = "plink.exe"` for a Windows box using PUTTY. SSH should be set up to use silent authentication: setups which require a password to be supplied may or may not work.
- Socket communication is done over port a randomly chosen port in the range `11000:11999`: site policies might require some other port to be used, in which case set argument `port` or environment variable `R_PARALLEL_PORT`.

5 Forking

Except on Windows, the package contains a copy of **multicore**: there a few names with the added prefix `mc`, e.g. `mccollect` and `mcparallel`. (Package **multicore** used these names, but also the versions without the prefix which are too easily masked: e.g. package **lattice** used to have a function `parallel`.)

The low-level functions from **multicore** are provided but not exported from the namespace.

There are high-level functions `mclapply` and `pvec`: unlike the versions in **multicore** these default to 2 cores, but this can be controlled by setting `options("mc.cores")`, and that takes its default from environment variable `MC_CORES` when the package is loaded. (Setting this to `1`

inhibits parallel operation: there are stub versions of these functions on Windows which force `mc.cores = 1`.)

Functions `mcmapply` and `mcMap` provide analogues of `mapply` and `Map`.

Note the earlier comments about using forking in a GUI environment.

The parent and forked R processes share the per-session temporary directory `tempdir()`, which can be a problem as a lot of code has assumed it is private to the R process. Further, prior to R 2.14.1 it was possibly for `tempfile` in two processes to select the same filename in that temporary directory, and do it sufficiently simultaneously that neither saw it as being in use.

The forked workers share file handles with the master: this means that any output from the worker should go to the same place as '`stdout`' and '`stderr`' of the master process. (This does not work reliably on all OSes: problems have also been noted when forking a session that is processing batch input from '`stdin`'.) Setting argument `mc.silent = TRUE` silences '`stdout`' for the child: '`stderr`' is not affected.

Sharing file handles also impacts graphics devices as forked workers inherit all open graphics devices of the parent: they should not attempt to make use of them.

6 Random-number generation

Some care is needed with parallel computation using (pseudo-)random numbers: the processes/threads which run separate parts of the computation need to run independent (and preferably reproducible) random-number streams. One way to avoid any difficulties is (where possible) to do all the randomization in the master process: this is done where possible in package **boot** (version 1.3-1 and later).

When an R process is started up it takes the random-number seed from the object `.Random.seed` in a saved workspace or constructs one from the clock time and process ID when random-number generation is first used (see the help on `RNG`). Thus worker processes might get the same seed because a workspace containing `.Random.seed` was restored or the random number generator has been used before forking: otherwise these get a non-reproducible seed (but with very high probability a different seed for each worker).

The alternative is to set separate seeds for each worker process in some reproducible way from the seed in the master process. This is generally plenty safe enough, but there have been worries that the random-number streams in the workers might somehow get into step. One approach is to take the seeds a long way apart in the random-number stream: note that random numbers taken a long (fixed) distance apart in a single stream are not necessarily (and often are not) as independent as those taken a short distance apart. Yet another idea (as used by e.g. **JAGS**) is to use different random-number generators for each separate run/process.

Package **parallel** contains an implementation of the ideas of **?**: this uses a single RNG and make *streams* with seeds 2^{127} steps apart in the random number stream (which has period approximately 2^{191}). This is based on the generator of **?**; the reason for choosing that generator[6] is that it has a fairly long period with a small seed (6 integers), and unlike R's default `"Mersenne-Twister"` RNG, it is simple to advance the seed by a fixed number of steps. The

[6] apart from the commonality of authors!

generator is the combination of two:

$$\begin{aligned} x_n &= 1403580 \times x_{n-2} - 810728 \times x_{n-3} \mod (2^{32} - 209) \\ y_n &= 527612 \times y_{n-1} - 1370589 \times y_{n-3} \mod (2^{32} - 22853) \\ z_n &= (x_n - y_n) \mod 4294967087 \\ u_n &= z_n / 4294967088 \text{ unless } z_n = 0 \end{aligned}$$

The 'seed' then consists of $(x_n, x_{n-1}, x_{n-2}, y_n, y_{n-1}, y_{n-2})$, and the recursion for each of x_n and y_n can have pre-computed coefficients for k steps ahead. For $k = 2^{127}$, the seed is advanced by k steps by R call `.Random.seed <- nextRNGStream(.Random.seed)`.

The `?` generator is available *via* `RNGkind("L'Ecuyer-CMRG")`. Thus using the ideas of `?` is as simple as

```
RNGkind("L'Ecuyer-CMRG")
set.seed(<something>)
## start M workers
s <- .Random.seed
for (i in 1:M) {
    s <- nextRNGStream(s)
    # send s to worker i as .Random.seed
}
```

and this is is implemented for SNOW clusters in function `clusterSetRNGStream`, and as part of `mcparallel` and `mclapply` (by default).

Apart from *streams* (2^{127} steps apart), there is the concept of *sub-streams* starting from seeds 2^{76} steps apart. Function `nextRNGSubStream` advances to the next substream.

A direct R interface to the (clunkier) original C implementation is available in CRAN package **rlecuyer** (**?**). That works with named streams, each of which have three 6-element seeds associated with them. This can easily be emulated in R by storing `.Random.seed` at suitable times. There is another interface using S4 classes in package **rstream** (**?**).

7 Load balancing

The introduction mentioned a different strategy which dynamically allocates tasks to workers: this is sometimes known as 'load balancing' and is implemented in `mclapply(mc.preschedule = FALSE)`, `clusterApplyLB` and wrappers.

Load balancing is potentially advantageous when the tasks take quite dissimilar amounts of computation time, or where the nodes are of disparate capabilities. But some *caveats* are in order:

(a) Random number streams are allocated to nodes, so if the tasks involve random numbers they are likely to be non-repeatable (as the allocation of tasks to nodes depends on the workloads of the nodes). It would however take only slightly more work to allocate a stream to each task.

(b) More care is needed is allocating the tasks. If 1000 tasks need to be allocated to 10 nodes, the standard approach send chunks of 100 tasks to each of the nodes. The load-balancing approach sends tasks one at a time to a node, and the communication overhead may be high. So it makes sense to have substantially more tasks than nodes, but not by a factor of 100 (and maybe not by 10).

8 Portability considerations

People wanting to provide parallel facilities in their code need to decide how hard they want to try to be portable and efficient: no approach works optimally on all platforms.

Using `mclapply` is usually the simplest approach, but will run serial versions of the code on Windows. This may suffice where parallel computation is only required for use on a single multi-core Unix-alike server—for `mclapply` can only run on a single shared-memory system. There is fallback to serial use when needed, by setting `mc.cores = 1`.

Using `parLapply` will work everywhere that socket communication works, and can be used, for example, to harness all the CPU cores in a lab of machines that are not otherwise in use. But socket communication may be blocked even when using a single machine and is quite likely to be blocked between machines in a lab. There is not currently any fallback to serial use, nor could there easily be (as the workers start with a different `R` environment from the one current on the master).

An example of providing access to both approaches as well as serial code is package **boot**, version `1.3-3` or later.

9 Extended examples

```
> library(parallel)
```

Probably the most common use of coarse-grained parallelization in statistics is to do multiple simulation runs, for example to do large numbers of bootstrap replicates or several runs of an MCMC simulation. We show an example of each.

Note that some of the examples will only work serially on Windows and some actually are computer-intensive.

9.1 Bootstrapping

Package **boot** (**?**) is support software for the monograph by **?**. Bootstrapping is often used as an example of easy parallelization, and some methods of producing confidence intervals require many thousands of bootstrap samples. As from version `1.3-1` the package itself has parallel support within its main functions, but we illustrate how to use the original (serial) functions in parallel computations.

We consider two examples using the `cd4` dataset from package **boot** where the interest is in the correlation between before and after measurements. The first is a straight simulation, often called a *parametric bootstrap*. The non-parallel form is

```
> library(boot)
> cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
> cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
> cd4.boot <- boot(cd4, corr, R = 999, sim = "parametric",
+                  ran.gen = cd4.rg, mle = cd4.mle)
> boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
+         conf = 0.9, h = atanh, hinv = tanh)
```

To do this with `mclapply` we need to break this into separate runs, and we will illustrate two runs of 500 simulations each:

```
> cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
> cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
> run1 <- function(...) boot(cd4, corr, R = 500, sim = "parametric",
+                            ran.gen = cd4.rg, mle = cd4.mle)
> mc <- 2 # set as appropriate for your hardware
> ## To make this reproducible:
> set.seed(123, "L'Ecuyer")
> cd4.boot <- do.call(c, mclapply(seq_len(mc), run1) )
> boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
+         conf = 0.9, h = atanh, hinv = tanh)
```

There are many ways to program things like this: often the neatest is to encapsulate the computation in a function, so this is the parallel form of

```
> do.call(c, lapply(seq_len(mc), run1))
```

To run this with `parLapply` we could take a similar approach by

```
> 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)
> ## make this reproducible
> clusterSetRNGStream(cl, 123)
> library(boot) # needed for c() method on master
> 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)
```

Note that whereas with `mclapply` all the packages and objects we use are automatically available on the workers, this is not in general[7] the case with the `parLapply` approach. There is often a delicate choice of where to do the computations: for example we could compute `cd4.mle` on the workers (as above) or on the master and send the value to the workers. We illustrate the latter by the following code

```
> cl <- makeCluster(mc)
> cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
> cd4.mle <- list(m = colMeans(cd4), v = var(cd4))
> clusterExport(cl, c("cd4.rg", "cd4.mle"))
> junk <- clusterEvalQ(cl, library(boot)) # discard result
> clusterSetRNGStream(cl, 123)
> res <- clusterEvalQ(cl, boot(cd4, corr, R = 500,
+                     sim = "parametric", ran.gen = cd4.rg, mle = cd4.mle))
> library(boot) # needed for c() method on master
> cd4.boot <- do.call(c, res)
> boot.ci(cd4.boot,  type = c("norm", "basic", "perc"),
+         conf = 0.9, h = atanh, hinv = tanh)
> stopCluster(cl)
```

[7] it is with clusters set up with `makeForkCluster`.

Running the double bootstrap on the same problem is far more computer-intensive. The standard version is

```
> R <- 999; M <- 999 ## we would like at least 999 each
> cd4.nest <- boot(cd4, nested.corr, R=R, stype="w", t0=corr(cd4), M=M)
> ## nested.corr is a function in package boot
> op <- par(pty = "s", xaxs = "i", yaxs = "i")
> qqplot((1:R)/(R+1), cd4.nest$t[, 2], pch = ".", asp = 1,
+         xlab = "nominal", ylab = "estimated")
> abline(a = 0, b = 1, col = "grey")
> abline(h = 0.05, col = "grey")
> abline(h = 0.95, col = "grey")
> par(op)
> nominal <- (1:R)/(R+1)
> actual <- cd4.nest$t[, 2]
> 100*nominal[c(sum(actual <= 0.05), sum(actual < 0.95))]
```

which took about 55 secs on one core of an 8-core Linux server.

Using `mclapply` we could use

```
> mc <- 9
> R <- 999; M <- 999; RR <- floor(R/mc)
> run2 <- function(...)
+     cd4.nest <- boot(cd4, nested.corr, R=RR, stype="w", t0=corr(cd4), M=M)
> cd4.nest <- do.call(c, mclapply(seq_len(mc), run2, mc.cores = mc) )
> nominal <- (1:R)/(R+1)
> actual <- cd4.nest$t[, 2]
> 100*nominal[c(sum(actual <= 0.05), sum(actual < 0.95))]
```

which ran in 11 secs (elapsed) using all of that server.

9.2 MCMC runs

? discusses the maximum-likelihood estimation of the Strauss process, which is done by solving a moment equation

$$E_c T = t$$

where T is the number of R-close pairs and t is the observed value, 30 in the following example. A serial approach to the initial exploration might be

```
> library(spatial)
> towns <- ppinit("towns.dat")
> tget <- function(x, r=3.5) sum(dist(cbind(x$x, x$y)) < r)
> t0 <- tget(towns)
> R <- 1000
> c <- seq(0, 1, 0.1)
> ## res[1] = 0
> res <- c(0, sapply(c[-1], function(c)
+     mean(replicate(R, tget(Strauss(69, c=c, r=3.5))))))
> plot(c, res, type="l", ylab="E t")
> abline(h=t0, col="grey")
```

which takes about 20 seconds today, but many hours when first done in 1985. A parallel version might be

```
> run3 <- function(c) {
+     library(spatial)
+     towns <- ppinit("towns.dat") # has side effects
+     mean(replicate(R, tget(Strauss(69, c=c, r=3.5))))
+ }
> cl <- makeCluster(10, methods = FALSE)
> clusterExport(cl, c("R", "towns", "tget"))
> res <- c(0, parSapply(cl, c[-1], run3)) # 10 tasks
> stopCluster(cl)
```

which took about 4.5 secs, plus 2 secs to set up the cluster. Using a fork cluster (not on Windows) makes the startup much faster and setup easier:

```
> cl <- makeForkCluster(10)  # fork after the variables have been set up
> run4 <- function(c)  mean(replicate(R, tget(Strauss(69, c=c, r=3.5))))
> res <- c(0, parSapply(cl, c[-1], run4))
> stopCluster(cl)
```

As one might expect, the `mclapply` version is slightly simpler:

```
> run4 <- function(c)  mean(replicate(R, tget(Strauss(69, c=c, r=3.5))))
> res <- c(0, unlist(mclapply(c[-1], run4, mc.cores = 10)))
```

If you do not have as many as 10 cores, you might want to consider load-balancing in a task like this as the time taken per simulation does vary with `c`. This can be done using `mclapply(mc.preschedule = FALSE)` or `parSapplyLB`. The disadvantage is that the results would not be reproducible (which does not matter here).

9.3 Package installation

With over 4000 R packages available, it is often helpful to do a comprehensive install in parallel. We provide facilities to do so in `install.packages` *via* a parallel `make`, but that approach may not be suitable.[8] Some of the tasks take many times longer than others, and we do not know in advance which these are.

We illustrate an approach using package **parallel** which is used on part of the CRAN check farm. Suppose that there is a function `do_one(pkg)` which installs a single package and then returns. Then the task is to run `do_one` on as many of the `M` workers as possible whilst ensuring that all of the direct and indirect dependencies of `pkg` are installed before `pkg` itself. As the installation of a single package can block several others, we do need to allow the number of installs running simultaneously to vary: the following code achieves that, but needs to use low-level functions to do so.

```
> pkgs <- "<names of packages to be installed>"
> M <- 20 # number of parallel installs
> M <- min(M, length(pkgs))
> library(parallel)
> unlink("install_log")
> cl <- makeCluster(M, outfile = "install_log")
> clusterExport(cl, c("tars", "fakes", "gcc")) # variables needed by do_one
> ## set up available via a call to available.packages() for
> ## repositories containing all the packages involved and all their
> ## dependencies.
```

[8] A parallel `make` might not be available, and we have seen a couple of instances of package installation not working correctly when run from `make`.

```
> DL <- utils:::.make_dependency_list(pkgs, available, recursive = TRUE)
> DL <- lapply(DL, function(x) x[x %in% pkgs])
> lens <- sapply(DL, length)
> ready <- names(DL[lens == 0L])
> done <- character() # packages already installed
> n <- length(ready)
> submit <- function(node, pkg)
+     parallel:::sendCall(cl[[node]], do_one, list(pkg), tag = pkg)
> for (i in 1:min(n, M)) submit(i, ready[i])
> DL <- DL[!names(DL) %in% ready[1:min(n, M)]]
> av <- if(n < M) (n+1L):M else integer() # available workers
> while(length(done) < length(pkgs)) {
+     d <- parallel:::recvOneResult(cl)
+     av <- c(av, d$node)
+     done <- c(done, d$tag)
+     OK <- unlist(lapply(DL, function(x) all(x %in% done) ))
+     if (!any(OK)) next
+     p <- names(DL)[OK]
+     m <- min(length(p), length(av)) # >= 1
+     for (i in 1:m) submit(av[i], p[i])
+     av <- av[-(1:m)]
+     DL <- DL[!names(DL) %in% p[1:m]]
+ }
```

9.4 Passing ...

The semantics of `...` do not fit well with parallel operation, since lazy evaluation may be delayed until the tasks have been sent to the workers. This is no problem in the forking approach, as the information needed for lazy evaluation will be present in the forked workers.

For **snow**-like clusters the trick is to ensure that `...` has any promises forced whilst the information is still available. This is how **boot** does it:

```
>     fn <- function(r) statistic(data, i[r, ], ...)
>     RR <- sum(R)
>     res <- if (ncpus > 1L && (have_mc || have_snow)) {
+         if (have_mc) {
+             parallel::mclapply(seq_len(RR), fn, mc.cores = ncpus)
+         } else if (have_snow) {
+             list(...) # evaluate any promises
+             if (is.null(cl)) {
+                 cl <- parallel::makePSOCKcluster(rep("localhost", ncpus))
+                 if(RNGkind()[1L] == "L'Ecuyer-CMRG")
+                     parallel::clusterSetRNGStream(cl)
+                 res <- parallel::parLapply(cl, seq_len(RR), fn)
+                 parallel::stopCluster(cl)
+                 res
+             } else parallel::parLapply(cl, seq_len(RR), fn)
+         }
+     } else lapply(seq_len(RR), fn)
```

Note that `...` is an argument to `boot`, and so after

```
>             list(...) # evaluate any promises
```

it refers to objects within the evaluation frame of `boot` and hence the environment of `fn` which will therefore be sent to the workers along with `fn`.

10 Differences from earlier versions

The support of parallel RNG differs from **snow**: **multicore** had no support.

10.1 Differences from multicore

multicore had quite elaborate code to inhibit the Aqua event loop for `R.app` and the event loop for the `quartz` graphics device. This has been replaced by recording a flag in the R executable for a child process.

The version of `detectCores` here is biased towards the number of physical CPUs. This was occasioned by serious problems on Sparc Solaris where **multicore** defaulted to a silly number of processes.

Functions `fork` and `kill` have `mc` prefixes and are not exported. This avoids confusion with other packages (such as package **fork**), and note that `mckill` is not as general as `tools::pskill`.

Aliased functions `collect` and `parallel` are no longer provided.

10.2 Differences from snow

snow set a timeout that exceeded the maximum value required by POSIX, and did not provide a means to set the timeout on the workers. This resulted in process interlocks on Solaris.

`makeCluster` creates MPI or NWS clusters by calling **snow**.

`makePSOCKcluster` has been streamlined, as package **parallel** is in a known location on all systems, and `Rscript` is these days always available. Logging of workers is set up to append to the file, so multiple processes can be logged.

`parSapply` has been brought into line with `sapply`.

`clusterMap()` gains `SIMPLIFY` and `USE.NAMES` arguments to make it a parallel version of `mapply` and `Map`.

The timing interface has not been copied.