Package 'parallel'

R-core

August 16, 2011

1 Introduction

A package **parallel** is being developed for release in R 2.14.0. It builds on the work done for CRAN packages **multicore** (Urbanek, 2009–present) and **snow** (Tierney *et al.*, 2003–present).

Until October 2011 anything in this package is strictly experimental and may be changed or removed without notice.

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 in parallel (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.

This package is about 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 important 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 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 `parLapply` and `mclapply` 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: this is done in `mclapply(mc.preschedule = FALSE)` and `clusterApplyLB`.

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

1. *Via* forking. *Fork* is a concept[1] 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 usually share 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.

   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 communication *via* sockets is supported in package **parallel**.

2. *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.

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 PVM ('parallel virtual machine') and MPI ('message passing interface') using R packages **rpvm** and **Rmpi** respectively. Communication overheads can dominate computation times in this approach, so it is most often used on tightly-coupled clusters of computers with high-speed interconnects.

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

   It will not be considered further in this vignette.

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, 12 or more are commonplace. It is such hardware that package **parallel** is designed to exploit.

2 Numbers of CPUs

In setting up parallel computations it can be helpful to have some idea of the number of CPUs available, but this is a rather slippery concept.

Note that all a program can possibly determine is the total number of CPUs 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 by using multiple threads through multi-threaded BLAS, compiled code using OpenMP or other low-level forms of parallelism. We

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

have even seen instances of **multicore**'s `mclapply` being called recursively, generating $2n + n^2$ processes on a machine estimated to have $n = 16$ CPUs.

But in so far as it is a helpful guideline, function `detectCores()` try 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. One OS where this is not the case is Windows, where only the number of logical CPUs is readily available. On modern hardware (e.g. Intel *Core i7*) the latter may not be unreasonable as hyper-threading does give a significant extra throughput: on what are now rather old Pentium 4's it will simply over-estimate by a factor of two. Note that we said 'available': in most cases when virtual machines are in use what is reported is the maximum number of virtual CPUs which can be used simultaneously by that VM.

On Solaris, we choose (unlike **multicore**) to give the number of physical CPUs. Modern Sparc CPUs have 8-way 'Chip Multi-Threading': this enables the CPU to switch rapidly between threads if there are cache misses but does not run threads in parallel.

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 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 SNOW Clusters

Currently the package contains a copy of much of **snow**, and the functions it contains can also be used with clusters created by **snow**.

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 (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.

5 Forking

Except on Windows, the package currently contains a copy of **multicore**: there a few names with the added prefix `mc`, e.g. `mccollect`, `mcexit`, `mcfork`, `mckill` and `mcparallel`. (For back-compatibility, functions `parallel` and `collect` are also exported: these names are easily masked.)

Most of the functions provided are not intended to be used by end users, and they may not in future be exported from the namespace.

There are copies of the higher-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.

One unexported function which might be useful is `parallel:::isChild`, which can be used to prevent `mclapply` being called recursively.

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 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 (see the help on `RNG`: as from R 2.14.0 it also uses the process ID). Thus worker processes might get the same seed, either because a workspace was restored or (in R $<$ 2.14.0) because the workers were started at the same time: otherwise these get a non-reproducible seed.

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 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 L'Ecuyer *et al.* (2002): this uses a single RNG and takes seeds 2^{127} apart in the random number stream (which has period approximately 2^{191}). This is based on the generator of L'Ecuyer (1999); the reason for choosing that generator[2] 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.

[2] apart from the commonality of authors!

The generator is the combination of two:

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

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 L'Ecuyer (1999) generator is available in R as from version 2.14.0 *via* `RNGkind("L'Ecuyer-CMRG")`. Thus using the ideas of L'Ecuyer *et al.* (2002) 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`.

Apart from *streams* (2^{127} apart), there is the concept of *sub-streams* starting from seeds 2^{76} steps apart.

A direct R interface to the (clunkier) original C implementation is available in CRAN package **rlecuyer** (Sevcikova and Rossini, 2004–present). That works with named streams, each of which have three 6-elements seeds associated with them, the original seed set for the package, the initial seed set for the stream and the current seed for the stream. This can easily be emulated in R by storing `.Random.seed` at suitable times. There is another interface using S4 classes in package **rstream** (Leydold, 2005–present).

7 Extended examples

Probably the most common use of coarse-grained paralllelization 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 are not available on Windows and some actually are computer-intensive.

7.1 Bootstrapping

Package **boot** (Canty and Ripley, 1999–present) is support software for the monograph by Davison and Hinkley (1997). 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 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 = mean(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:

```
> library(boot)
> cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
> cd4.mle <- list(m = mean(cd4), v = var(cd4))
> mc <- 2
> run1 <- function(...) boot(cd4, corr, R = 500, sim = "parametric",
+                            ran.gen = cd4.rg, mle = cd4.mle)
> library(parallel)
> ## To make this reproducible:
> set.seed(123, "L'Ecuyer"); mc.reset.stream()
> 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 on 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 = mean(cd4), v = var(cd4))
+    boot(cd4, corr, R = 500, sim = "parametric",
+         ran.gen = cd4.rg, mle = cd4.mle)
+ }
> library(parallel)
> mc <- 2
> 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 set in the workers, this is not in general[3] the case with the `parLapply` approach. There is often a

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

delicate choice of where to do the computations: for example we could compute `cd4.mle` on the master and send the value to the workers, or (as here) compute it on the workers. We illustrate that by the following code

```
> library(parallel)
> mc <- 2
> cl <- makeCluster(mc)
> cd4.rg <- function(data, mle) MASS::mvrnorm(nrow(data), mle$m, mle$v)
> cd4.mle <- list(m = mean(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)
```

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

```
> library(boot)
> 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)
> op <- par(pty = "s", xaxs = "i", yaxs = "i")
> qqplot((1:R)/(R+1), cd4.nest$t[, 2], pch = ".", asp = 1,
+         xlab = "nominal", ylab = "estimated")
> abline(a = 0, b = 1, col = "grey")
> par(op)
> abline(h=0.05, col = "grey")
> abline(h=0.95, col = "grey")
> nominal <- (1:R)/(R+1)
> actual <- cd4.nest$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

```
> library(parallel)
> library(boot)
> 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 = 9) )
> 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).

7.2 MCMC runs

Ripley (1988) discusses the maximum-likelihood estimation of the Straus 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 several 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))
> 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)  # after the variables have been setup
> run4 <- function(c)  mean(replicate(R, tget(Strauss(69, c=c, r=3.5))))
> res <- c(0, parSapply(cl, c[-1], run3))
> 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], run3, mc.cores = 10)))
```

References

Canty A, Ripley B (1999–present). "boot: Bootstrap Functions." URL http://cran.r-project.org/web/packages/boot/index.html.

Davison AC, Hinkley DV (1997). *Bootstrap Methods and Their Application*. Cambridge University Press, Cambridge.

L'Ecuyer P (1999). "Good parameters and implementations for combined multiple recursive random number generators." *Operations Research*, **47**, 195–164. URL http://www.iro.umontreal.ca/~lecuyer/myftp/papers/combmrg2.ps.

L'Ecuyer P, Simard R, Chen EJ, Kelton WD (2002). "An object-oriented random-number package with many long streams and substreams." *Operations Research*, **50**, 1073–5. URL http://www.iro.umontreal.ca/~lecuyer/myftp/papers/streams00.pdf.

Leydold J (2005–present). "`rstream`: Streams of random numbers." URL http://cran.r-project.org/web/packages/rstream/index.html.

Ripley BD (1988). *Statistical Inference for Spatial Processes*. Cambridge University Press, Cambridge.

Sevcikova H, Rossini T (2004–present). "`rlecuyer`: R interface to RNG with multiple streams." URL http://cran.r-project.org/web/packages/rlecuyer/index.html.

Tierney L, Rossini AJ, Li N, Sevcikova H (2003–present). "Simple Network of WorkStations for R." URL http://www.stat.uiowa.edu/~luke/R/cluster/cluster.html.

Urbanek S (2009–present). "`multicore`: Parallel processing of R code on machines with multiple cores or CPUs." URL http://cran.r-project.org/web/packages/multicore/index.html.