The `multicore` package

Script

Manuel J. A. Eugster*

"Parallel Computing with R" Tutorial,
Statistical Computing 2009

Introduction

The package `multicore` is available from CRAN (version 0.1-3) and from `http://www.rforge.de/multicore` (version 0.1-4). The latest version has (very) experimental Microsoft Windows support, therefore it is suggested to use `multicore` on a Unix platform.

```
> library(multicore)
> multicore:::detectCores()
```

```
[1] 4
```

The computer in this script has four CPUs/Cores. Even though your computer has only one CPU/Core you can use `multicore` "to play with it"; but there is no speed up (but rather a slow down) because the spawned child processes have to share the same CPU/Core. In both cases, set the `cores` option to the preferred number of cores to use (if you want to use all available cores leave this option):

```
> options(cores = 2)
> getOption('cores')
```

```
[1] 2
```

Example: 2-dimensional sinc function

We want to calculate the 2-dimensional sinc function on a grid between $[-10, 10]$ consisting of 500 points:

```
> sinc <- function(x) {
+   r <- sqrt(x[1]^2 + x[2]^2)
+   10 * sin(r) / r
+ }
> x <- seq(-10, 10, length.out=500)
> g <- expand.grid(x=x, y=x)
```

The **sequential calculation** simply calls the `sinc()` function for each row of the grid:

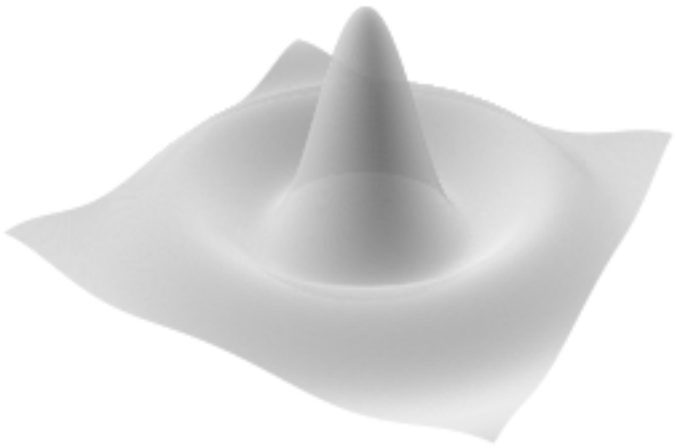
*Insitut für Statistik, Ludwig-Maximiliams-Universität München, `Manuel.Eugster@stat.uni-muenchen.de`

```
> system.time(z <<- apply(g, 1, sinc))

   user  system elapsed
 10.448   0.135  11.352
```

The calculation needs about 8 to 10 seconds. To assure the correctness of the calculation we plot the function:

```
> dim(z) <- c(length(x), length(x))
> persp(x, x, z, theta=30, phi=30, expand=0.5, col='white',
+       border=NA, shade=0.3, box=FALSE)
```



A strategy for a **parallel computation** of the sinc function is to split the grid into subgrids and calculate the function for these subgrids in parallel. Here two CPUs/Cores are given, therefore the grid is split into two parts along $y = 0$:

```
> glist <- split(g, g['y'] > 0)
```

Now, the `apply(...)` call from the sequential calculation is executed for each element of the list. A sequential execution is done using `lapply()`:

```
> system.time(zlist <<- lapply(glist, apply, 1, sinc))

   user  system elapsed
  8.302   0.091   8.620
```

The parallel execution is done using the parallel analogon, `mclapply()`:

```
> system.time(zlist <<- mclapply(glist, apply, 1, sinc))
```

```
   user  system elapsed
  4.316   4.266   5.390
```

The parallel execution needs only half of the time. In background the `mclapply()` has forked two child processes and each of the them has calculated the result for one list element:

```
> par(mfrow=c(1,2))
> # First half:
> x1 <- x[x > 0]
> z1 <- zlist[[1]]
> dim(z1) <- c(length(x), length(x1))
> persp(x, x1, z1,  theta=90, phi=30, expand=0.5, col='white',
+       border=NA, shade=0.3, box=FALSE)
> # Second half:
> x2 <- x[x <= 0]
> z2 <- zlist[[2]]
> dim(z2) <- c(length(x), length(x2))
> persp(x, x2, z2,  theta=90, phi=30, expand=0.5, col='white',
+       border=NA, shade=0.3, box=FALSE)
```

For the final result the elements of the list must be put together:

```
> z <- do.call('c', zlist)
```

During the execution the two child R processes can be watched in the processes list of the operating system (e.g., Task manager on Windows, Activity monitor on Mac OS, and `top` on a Unix/Linux systems).

Example: Sudoku

Imagine you are working in a "big" R session with lots of packages loaded and variables defined. You have to do a longer calculation but also some shorter things which can be done in parallel. With `multicore` you can just fork the current R, start the long calculation in the cild process, continue working in the parent process and collect the result by the time the calculation has finished.

As example a 9×9 Sudoku with 70 blank cells is solved in a parallel child process:

```
> library(sudoku)
> set.seed(1234)

> s <- generateSudoku(70)
> s

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    0    0    4    0    0    6    0    0    0
 [2,]    0    0    0    0    0    0    0    0    0
 [3,]    0    6    0    0    0    0    0    1    0
 [4,]    0    0    6    0    0    8    0    0    1
 [5,]    0    0    0    0    2    0    0    0    0
 [6,]    4    0    0    0    0    0    0    0    0
 [7,]    0    0    0    0    0    0    0    0    0
 [8,]    0    0    0    0    0    0    0    0    0
 [9,]    0    0    0    0    0    0    0    8    3
```

The Sudoku is solved using `solveSudoku(s)`; the expression is started in a parallel process using the `parallel()` function:

```
> p <- parallel(solveSudoku(s, print.it=FALSE))
> p

 parallelJob: processID=778
```

`p` "represents" the parallel job and contains the job ID. The Sudoku is now solved in a child process and one can continue working in the parent process. `collect()` then collects the results from one or more child processes:

```
> collect(p, wait=TRUE)

$`778`
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    1    9    4    8    5    6    2    3    7
 [2,]    7    8    2    3    9    1    5    4    6
 [3,]    3    6    5    7    4    2    8    1    9
 [4,]    2    5    6    4    3    8    7    9    1
 [5,]    8    1    3    9    2    7    6    5    4
 [6,]    4    7    9    1    6    5    3    2    8
 [7,]    6    3    8    2    1    9    4    7    5
 [8,]    9    4    7    5    8    3    1    6    2
 [9,]    5    2    1    6    7    4    9    8    3
```

It returns a list with an element for each process. If `wait = TRUE` the function waits until all processes are finished. Otherwise it checks for any results that are available within `timeout` seconds from now.

Random numbers and job scheduling

Working with random numbers can be tricky in parallel environments. `multicore` provides the `mc.seet.seed` argument for the functions `mclapply()` and `parallel()`. If set to `TRUE` then each parallel process first sets

its seed to something different from other processes; otherwise all processes start with the same (namely current) seed.

As example, two child processes draw five numbers from $[1, 10000]$. First, the seed is set in the parent process and `mc.set.seed` is set to `FALSE`:

```
> set.seed(1234)
> mclapply(rep(10000, 2), sample, 5, mc.set.seed=FALSE)

[[1]]
[1] 1138 6223 6092 6232 8606

[[2]]
[1] 1138 6223 6092 6232 8606
```

The drawn numbers are exactly the same. Otherwise, with `mc.set.seed` set to `TRUE` each child sets its seed to its process ID and therefore the drawn numbers are different:

```
> set.seed(1234)
> mclapply(rep(10000, 2), sample, 5, mc.set.seed=TRUE)

[[1]]
[1] 6335 3534 7387 9597 3252

[[2]]
[1] 2546 2029 3773 9663 5026
```

Note: In `multicore` version 0.1-4 the `mc.set.seed` default values of `mclapply()` and `parallel()` are different.

`mclapply()` provides two kinds of job scheduling for the passed list. If `mc.preschedule` is set to `TRUE` then the computation is first divided to (at most) as many jobs as there are cores and then the jobs are started; each job possibly covering more than one value. If `mc.preschedule` is set to `FALSE` then one job is spawned for each value of the list sequentially.

For demonstration we start three jobs (remember there are two CPUs/Cores available) and just return the process ID of the corresponding child process:

```
> xpid <- function(x) c(x=x, pid=Sys.getpid())
```

In case of `mc.preschedule` set to `TRUE` only two process ID numbers appears alternately. Job 1 and 3 are executed in the same child:

```
> mclapply(1:3, xpid, mc.preschedule=TRUE)

[[1]]
  x pid
  1 783

[[2]]
  x pid
  2 784

[[3]]
```

```
  x pid
  3 783
```

In case of `mc.preschedule` set to `FALSE` each job is executed in a separate child process:

```
> mclapply(1:3, xpid, mc.preschedule=FALSE)

[[1]]
  x pid
  1 785

[[2]]
  x pid
  2 786

[[3]]
  x pid
  3 787
```

The first kind of job scheduling is better for short computations or large number of values in the passed list, the second kind of job scheduling is better for jobs that have high variance of completion time and not too many values in the passed list.

Note: If more jobs than available CPUs/Cores are forked than `mc.set.seed = FALSE` and `mc.preschedule = FALSE` provides identical random numbers for all jobs.

References

[1] Simon Urbanek (2009). `multicore`: Parallel processing of R code on machines with multiple cores or CPUs. R package version 0.1-4. `http://www.rforge.net/multicore/`.

[2] David Brahm, Greg Snow, with contributions from Curt Seeliger and Henrik Bengtsson (2009). `sudoku`: Sudoku Puzzle Generator and Solver. R package version 2.2. `http://cran.r-project.org/package=sudoku`.