Distributed computations with the NMOF package

Enrico Schumann
es@enricoschumann.net

2014-10-23

Contents

1 Overview

This manuscript contains several examples of distributed computations – more specifically, using several cores on one machine – in the NMOF package (Gilli, Maringer, and Schumann, 2011). Such computations are supported in the functions `bracketing`, `GAopt`, `gridSearch`, `restartOpt` through package `parallel`. (Before, packages `multicore` Urbanek (2011) and `snow` Tierney et al. (2011) were supported.) This report mainly serves as a convenient collection of test cases for these functions. I am grateful for comments and corrections.

The latest version of the package is available from
`http://enricoschumann.net/R/packages/NMOF/index.htm`;
the package is also available from CRAN. To install the package from within R, type

```
> install.packages("NMOF") ## CRAN
> install.packages("NMOF", repos = "http://enricoschumann.net/R")
```

to download and install it. For all examples to follow, the package needs to be attached.

```
> require("NMOF")
> set.seed(1122344)
> nC <- 4L ## the number of cores to be used
```

The distributed computations rely on package parallel. We will use package rbenchmark (Kusnierczyk, 2010) to measure the time that particular computations need.

```
> require("rbenchmark")
> require("parallel")
```

This report is written with Sweave (Leisch, 2002). The code is part of the package; it can be found in the subdirectory `NMOFex`. To show the code in R, you can use the function `system.file`.

```
> whereToLook <- system.file("NMOFex/NMOFdist.R", package = "NMOF")
> file.show(whereToLook, title = "NMOF examples")
```

The latest pdf version of this report can be obtained from
`http://enricoschumann.net/NMOF.htm`

2 Simple tests

2.1 Slowing things down

We first run a simple test to see if a specific function is expensive enough for distributed computation.

```
> testFun <- function(ignore, delay) {
     Sys.sleep(delay)
     1
 }
> delay <- 0.05     ## running time of function
> n <- 8            ## how many calls per lapply
> repl <- 10        ## how many restarts
> sq <- seq_len(n)
```

Note that we set up the cluster for `clusterApply` before we measure the computing time.

```
> cl <- makeCluster(c(rep("localhost", nC)), type = "SOCK")
> benchmark(lapply(sq, testFun, delay),                ## serial
           mclapply(sq, testFun, delay),              ## formerly 'multicore'
           clusterApply(cl, sq, testFun, delay),      ## formerly 'snow'
           columns = c("test", "elapsed", "relative"),
           order = "relative", replications = repl)
```

```
                                   test elapsed relative
3 clusterApply(cl, sq, testFun, delay)    1.76     1.00
2         mclapply(sq, testFun, delay)    2.10     1.20
1           lapply(sq, testFun, delay)    4.01     2.28
```

```
> stopCluster(cl)
```

2.2 Copying arguments

With `clusterApply`, it can be beneficial to copy constant function arguments to the nodes. Suppose we have a Least-Squares objective function.

```
> OF <- function(b, X, y) {
     temp <- X %*% b - y
     sum(temp^2)
 }
```

For this function, `b` are the parameters to be optimised, but `X` and `y` stay fixed.

We run an experiment. We start by creating random `X`, `y` and `b`.

```
> ncol <- 10; nrow <- 200
> X <- array(rnorm(nrow * ncol), dim = c(nrow, ncol))
> y <- rnorm(nrow)
> b <- rnorm(ncol)
```

Just a test.

```
> OF(b, X, y)
```

```
[1] 2032
```

Suppose we used such a function in a multiple-solution heuristic such as DE. We assume the population `lP` has size `n`.

```
> n <- 50            ## how many calls per lapply
> sq <- seq_len(n)
> lP <- vector("list", length = n)
> for (i in sq)
     lP[[i]] <- b
```

Next we define various expressions.

```
> snow_with_copying <- expression({
     ignore1 <- clusterApply(cl, lP, OF, X, y)
 })
> lapply <- expression({
     ignore2 <- lapply(lP, OF, X, y)
 })
```

```
> snow_without_copying <- expression({
     OF1 <- function(b) {
         temp <- X %*% b - y
         sum(temp^2)
     }
     ignore3 <- clusterApply(cl, lP, OF1)
 })
> cl <- makeCluster(rep("localhost", nC), type = "SOCK")
> clusterExport(cl, list("X", "y"))
> benchmark(lapply,
            snow_with_copying,
            snow_without_copying,
            columns = c("test", "elapsed", "relative"),
            order = "relative", replications = 50)
```

```
                  test elapsed relative
1               lapply   0.032      1.0
2    snow_with_copying   0.624     19.5
3 snow_without_copying  26.252    820.4
```

```
> stopCluster(cl)
```

Both parallel computations are slow, as the objective function is not expensive enough. But parallel without copying is faster, nevertheless.

We compare the results.

```
> all.equal(ignore1, ignore2)
```

```
[1] TRUE
```

```
> all.equal(ignore2, ignore3)
```

```
[1] TRUE
```

3 bracketing

We repeat the example from Gilli et al. (2011, p. 290); again we add a small delay to the function.

```
> testFun <- function(x) {
     Sys.sleep(0.1)
     cos(1/x^2)
 }
> with_loop <- expression(
     sol1 <- bracketing(testFun,
                        interval = c(0.3, 0.9),
                        n = 100L))
```

```
> with_multicore <- expression(
     sol2 <- bracketing(testFun,
                        interval = c(0.3, 0.9),
                        n = 100L,
                        method = "multicore",
                        mc.control = list(mc.cores = nC)))
> with_snow  <- expression(
     sol3 <- bracketing(testFun,
                        interval = c(0.3, 0.9),
                        n = 100L, method = "snow", cl = nC))
> benchmark(with_loop,
           with_multicore,
           with_snow,
           columns = c("test", "elapsed", "relative"),
           order = "relative", replications = 1)
```

```
            test elapsed relative
2 with_multicore    2.51     1.00
3      with_snow    3.29     1.31
1      with_loop   10.02     3.99
```

We check.

```
> all.equal(sol1, sol2)
```

```
[1] TRUE
```

```
> all.equal(sol1, sol3)
```

```
[1] TRUE
```

4 Multiple-solution (a.k.a. population-based) optimisation methods

4.1 Evaluting several solutions – a prototype

We create a population P of solution vectors, shaped as a matrix in which every column is one solution.

```
> ncol <- 20
> nrow <- 1000
> P <- array(rnorm(nrow * ncol), dim = c(nrow, ncol))
```

As an example, we define a simple objective function, which partially sorts a vector x. Such a function could be used, for instance, in a Value-at-Risk calculation, or for some kind of robust statistic.

```
> fun <- function (x, h)
     sort(x, partial = h)[h]
```

We check the function on the first column of P.

```
> h <- 5L
> fun(P[ ,1L], h)
```

```
[1] -2.73
```

The most natural way to evaluate all solutions is to loop over the columns of P, for which we create a new function loopfun. The function takes as arguments a matrix x and a function f, to be applied to the columns of x. Further arguments to f are passed through ....

```
> loopfun <- function(x, f, ...) {
     ns <- ncol(x)
     fv <- numeric(ns)
     for (i in seq_len(ns))
         fv[i] <- f(x[ ,i], ...)
     fv
 }
```

With this function, we can evaluate the whole population, not just a single column.

```
> loopresult <- loopfun(P, fun, h)
```

Now, how to exploit the fact that we can evaluate the columns of P in any order? That is, how to distribute the computations? The simplest way is to call a member of the apply family, of the sort that comes with packages parallel; we will use clusterApply here.

First, we create a list from the columns of P.

```
> mat2list <- function(x) {
     nx <- ncol(x)
     listP <- vector(mode = "list", length = nx)
     for (s in seq_len(nx))
         listP[[s]] <- P[ ,s]
     listP
 }
> listP <- mat2list(P)
```

This could more have been written more compactly as split(P, col(P)); but the loop version is faster. So, we can compare the results of loopfun with a distributed computation.

```
> cl <- makeCluster(c(rep("localhost", nC)), type = "SOCK")
> snowresult <- unlist(clusterApply(cl, listP, fun, h))
> stopCluster(cl)
> all.equal(loopresult, snowresult)
```

```
[1] TRUE
```

We can check the running time. We set up the cluster outside the test; we also leave out the call to unlist.

```
> cl <- makeCluster(c(rep("localhost", nC)), type = "SOCK")
> benchmark(clusterApply(cl, listP, fun, h),
           loopfun(P, fun, h),
           columns = c("test", "elapsed", "relative"),
           order = "relative", replications = 100)
```

```
                             test elapsed relative
2              loopfun(P, fun, h)   0.134        1
1 clusterApply(cl, listP, fun, h)  20.003      149
```

```
> stopCluster(cl)
```

Using `clusterApply` is much slower; so, apparently, `fun` is far too cheap to benefit from distribution. But how about `loopfun`? That is, why not split the matrix `P` into smaller submatrices, and then let the nodes loop over these submatrices.

We make our matrix larger (more columns).

```
> ncol <- 100
> nrow <- 1000
> P <- array(rnorm(nrow * ncol), dim = c(nrow, ncol))
> system.time(for (i in seq_len(10000L)) fun(P[ ,1L], 10L))
```

```
   user  system elapsed
  0.580   0.000   0.559
```

The number of columns per core.

```
> d <- round(ncol/nC) ## nC is the number of cores
```

We split our matrix into `d` parts.

```
> listP <- vector(mode = "list", length = nC)
> for (s in seq_len(nC))
     listP[[s]] <- P[ ,(d*s-d+1):min(ncol, d*s)]
```

Finally, we run a test.

```
> cl <- makeCluster(c(rep("localhost", nC)), type = "SOCK")
> benchmark(parallel.result <- clusterApply(cl, listP, loopfun, fun, h),
           loop.result <- loopfun(P, fun, h),
           columns = c("test", "elapsed", "relative"),
           order = "relative", replications = 100)
```

```
                                                          test
2                          loop.result <- loopfun(P, fun, h)
1 parallel.result <- clusterApply(cl, listP, loopfun, fun, h)
  elapsed relative
2   0.665     1.00
1   4.177     6.28
```

```
> stopCluster(cl)
> all.equal(loop.result, unlist(parallel.result))
```

```
[1] TRUE
```

Thus, the distributed computation is still slower than the simple loop, but we have improved massively compared with the first approach of sending single solutions to the nodes.

4.2 GAopt

GAopt currently supports distributed evaluation of the objective function (in future versions, the repair and penalty functions may also be distributed).

We use the matching example from ?GAopt,

```
> OF <- function(x, y) {
     Sys.sleep(0.001)
     sum(x != y)
 }
> size <- 20L            ## the length of the string
> y <- runif(size) > 0.5 ## the true solution
> with_loop <- list(nB = size, nP = 200L, nG = 50L, prob = 0.002,
                    printBar = FALSE, printDetail = FALSE,
                    methodOF = "loop")
> with_snow <- list(nB = size, nP = 200L, nG = 50L, prob = 0.002,
                    printBar = FALSE, printDetail = FALSE,
                    methodOF = "snow", cl = nC)
> with_multicore <- list(nB = size, nP = 200L, nG = 50L, prob = 0.002,
                         printBar = FALSE, printDetail = FALSE,
                         methodOF = "multicore")
> benchmark(GAopt(OF, algo = with_loop, y = y),
           GAopt (OF, algo = with_snow, y = y),
           GAopt(OF, algo = with_multicore, y = y),
           columns = c("test", "elapsed", "relative"),
           order = "relative", replications = 1)
```

```
                                     test elapsed relative
3 GAopt(OF, algo = with_multicore, y = y)    6.44     1.00
1      GAopt(OF, algo = with_loop, y = y)   11.90     1.85
2      GAopt(OF, algo = with_snow, y = y)  107.90    16.77
```

```
>
```

To pass optional arguments to mclapply, we need to collect them in a list mc.control, which needs to be added to algo. As an example, we instruct parallel to use just one core; thus, we should see no speedup.

```
> with_multicore$mc.control <- list(mc.cores = 1L)
> ## system.time(GAopt(OF, algo = with_multicore, y = y))
> benchmark(GAopt(OF, algo = with_loop, y = y),
            GAopt(OF, algo = with_multicore, y = y),
            columns = c("test", "elapsed", "relative"),
            order = "relative", replications = 1)
```

```
                                     test elapsed relative
2 GAopt(OF, algo = with_multicore, y = y)    11.7     1.00
1      GAopt(OF, algo = with_loop, y = y)    11.9     1.02
```

A few more tests.

```
> OF <- function(x, y) {
     Sys.sleep(0.01)
     sum(x != y)
 }
> size <- 10L; y <- runif(size) > 0.5
> algo <- list(nB = size, nP = 20L, nG = 100L, prob = 0.002,
               printBar = FALSE, methodOF = "loop")
> t1 <- system.time(sol <- GAopt(OF, algo = algo, y = y))
```

```
Genetic Algorithm.
Best solution has objective function value 0 ;
standard deviation of OF in final population is 0 .
```

```
> all.equal(sol$xbest, y)
```

```
[1] TRUE
```

```
> all.equal(sol$OFvalue, 0)
```

```
[1] TRUE
```

```
> algo <- list(nB = size, nP = 20L, nG = 100L, prob = 0.002,
               printBar = FALSE, methodOF = "snow", cl = nC)
> t2 <- system.time(sol <- GAopt(OF, algo = algo, y = y))
```

```
Genetic Algorithm.
Best solution has objective function value 0 ;
standard deviation of OF in final population is 0 .
```

```
> all.equal(sol$xbest, y)
```

```
[1] TRUE
```

```
> all.equal(sol$OFvalue, 0)
```

```
[1] TRUE
```

This allows us to check the speedup (but from only one replication).

```
> round(t1[[3L]]/t2[[3L]],1)
```

```
[1] 0.8
```

We can also pass further parameters to the objective function.

```
> OF <- function(x, y, k) {
     Sys.sleep(0.01)
     sum(x != y) + k
 }
> size <- 10L; y <- runif(size) > 0.5; k <- 10
> algo <- list(nB = size, nP = 20L, nG = 100L, prob = 0.002,
              printBar = FALSE, printDetail = FALSE,
              methodOF = "loop")
> t1 <- system.time(sol <- GAopt(OF, algo = algo, y = y, k = k))
> all.equal(sol$xbest, y)
```

```
[1] TRUE
```

```
> all.equal(sol$OFvalue, k)
```

```
[1] TRUE
```

```
> algo <- list(nB = size, nP = 20L, nG = 100L, prob = 0.002,
              printBar = FALSE, printDetail = FALSE,
              methodOF = "snow", cl = nC)
> t2 <- system.time(sol <- GAopt(OF, algo = algo, y = y, k = k))
> all.equal(sol$xbest,y)
```

```
[1] TRUE
```

```
> all.equal(sol$OFvalue, k)
```

```
[1] TRUE
```

5 gridSearch

We use a simple test function.

```
> testFun  <- function(x) {
     Sys.sleep(0.1)
     x[1L] + x[2L]^2
 }
> lower <- 1:2; upper <- 5; n <- 10
> with_loop <- expression(
     sol1 <- gridSearch(fun = testFun,
                        lower = lower, upper = upper,
                        n = n, printDetail = FALSE))
> with_multicore <- expression(
     sol2 <- gridSearch(fun = testFun,
                        lower = lower, upper = upper,
                        n = n, printDetail = FALSE,
                        method = "multicore"))
> with_snow <- expression(
     sol3 <- gridSearch(fun = testFun,
                        lower = lower, upper = upper,
                        n = n, printDetail = FALSE,
                        method = "snow",
                        cl = nC))
> benchmark(with_loop, with_multicore, with_snow,
           columns = c("test", "elapsed", "relative"),
           order = "relative", replications = 1)
```

```
            test elapsed relative
3      with_snow    4.14     1.00
2 with_multicore    5.02     1.21
1      with_loop   10.03     2.42
```

```
> all.equal(sol1, sol2)
```

```
[1] TRUE
```

```
> all.equal(sol1, sol3)
```

```
[1] TRUE
```

```
> all.equal(sol3$minlevels, 1:2)
```

```
[1] TRUE
```

This test function may also need additional arguments. Here we pass a variable k.

```
> testFun  <- function(x, k) {
     Sys.sleep(0.1)
     x[1L] + x[2L]^2 + k
 }
> lower <- 1:2; upper <- 5; n <- 5; k <- 1
> sol1 <- gridSearch(fun = testFun, k = k,
                     lower = lower, upper = upper,
                     n = n, printDetail = FALSE)
> sol2 <- gridSearch(fun = testFun,k = k,
                     lower = lower, upper = upper,
                     n = n, printDetail = FALSE,
                     method = "multicore")
> sol3 <- gridSearch(fun = testFun,k = k,
                     lower = lower, upper = upper,
                     n = n, printDetail = FALSE,
                     method = "snow", cl = nC)
> all.equal(sol1, sol2)
```

```
[1] TRUE
```

```
> all.equal(sol1, sol3)
```

```
[1] TRUE
```

```
> all.equal(sol3$minlevels, 1:2)
```

```
[1] TRUE
```

To pass optional arguments to `parallel`'s `mclapply`, we need to collect them in a list `mc.control`, which needs to be added to `algo`. Here we set a seed.

```
> testFun  <- function(x) {
     Sys.sleep(0.1)
     x[1L] + x[2L] + runif(1)
 }
> lower <- 1:2; upper <- 5; n <- 3
> set.seed(5)
> sol2 <- gridSearch(fun = testFun,
                    lower = lower, upper = upper,
                    n = n, printDetail = FALSE,
                    method = "multicore",
                    mc.control = list(mc.set.seed = FALSE))
> temp <- sol2$values
> set.seed(5)
> sol2 <- gridSearch(fun = testFun,
                    lower = lower, upper = upper,
                    n = n, printDetail = FALSE,
                    method = "multicore",
                    mc.control = list(mc.set.seed = FALSE))
> all.equal(sol2$values, temp)
```

```
[1] TRUE
```

Setting a seed is also possible with method `snow`.

```
> cl <- makeCluster(c(rep("localhost", nC)), type = "SOCK")
> clusterSetRNGStream(cl, 2222)
> sol3 <- gridSearch(fun = testFun, lower = lower, upper = upper,
                    n = n, printDetail = FALSE,
                    method = "snow", cl = cl)
> stopCluster(cl)
> temp <- sol3$values
> ## ... and again
> cl <- makeCluster(c(rep("localhost", nC)), type = "SOCK")
> clusterSetRNGStream (cl, 2222)
> sol3 <- gridSearch(fun = testFun, lower = lower, upper = upper,
                     n = n, printDetail = FALSE,
                     method = "snow", cl = cl)
> stopCluster(cl)
> all.equal(sol3$values, temp)
```

```
[1] TRUE
```

6 restartOpt

We test with TAopt and a toy problem: find a numeric vector x that matches another numeric vector xTRUE through randomly changing x.

```
> xTRUE <- runif(5L)
> data <- list(xTRUE = xTRUE,  ## the TRUE solution
              step = 0.02     ## step size for neighbourhood
              )
> OF <- function(x, data)
     max(abs(x - data$xTRUE))
> neighbour <- function(x, data)
     x + runif(length(data$xTRUE))*data$step - data$step/2
> x0 <- runif(5L)               ## a random starting solution
> algo <- list(q = 0.05, nS = 200L, nT = 10L,
              neighbour = neighbour, x0 = x0,
              printBar = FALSE, printDetail = FALSE)
```

Now we call restartOpt.

```
> with_loop <- expression(
     sols1 <- restartOpt(fun = TAopt, n = 100L,
                         OF = OF, algo = algo, data = data))
> with_multicore <- expression(
     sols2 <- restartOpt(fun = TAopt, n = 100L,
                         OF = OF, algo = algo, data = data,
                         method = "multicore"))
> with_snow <- expression(
     sols3 <- restartOpt(fun = TAopt, n = 100L,
                         OF = OF, algo = algo, data = data,
                         method = "snow", cl = nC))
> benchmark(with_loop, with_multicore, with_snow,
           columns = c("test", "elapsed", "relative"),
           order = "relative", replications = 1)
```

```
             test elapsed relative
3       with_snow    3.08     1.00
2  with_multicore    3.37     1.09
1       with_loop    6.19     2.01
```

```
> all.equal(length(sols1), 100L)
```

```
[1] TRUE
```

```
> all.equal(length(sols2), 100L)
```

```
[1] TRUE
```

```
> all.equal(length(sols3), 100L)
```

```
[1] TRUE
```

A Resources

You can download all the book's code examples from the book's home page,

`http://nmof.net`

The latest version of the NMOF package is available from

`http://enricoschumann.net/R/packages/NMOF/index.htm`

but note that this is the development version. More stable versions are available from CRAN.

New versions of the package and other news are announced through the `NMOF-news` mailing list; to browse the archives or to subscribe, go to

`https://lists.r-forge.r-project.org/cgi-bin/mailman/listinfo/nmof-news`

B Package version

```
> toLatex(sessionInfo())
```

- R version 3.1.1 (2014-07-10), `x86_64-pc-linux-gnu`
- Locale: `LC_CTYPE=en_GB.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=en_GB.UTF-8`, `LC_MONETARY=en_US.UTF-8`, `LC_MESSAGES=en_GB.UTF-8`, `LC_PAPER=en_US.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_US.UTF-8`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, graphics, grDevices, methods, parallel, stats, utils
- Other packages: NMOF 0.34-0, rbenchmark 1.0.0
- Loaded via a namespace (and not attached): snow 0.3-13, tools 3.1.1

References

Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. Elsevier/Academic Press, 2011.

Wacek Kusnierczyk. *rbenchmark: Benchmarking routine for R*, 2010. URL `http://CRAN.R-project.org/package=rbenchmark`. R package version 0.3.

Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In Wolfgang Härdle and Bernd Rönz, editors, *Compstat 2002 – Proceedings in Computational Statistics*, pages 575–580. Physica Verlag, 2002.

Luke Tierney, A. J. Rossini, Na Li, and H. Sevcikova. *snow: Simple Network of Workstations*, 2011. URL `http://CRAN.R-project.org/package=snow`. R package version 0.3-7.

Simon Urbanek. *multicore: Parallel processing of R code on machines with multiple cores or CPUs*, 2011. URL `http://CRAN.R-project.org/package=multicore`. R package version 0.1-7.