optimParallel: An R Package Providing a Parallel Version of the L-BFGS-B Optimization Method

by Florian Gerber, Reinhard Furrer

Abstract The R package **optimParallel** (Gerber, 2019) provides a parallel version of the L-BFGS-B optimization method of `optim()`. The main function of the package is `optimParallel()`, which has the same usage and output as `optim()`. Using `optimParallel()` can significantly reduce the optimization time, especially when the evaluation time of the objective function is large and no analytical gradient is available. We introduce the R package and illustrate its implementation, which takes advantage of the lexical scoping mechanism of R.

Introduction

Many statistical tools involve optimization algorithms, which aim to find the minima or maxima of an objective function $fn : \mathbb{R}^p \to \mathbb{R}$, where $p \in \mathbb{N}$ denotes the number of parameters. Depending on the specific application different optimization algorithms may be preferred; see the book by Nash (2014), the special issue of the Journal of Statistical Software (Varadhan, 2014), and the CRAN Task View *Optimization* for overviews of the optimization software available for R. A widely used algorithm is the limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm with box constraints (L-BFGS-B, Byrd et al., 1996), which is available through the general-purpose optimizer `optim()` of the R package **stats** and the more recent R packages **lbfgsb3** (Nash et al., 2015) and **lbfgsb3c** (Fidler et al., 2018). The L-BFGS-B algorithm has proven to work well in numerous application. However, long optimization times of computationally intensive functions sometimes hinder their application; see Gerber et al. (2017) for an example of such a function from our research in spatial statistics. For this reason we present a parallel version of the `optim()` L-BFGS-B algorithm denoted with `optimParallel()` and explore its potential to reduce optimization times.

To illustrate the possible speed gains of a parallel L-BFGS-B implementation let $gr : \mathbb{R}^p \to \mathbb{R}^p$ denote the gradient of $fn()$. L-BFGS-B always first evaluates $fn()$ and then $gr()$ at the same parameter value and we call one such sequential evaluation one *step*. Note that this step should not be confused with the *iteration* as defined in Byrd et al. (1996) and used in the help page of `optim()`, because the latter may involve several steps. Let T_{fn} and T_{gr} denote the evaluation times of $fn()$ and $gr()$, respectively. In the case where $gr()$ is specified in the `optim()` call, one step of the L-BFGS-B algorithm evaluates $fn()$ and $gr()$ sequentially, and hence, the evaluation time is little more than $T_{fn} + T_{gr}$ per step. In contrast, `optimParallel()` evaluates both functions in parallel, which reduces the evaluation time to little more than $\max\{T_{fn}, T_{gr}\}$ per step. In the case where no gradient is provided, `optim()` calculates a numeric central difference gradient approximation (CGA). For $p = 1$ the CGA is defined as

$$\widetilde{gr}(x) = (fn(x+\epsilon) - fn(x-\epsilon)) \,/\, 2\epsilon, \tag{1}$$

and hence, requires two evaluations of $fn()$. Similarly, calculating $\widetilde{gr}()$ requires $2p$ evaluations of $fn()$ if $fn()$ has p parameters. In total, `optim()` sequentially evaluates $fn()$ $1 + 2p$ times per step, resulting in an elapsed time of little more than $(1 + 2p)T_{fn}$ per step. Given $1 + 2p$ available processor cores, `optimParallel()` evaluates all calls of $fn()$ in parallel, which reduces the elapsed time to little more than T_{fn} per step.

`optimParallel()` by examples

The main function of the R package **optimParallel** is `optimParallel()`, which has the same usage and output as `optim()`, but evaluates $fn()$ and $gr()$ in parallel. For illustration, consider 10^7 samples from a normal distribution with mean $\mu = 5$ and standard deviation $\sigma = 2$. Then, we define the following negative log-likelihood and use `optim()` to estimate the parameters μ and σ.[1]

[1]It is obvious that simpler ways exists to estimate μ and σ. Moreover, a computationally more efficient version of `negll()` can be constructed based on `mean(x)` and `sd(x)`.

```
> x <- rnorm(n = 1e7, mean = 5, sd = 2)
> negll <- function(par, x) -sum(dnorm(x = x, mean = par[1], sd = par[2], log = TRUE))
> o1 <- optim(par = c(1, 1), fn = negll, x = x, method = "L-BFGS-B",
+             lower = c(-Inf, 0.0001))
> o1$par
[1] 5.000597 2.000038
```

Using `optimParallel()`, we can do the same in parallel. The functions `makeCluster()`, `detectCores()`, and `setDefaultCluster()` from the R package **parallel** are used to set up a default cluster for the parallel execution.

```
> install.packages("optimParallel")
> library("optimParallel")
Loading required package: parallel
> cl <- makeCluster(detectCores()); setDefaultCluster(cl = cl)
> o2 <- optimParallel(par = c(1, 1), fn = negll, x = x, lower = c(-Inf, 0.0001))
> identical(o1, o2)
[1] TRUE
```

On our computer with 12 Intel Xeon E5-2640 @ 2.50GHz processors one evaluation of `negll()` took 0.9 seconds. `optim()` run with 6.2 seconds per step, whereas `optimParallel()` run with 1.8 seconds per step. Thus, the optimization time of `optimParallel()` is reduced by 71% compared to that of `optim()`. Note that for $p = 2$, 71% is below the maximal possible reduction of $1 - 1/(2p + 1) = 80\%$ because of the overhead associated with the parallel execution and the time needed to run the L-BFGS-B algorithm, which are both not taken into account in this calculation. In general, the reduction of the optimization time is large if the parallel overhead is small relative to the execution time of $fn()$. Hence, for this example, the reduction of the optimization time approaches 80% when the evaluation time of `negll()` is increased, e. g., by increasing the number of data points in `x`.

In addition to the arguments of `optim()`, `optimParallel()` has the argument `parallel`, which takes a list of arguments. For example, we can set `parallel = list(loginfo = TRUE)` to store all evaluated parameters and the corresponding gradients.

```
> o3 <- optimParallel(par = c(1, 1), fn = negll, x = x, lower = c(-Inf, 0.0001),
+                     parallel = list(loginfo = TRUE))
> head(o3$loginfo, n = 3)
     step     par1     par2        fn       gr1        gr2
[1,]    1 1.000000 1.000000 109213991 -40005963 -190049608
[2,]    2 1.205988 1.978554  39513324  -9693283  -18700810
[3,]    3 1.265626 2.086455  37160791  -8579638  -14969646
> tail(o3$loginfo, n = 3)
      step     par1     par2       fn         gr1       gr2
[16,]   16 5.000840 2.000140 21121045 609.9480540 507.56421
[17,]   17 5.000586 2.000041 21121045 -26.8237162  15.17266
[18,]   18 5.000597 2.000038 21121045   0.6494038  -1.67717
```

This can be used to visualize the optimization path as shown in Figure 1 and simplifies the following study, which illustrates the impact of using different (approximate) gradient specifications.

In the `optimParallel()` calls above the argument `gr` was not specified, and hence, the CGA was used. Another way of using `optimParallel()` is with a gradient function given to the argument `gr`. If the computation of the analytical gradient is tractable and does not take more time than evaluating $fn()$, this usage is preferred.

```
> negll_gr <- function(par, x){
+     sm <- mean(x); n <- length(x)
+     c(-n*(sm-par[1])/par[2]^2,
+       n/par[2] - (sum((x-sm)^2) + n*(sm-par[1])^2)/par[2]^3)
+ }
> o4 <- optimParallel(par = c(1, 1), fn = negll, gr = negll_gr, x = x,
+                     lower = c(-Inf, 0.0001), parallel = list(loginfo = TRUE))
> tail(o4$loginfo, n = 3)
      step     par1     par2       fn         gr1        gr2
[16,]   16 5.000840 2.000139 21121045 609.9651113 507.625076
[17,]   17 5.000586 2.000041 21121045 -26.8233339  15.172072
[18,]   18 5.000597 2.000037 21121045   0.6494452  -1.677113
```

We see that the resulting optimization path is very similar to that based on the CGA (`o3` above).

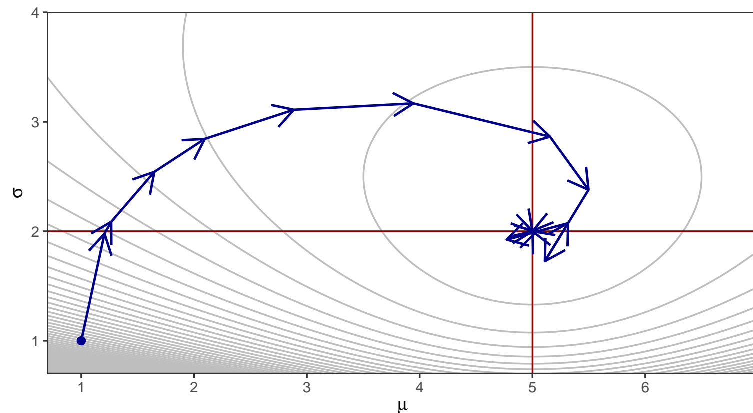

Figure 1: Visualization of the optimization path based on the log information obtained with `optimParallel(..., parallel = list(loginfo = TRUE))`. The red lines mark the estimates $\hat{\mu}$ and $\hat{\sigma}$.

Besides the CGA and the option to explicitly specify a gradient function, `optimParallel()` provides a third option, namely the numeric forward difference gradient approximation (FGA) defined as $\widetilde{gr}(x) = (fn(x+\epsilon) - fn(x))/\epsilon$ for $p = 1$ and x sufficiently away from the boundaries. Using the FGA, the return value of $fn(x)$ can be reused for the computation of the gradient, and hence, the number of evaluations of $fn()$ is reduced to $1+p$ per step. This can be helpful if the number of available cores is less than $1+2p$ or insufficient memory is available to run $1+2p$ evaluations of $fn()$ in parallel.

```
> o5 <- optimParallel(par = c(1, 1), fn = negll, x = x, lower = c(-Inf, 0.0001),
+                     parallel = list(loginfo = TRUE, forward = TRUE))
> o5$loginfo[17:19, ]
     step     par1     par2       fn       gr1       gr2
[1,]   17 5.000086 1.999541 21121046 -26.33029  14.35781
[2,]   18 5.000331 1.999645 21121045 586.89953 534.85303
[3,]   19 5.000346 1.999651 21121045 625.10421 567.27441
> tail(o5$loginfo, n = 3)
      step     par1     par2       fn      gr1      gr2
[31,]   31 5.000347 1.999652 21121045 627.8436 569.5991
[32,]   32 5.000347 1.999652 21121045 627.8436 569.5991
[33,]   33 5.000347 1.999652 21121045 627.8436 569.5991
```

We see that the optimizer only stopped after 33 steps, whereas all previous optimization calls stopped after 18 steps. Hence, it is obvious that the choice of the gradient approximation affects the optimization. But what happened exactly?—It should be noted that the FGA is less accurate than the CGA; see, e. g., the numerical study in Nash (2014), Section 10.7. Hence, small differences in the optimization path are expected, but this does hardly explain the additional 15 steps used by the FGA based optimization. A closer inspection of the optimization path reveals that up to step 18 the path is very similar to those of the previous optimization calls and the remaining steps 19–33 only marginally change `par1` and `par2`. This suggests that using the FGA prevents the algorithm from stopping. One way to handle this issue is to set a less restrictive stopping criterion by increasing the value of `factr`.

```
> o6 <- optimParallel(par = c(1, 1), fn = negll, x = x, lower = c(-Inf, 0.0001),
+                     parallel = list(loginfo = TRUE, forward = TRUE),
+                     control = list(factr = 1e-6/.Machine$double.eps))
> tail(o6$loginfo, n = 3)
      step     par1     par2       fn        gr1        gr2
[14,]   14 4.996680 2.001974 21121074 -8524.7678 12125.5022
[15,]   15 4.999743 1.998478 21121052  -884.4955 -5305.2516
[16,]   16 5.000347 1.999652 21121045   627.8436   569.5991
```

Now the resulting optimization path and the evaluation counts are similar to those resulting from the optimization using the analytic gradient (o4 above). The take-home message of this study is that the choice of the (approximate) gradient can affect the optimization path and it may be necessary to adjust control parameters such as `factr`, `ndeps`, and `parscale` to obtain satisfactory results. A more detailed discussion of the use of (approximate) gradients in optimization can be found in Nash (2014), Chapter 10.

Implementation

optimParallel() is a wrapper to optim() and enables the parallel evaluation of all function calls involved in one step of the L-BFGS-B optimization method. It is implemented in R and interfaces compiled C code only through optim(). The reuse of the stable and well-tested C code of optim() has the advantage that optimParallel() leads to the exact same results as optim(). To ensure that optimParallel() and optim() indeed return the same results **optimParallel** contains systematic unit tests implemented with the R package **testthat** (Wickham, 2017, 2011).

The basic idea of the implementation is that calling fn() (or gr()) triggers the evaluation of both fn() and gr(). Their return values are stored in a local environment. The next time fn() (or gr()) is called with the same parameters the results are read from the local environment without evaluating fn() and gr() again. The following R code illustrates how optimParallel() takes advantage of the lexical scoping mechanism of R to store the return values of fn() and gr().

```
> demo_generator <- function(fn, gr) {
+     par_last <- value <- grad <- NA
+     eval <- function(par) {
+         if(!identical(par, par_last)) {
+             message("--> evaluate fn() and gr()")
+             par_last <<- par
+             value <<- fn(par)
+             grad <<- gr(par)
+         } else
+             message("--> read stored value")
+     }
+     f <- function(par) {
+         eval(par = par)
+         value
+     }
+     g <- function(par) {
+         eval(par = par)
+         grad
+     }
+     list(fn = f, gr = g)
+ }
> demo <- demo_generator(fn = sum, gr = prod)
```

Calling demo$fn() triggers the evaluation of both fn() and gr().

```
> demo$fn(1:5)
--> evaluate fn() and gr()
[1] 15
```

The subsequent call of demo$gr() with the same parameters returns the stored value grad without evaluating gr() again.

```
> demo$gr(1:5)
--> read stored value
[1] 120
```

A similar construct allows optimParallel() to evaluate $fn()$ and $gr()$ at the same occasion. It is then straightforward to parallelize the evaluations of the functions using the R package **parallel**.

Speed gains and performance test

To illustrate the speed gains that can be achieved with optimParallel() we measure the elapsed times per step when optimizing the following test function and compare them to those of optim().

```
> fn <- function(par, sleep) {
+     Sys.sleep(sleep)
+     sum(par^2)
+ }
> gr <- function(par, sleep) {
+     Sys.sleep(sleep)
+     2*par
+ }
```

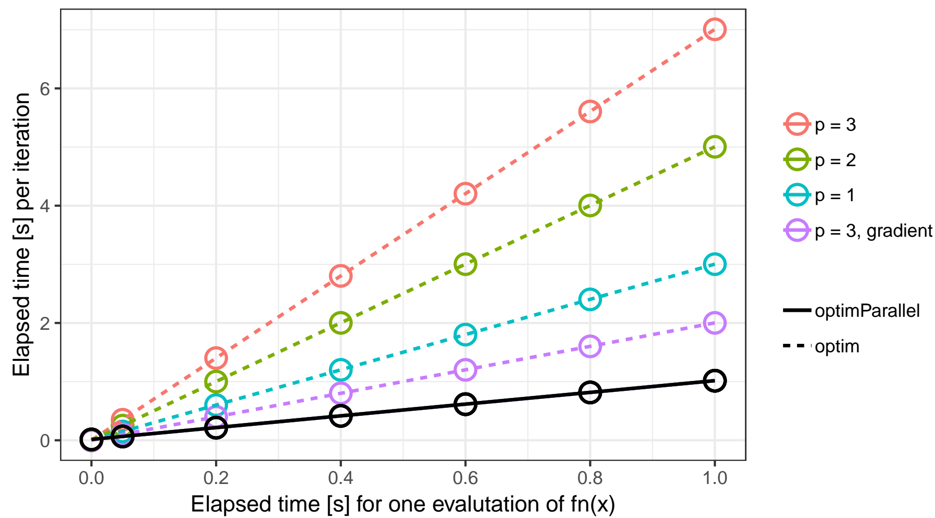

Figure 2: Benchmark experiment comparing the L-BFGS-B method from `optimParallel()` and `optim()`. Plotted are the elapsed times per step (y-axis) and the evaluation time of $fn()$ (x-axis) for $p = 1$, 2, and 3 using an approximate gradient and $p = 3$ using an analytic gradient. The elapsed times of `optimParallel()` (solid line) are independent of p and the specification of an analytic gradient.

In both functions the argument `par` can be a numeric vector with one or more elements and the argument `sleep` controls the evaluation time of the functions. We measure the elapsed time per step using all combinations of $p = 1, 2, 3$, `sleep` = 0, 0.05, 0.2, 0.4, 0.6, 0.8, 1 seconds with and without analytic gradient `gr()`. All measurements are taken on a computer with 12 Intel Xeon E5-2640@2.50GHz processors. However, because of the experimental design a maximum of 7 processors are used in parallel. We repeat each measurement 5 times using the R package **microbenchmark** (Mersmann et al., 2018). The complete R script of the benchmark experiment is contained in **optimParallel**.

The results of the benchmark experiment are summarized in Figure 2. They show that for `optimParallel()` the elapsed time per step is only marginally larger than T_{fn} (black circles in Figure 2). Conversely, the elapsed time for `optim()` is $T_{fn} + T_{gr}$ if a gradient function is specified (violet circles) and $(1 + 2p)T_{fn}$ if no gradient function is specified (red, green, and blue circles). Moreover, `optimParallel()` adds a small overhead, and hence, it is only faster than `optim()` for T_{fn} larger than 0.05 seconds.

The use of `Sys.sleep()` in this illustration is useful to characterize the implementation and its overhead. However, it does not represent a practical use case of `optimParallel()` and the speed gains might be less pronounced for other examples. One factor that reduces the speed of `optimParallel()` is the specification of large objects in its `"..."` argument. All those objects are copied to the running R sessions in the cluster, which increases the elapsed time. Related to that is the increased memory usage, which may slowdown the optimization when not enough memory is available.

Summary

The R package **optimParallel** provides a parallel version of the L-BFGS-B optimization method of `optim()`. After a brief theoretical illustration of the possible speed improvement based on parallel processing, we illustrate `optimParallel()` by examples. The examples demonstrate that one can replace `optim()` by `optimParallel()` to execute the optimization in parallel and illustrate additional features like capturing log information and using different (approximate) gradients. Moreover, we briefly sketch the basic idea of the implementation, which is based on the lexical scoping mechanism of R. Finally, a performance test shows that using `optimParallel()` reduces the elapsed time to optimize computationally demanding functions significantly. For functions with evaluation times of more than 0.1 seconds we measured speed gains of about factor 2 in the case where an analytic gradient was specified and about factor $1 + 2p$ otherwise (p is the number of parameters). Our results suggest that using `optimParallel()` is most beneficial when (i) the evaluation time of $fn()$ is large (more than 0.1 seconds), (ii) no analytical gradient is available, and (iii) p or more processors as well as enough memory are available.

Bibliography

R. Byrd, L. Peihuang, and J. Nocedal. A limited-memory algorithm for bound-constrained optimization. *Technical Report*, 1996. URL https://doi.org/10.2172/204262. [p1]

M. L. Fidler, J. C. Nash, C. Zhu, R. Byrd, J. Nocedal, and J. L. Morales. *lbfgsb3c: Limited memory BFGS minimizer with bounds on parameters with optim() 'C' interface*, 2018. URL https://CRAN.R-project.org/package=lbfgsb3c. R package version 2018-2.13-1. [p1]

F. Gerber. *optimParallel: A parallel version of the L-BFGS-B method optim()*, 2019. URL https://CRAN.R-project.org/package=optimParallel. R package version 0.8. [p1]

F. Gerber, K. Mösinger, and R. Furrer. Extending R packages to support 64-bit compiled code: An illustration with spam64 and GIMMS $NDVI_{3g}$ data. *Computers & Geosciences*, 104:109–119, 2017. URL https://doi.org/10.1016/j.cageo.2016.11.015. [p1]

O. Mersmann, C. Beleites, R. Hurling, A. Friedman, and J. M. Ulrich. *microbenchmark: Accurate timing functions*, 2018. URL https://CRAN.R-project.org/package=microbenchmark. R package version 1.4-4. [p5]

J. C. Nash. *Nonlinear Parameter Optimization Using R Tools*. Wiley & Sons, Ltd, 2014. ISBN 9781118883969. URL https://doi.org/10.1002/9781118884003. [p1, 3]

J. C. Nash, C. Zhu, R. Byrd, J. Nocedal, and J. L. Morales. *lbfgsb3: limited memory BFGS minimizer with bounds on parameters*, 2015. URL https://CRAN.R-project.org/package=lbfgsb3. R package version 2015-2.13. [p1]

R. Varadhan. Special volume: Numerical optimization in R: Beyond optim. *Journal of Statistical Software*, 6, 2014. ISSN 1548-7660. URL https://www.jstatsoft.org/issue/view/v060. [p1]

H. Wickham. testthat: Get started with testing. *The R Journal*, 3:5–10, 2011. URL http://journal.R-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf. [p4]

H. Wickham. *testthat: Unit testing for R*, 2017. URL https://CRAN.R-project.org/package=testthat. R package version 2.0.0. [p4]

Dr. Florian Gerber
Department of Applied Mathematics and Statistics
Colorado School of Mines, Colorado, USA
gerber@mines.edu, https://orcid.org/0000-0001-8545-5263

Prof. Dr. Reinhard Furrer
Department of Mathematics & Department of Computational Science
University of Zurich, Switzerland
reinhard.furrer@math.uzh.ch, https://orcid.org/0000-0002-6319-2332