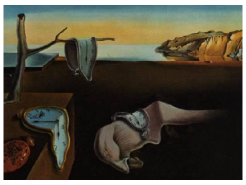

Journal of Statistical Software

MMMMMM YYYY, Volume VV, Code Snippet II. http://www.jstatsoft.org/

dcpar: Parallel Computing with MCMC and Data Cloning in R

Péter Sólymos
University of Alberta

Abstract

Data cloning is a resource intensive global optimization technique, that uses Bayesian MCMC tools to get maximum likelihood estimates and corresponding standard errors of statistical models. The **dcpar** R package is a parallel computing extension for the **dclone** package that can be used to fit models based on data cloning. Data cloning often requires that the same model is fitted with different number of clones of the data set. Processing time is linearly related to the size of the problems (number of clones). Both by using parallel MCMC chains, or by partitioning the problem into parallel subsets, computing time decreases considerably even with few parallel workers. The **dcpar** package also includes functions to help in optimizing workloads on parallel workers.

Keywords: Bayesian statistics, data cloning, maximum likelihood inference, generalized linear models, R, parallel computing.

1. Introduction

Data cloning is a statistical computing method introduced by Lele *et al.* (2007). It is a global optimization technique, that exploits the computational simplicity of the Markov chain Monte Carlo (MCMC) algorithms used in the Bayesian statistical framework, but it provides valid frequentist inferences such as the maximum likelihood estimates and their standard errors. The basic idea of data cloning is, that if we copy identical clones of the data k times, we can approach the asymptotic maximum likelihood estimates corresponding to k times the log-likelihood. This approach enables to fit complex hierarchical models (Lele *et al.* 2007; Ponciano *et al.* 2009) and helps in studying parameter identifiability issues (Lele *et al.* 2009), but comes with a price in processing time that increases with the number of clones. Because of the "embarassingly parallel" nature of the MCMC computations, and the availability of multiple core machines and computing clusters, parallelization naturally follows as a reply to the often demanding computational requirements of data cloning.

The R (R Development Core Team 2009) package **dclone** (Sólymos 2009, 2010 under review) provide low level functionality to implement estimating procedures based on data cloning. The **dcpar** package (Sólymos 2010) extends the **dclone** package for parallel computations, building on the parallel computing infrastructure provided by the **snow** package (Tierney *et al.* 2010). In this paper I demonstrate how computing time can improve by using **dcpar**, and how can the parallelization effectively been optimized. The **dcpar** package currently supports the JAGS software (Plummer 2009), but future development might include support to other MCMC software (e.g. WinBUGS (Spiegelhalter *et al.* 2003) and OpenBUGS (Spiegelhalter *et al.* 2007)[1]).

2. Ways of parallelization

There is no universal number of clones that is needed to reach convergence (the asymptotic joint distribution of the parameters is degenerate multivariate normal (Lele *et al.* 2007)), so k has to be determined empirically, by fitting the same model with the data of different number of clones, and checking if the variances of the parameters are going to zero (Lele *et al.* 2007, 2009). For example, if we fit the same model with different number of clones, $k = (1, 2, 5, 10, 20)$, it means that we have to fit five models. Without parallelization, we can use subsequent calls of the `jags.fit` function with different cloned data sets (corresponding to different number of clones), or use the `dc.fit` wrapper function of the **dclone** R package.

Parallelization can happen in two ways: (1) we parallelize the computation of the MCMC chains (ususally we use more than one chain to check proper mixing behaviour of the chains); or (2) we split the problem into subsets and fit the models in each subset on a different worker of the parallel computing environment. The first type of parallelization can be addressed by the `jags.parfit` function (parallelized version of `jags.fit`), while the second type can be addressed by the `dc.parfit` function (parallelized version of `dc.fit`). Both `jags.parfit` and `dc.parfit` are built upon the parallel computing infrastructure provided by the **snow** package, with some additional convenince functions (see next sections). The most important additional function is the `snowWrapper`, which unites several basic **snow** functions (i.e. `clusterCall`, `clusterEvalQ`, `clusterApply`), and most importantly can execute the `clusterExport` function call from within a function (where newly created objects are not defined in the global environment).

3. Example data and the Bayesian model

We will use a simulated data set correponding to a Poisson generlaized linear model with random intercept for i.i.d. observations of Y_i counts from $i = 1, 2, \ldots, n$ localities (for a rationale of this model, see Sólymos (2009)):

$$\begin{aligned}(Y_i \mid \lambda_i) &\sim \text{Poisson}(\lambda_i)\\ \log(\lambda_i) &= \beta_{0i} + X_i\beta_1\\ \beta_{0i} &\sim \text{Normal}(\beta_0, \sigma^2).\end{aligned}$$

[1]The **bugsParallel** project (Metrum Institute 2010) provide R scripts for parallel computing with WinBUGS.

The R code for the data generation corresponding to this model (with parameters $n = 200$, $\beta_0 = 1$, $\beta_1 = -1$, $\sigma = 0.25$) is:

```
R> n <- 200
R> beta <- c(1, -1)
R> sigma <- 0.25
R> set.seed(1234)
R> x <- runif(n)
R> X <- model.matrix(~x)
R> mu <- rnorm(n, mean=drop(X %*% beta), sd=sigma)
R> Y <- rpois(n, exp(mu))
```

We put these objects into a list (this will be the data for JAGS):

```
R> dat <- list(Y = Y, X = X, n = n, np = NCOL(X))
```

The corresponding JAGS model is:

```
R> glmm.model <- function() {
+       for (i in 1:n) {
+           Y[i] ~ dpois(exp(mu[i]))
+           mu[i] ~ dnorm(inprod(X[i,], beta), 1/exp(log.sigma)^2)
+       }
+       for (j in 1:np) {
+           beta[j] ~ dnorm(0, 0.001)
+       }
+       log.sigma ~ dnorm(0, 0.001)
+   }
```

Note that we use $\log(\sigma)$ with Normal prior distribution to enhance chanin mixing, and because of the asymptotic multivariate normality involved in the theory of data cloning (Lele *et al.* 2007). We are using the following settings (`n.adapt` + `n.update` = burn-in iterations, `n.iter` is the number of samples taken from the posterior distribution) for fitting the JAGS model:

```
R> n.adapt <- 2000
R> n.update <- 8000
R> n.iter <- 2000
```

And finally set the vector of k for the number of clones to be used:

```
R> k <- c(1, 2, 5, 10, 20)
```

All computations presented here were run on a computer with Intel® Core™ i7-920 CPU, 3 GB RAM and Windows XP operating system.

4. Parallel MCMC chains

The following simple function allows us to switch parallelization on/off (by setting the `parallel` argument as `TRUE`/`FALSE`, respectively), and also measure the time elapsed in minutes:

```
R> timerfitfun <- function(parallel = FALSE, ...) {
+       t0 <- proc.time()
+       mod <- if (parallel)
+           jags.parfit(...) else jags.fit(...)
+       attr(mod, "timer") <- (proc.time() - t0)[3] / 60
+       mod
+   }
```

First we fit the five models sequentially (one model for each element of `k`) without parallelization of the three MCMC chains:

```
R> res1 <- lapply(k, function(z)
+       timerfitfun(parallel = FALSE,
+           data = dclone(dat, z, multiply = "n", unchanged = "np"),
+           params = c("beta", "log.sigma"),
+           model = glmm.model,
+           n.adapt = n.adapt, n.update = n.update, n.iter = n.iter))
```

Now we fit the models by using MCMC chains computed on parallel workers. We use three workers (one worker for each MCMC chain, the type we use here is "socket", but type can be anything else accepted by the **snow** package), then fit the five models sequentially, and close the connection at the end of the process:

```
R> cl <- makeCluster(3, type = "SOCK")
R> res2 <- lapply(k, function(z)
+       timerfitfun(parallel = TRUE,
+           cl = cl,
+           data = dclone(dat, z, multiply = "n", unchanged = "np"),
+           params = c("beta", "log.sigma"),
+           model = glmm.model,
+           n.adapt = n.adapt, n.update = n.update, n.iter = n.iter))
R> stopCluster(cl)
```

Results are shown in Fig. 2. We can extract timer information from the result objects:

```
R> pt1 <- sapply(res1, function(z) attr(z, "timer"))
R> pt2 <- sapply(res2, function(z) attr(z, "timer"))
```

Processing time (column `time`) increased linearly with k (with and without parallelization), columns `rel.change` indicate change relative to $k = 1$:

```
R> tab1 <- data.frame(n.clones=k,
+       time=pt1,
+       rel.change = pt1 / pt1[1],
+       time.par=pt2,
+       rel.change.par = pt2 / pt2[1])
R> round(tab1, 2)

  n.clones  time rel.change time.par rel.change.par
1        1  0.72       1.00     0.30           1.00
2        2  1.39       1.94     0.53           1.76
3        5  3.42       4.76     1.20           3.99
4       10  6.81       9.47     2.41           8.01
5       20 13.63      18.97     5.05          16.78
```

It took a total of 25.97 minutes to fit the five models without, and only 9.49 minutes with parallel MCMC chains (36.53 %). Parallel MCMC computations have effectively reduced the processing time to almost 1/(number of chains) = 1/3.

5. Partitioning the problem into parallel subsets

The other way of parallelizing the iterative fitting procedure with data cloning is to split the problem into subsets with respect to the problem size (i.e. approximate processing time). As we saw in the previous section, processing time increased linearly with k. Consequently, relative processing time can be effectively approximated by k.

The **dcpar** package has some underlying functions (e.g. `clusterSplitSB`, `parLapplySB`[2]) developped to deal with *size balancing*. In size balancing, the problems are re-ordered from largest to smallest, and then subsets are determined by minimizing the total approximate processing time. This splitting is deterministic, thus computations are reproducible. However, size balancing can be combined with load balancing (Rossini *et al.* 2003). This option is implemented in the `parLapplySLB` function. If size (processing time) is correct, this should be identical to size balancing, but the actual sequence of execution might depend on unequal performance of the workers. The splitting in this case is non-deterministic (might not be reproducible). Load balancing can be set as a global **dcpar** option by setting `options("dcpar.LB" = TRUE)` (this value is `FALSE` by default).

The `clusterSize` function can be used to determine the optimal number of workers needed for a given size vecor:

```
R> clusterSize(pt1)

  workers     none     load     size     both
1       1 25.97083 25.97083 25.97083 25.97083
2       2 20.43883 17.77417 13.63317 13.63317
3       3 20.43883 15.02417 13.63317 13.63317
4       4 13.63317 14.35183 13.63317 13.63317
5       5 13.63317 13.63317 13.63317 13.63317
```

The function returns approximate processing times (based on the size vector) as a function of the number of workers and balancing type. As we can see, only two workers are needed to accomplish the task by size balancing. The `plotCclusterSize` function can help to visualize the effect of using different balancing options:

```
R> opar <- par(mfrow=c(1,3), cex.lab=1.5, cex.main=1.5, cex.sub=1.5)
R> col <- heat.colors(length(k))
R> plotClusterSize(2, pt1, "none", col = col, xlim = c(0, 20))
R> plotClusterSize(2, pt1, "load", col = col[c(1, 3, 5, 2, 4)], xlim = c(0, 20))
R> plotClusterSize(2, pt1, "size", col = col[c(5, 4, 3, 2, 1)], xlim = c(0, 20))
R> par(opar)
```

We can see in Fig. 1, that, for two workers, size balancing results in the shortest processing time. While the model with $k = \max(k)$ is running, all the other models can be fitted on the other worker. Total processing time is determined by the largest problem size.

First, we fit the five models iteratively without parallelization by using the `dc.fit` function of the **dclone** package:

```
R> t0 <- proc.time()
R> res3 <- dc.fit(dat, c("beta", "log.sigma"), glmm.model,
```

[2]These functions resemble in name to the similar `clusterSplit` and `parLapply` functions in the **snow** package.

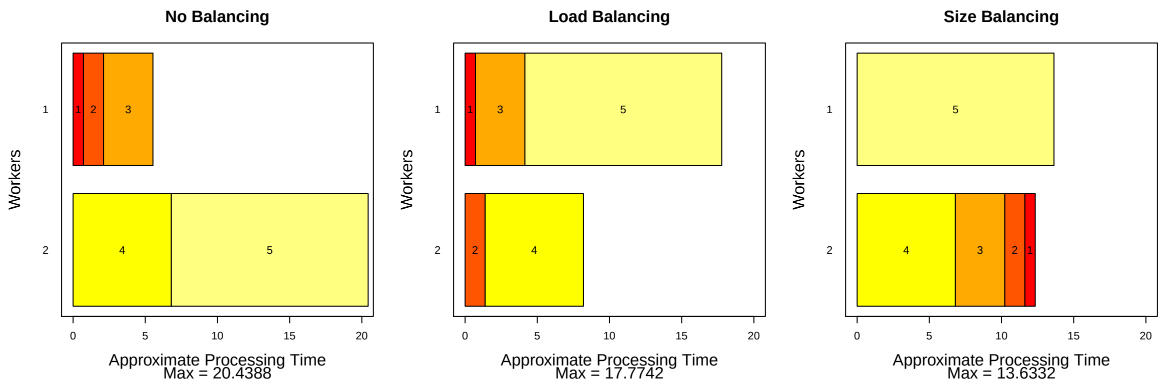

Figure 1: Effect of balancing type on approximate total processing time by showing the subsets on each worker. Note, the actual sequence of execution might be different for load balancing, because it is non-deterministic. Here, the actual times are shown, but relative performace can be well approximated by using the k vector.

```
+       n.clones = k, multiply = "n", unchanged = "np",
+       n.update = n.update, n.iter = n.iter)
R> attr(res3, "timer") <- (proc.time() - t0)[3] / 60
```

Now we fit the same model, using the same arguments, but with parallelization (using "socket" cluster type with two workers). The default balancing option in `dc.parfit` is size balancing:

```
R> cl <- makeCluster(2, type = "SOCK")
R> t0 <- proc.time()
R> res4 <- dc.parfit(cl, dat, c("beta", "log.sigma"), glmm.model,
+       n.clones = k, multiply = "n", unchanged = "np",
+       n.update = n.update, n.iter = n.iter)
R> attr(res4, "timer") <- (proc.time() - t0)[3] / 60
R> stopCluster(cl)
```

Results are shown in Fig. 2.

It took 23.44 minutes without parallelization, as compared to 12.31 minutes with parallelization (52.54 %). The parallel processing time in this case was close to the time needed to fit the model with $k = 20$ (13.63 minutes), so it was really the larges problem that determined the processing time.

6. Conclusions

If the length of the vector k (numbers of clones used) is short, or the number of available workers is limited to a few (2–4, i.e. the number of chains), the first type of parallelization (parallel MCMC chanis) and the use of the `jags.parfit` function can be more efficient. If the length of k is longer and the size of the computing cluster is large enough, the second type of parallelization (splitting the problem with respect to size) and the use of the `dc.parfit` function can be more efficient.

The user can decide which is the best way of parallellization by using the k vector and the

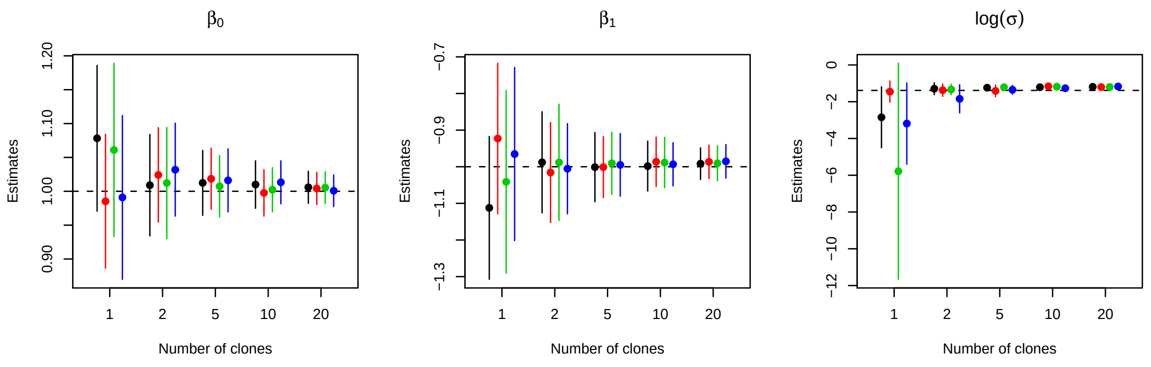

Figure 2: Estimated posterior means and posterior standard deviations for the parameters fitted by the functions `jags.fit` (black), `jags.parfit` (red), `dc.fit` (green), `dc.parfit` (blue). Means are converging to the maximum likelihood estimates (true values used in the simulations are presented as horizontal lines), posterior standard errors are decreasing with the number of clones in all four cases (this is an indication of data cloning convergence).

number of workers available. Performance of parallel MCMC chains can be well approximated by 1/(number of chains) = 1/3 = 0.33 (very close to the actual 0.37). Performance of parallel subsets can be calculated as `min(clusterSize(k)$size)/sum(k)` = 0.53 (which is the same as the actual 0.53). These values indicate that in this particular situation, choosing parallel MCMC chains is more efficient. The actual change in computing efficiency for other settings depends on the problem at hand and the performace of the computing environment. But these simple calculations help in guiding the decision.

The **dcpar** R package contains main functions to execute parallel computations with data cloning, and helper functions to optimize the workload on parallel workers in order to minimize processing time for computationally challenging problems.

7. Acknowledgments

Subhash Lele and Khurram Nadeem have provided invaluable suggestions and feedback on this work.

References

Lele SR, Dennis B, Lutscher F (2007). "Data cloning: easy maximum likelihood estimation for complex ecological models using Bayesian Markov chain Monte Carlo methods." *Ecology Letters*, **10**, 551–563.

Lele SR, Nadeem K, Schmuland B (2009). "Estimability and likelihood inference for generalized linear mixed models using data cloning." Submitted manuscript.

Metrum Institute (2010). *bugsParallel: R scripts for parallel computation of multiple MCMC chains with WinBUGS*. URL `http://code.google.com/p/bugsparallel/`.

Plummer M (2009). *JAGS Version 1.0.3 manual (April 23, 2009)*. URL http://mcmc-jags.sourceforge.net.

Ponciano JM, Taper ML, Dennis B, Lele SR (2009). "Hierarchical models in ecology: confidence intervals, hypothesis testing, and model selection using data cloning." *Ecology*, **90**, 356–362.

R Development Core Team (2009). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Rossini A, Tierney L, Li N (2003). "Simple Parallel Statistical Computing in R." *UW Biostatistics Working Paper Series*, **Working Paper 193.**, 1–27. URL http://www.bepress.com/uwbiostat/paper193.

Sólymos P (2009). *dclone: Data Cloning and MCMC Tools for Maximum Likelihood Methods*. R package version 1.0-0, URL http://cran.r-project.org/packages=dclone.

Sólymos P (2010). *dcpar: Parallel Computing with MCMC and Data Cloning*. R package version 0.2-0, URL http://dcr.r-forge.r-project.org.

Sólymos P (2010 under review). "dclone: Data Cloning in R." Submitted to R Journal.

Spiegelhalter D, Thomas A, Best N, Lunn D (2007). *OpenBUGS User Manual, Version 3.0.2, September 2007*. URL http://mathstat.helsinki.fi/openbugs/.

Spiegelhalter DJ, Thomas A, Best NG, Lunn D (2003). "WinBUGS Version 1.4 Users Manual." MRC Biostatistics Unit, Cambridge. URL http://www.mrc-bsu.cam.ac.uk/bugs/.

Tierney L, Rossini AJ, Li N, Sevcikova H (2010). *snow: Simple Network of Workstations*. R package version 0.3-3, URL http://cran.r-project.org/packages=snow.

Affiliation:

Péter Sólymos
Alberta Biodiversity Monitoring Institute
and Boreal Avian Modelling Project
Department of Biological Sciences
CW 405, Biological Sciences Bldg
University of Alberta
Edmonton, Alberta, T6G 2E9, Canada
E-mail: solymos@ualberta.ca

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume VV, Code Snippet II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd