Technical Report

Performance Analysis for Parallel R Programs: Towards Efficient Resource Utilization

Helena Kotthaus, Ingo Korb,
Peter Marwedel

01/2015

technische universität
dortmund

SFB 876 Verfügbarkeit von
Information durch Analyse unter
Ressourcenbeschränkung

Part of the work on this technical report has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A3.

Speaker: Prof. Dr. Katharina Morik
Address: TU Dortmund University
Joseph-von-Fraunhofer-Str. 23
D-44227 Dortmund
Web: http://sfb876.tu-dortmund.de

1 Introduction

The R programming language [1] is widely used in biostatistics with high-dimensional data sets. Here, a vast amount of resources is needed. Our tool traceR [5] allows the user to profile the resource usage of an R application to locate bottlenecks and develop new optimizations [8, 7]. TraceR is a deterministic profiling tool. In contrast to existing R language profiling tools, traceR is directly integrated with the R interpreter. This enables the generation of more detailed and accurate data about memory and runtime behavior of an R application. This profiling functionality was previously limited to sequential R applications.

Parallel computing however is becoming more and more popular, since R is increasingly used to process large data sets. We therefore have improved traceR to allow for profiling parallel applications also. TraceR can be used for common cases like parallelization on multiple cores or parallelization on multiple machines. For the parallel performance analysis we added measurements like CPU utilization of parallel tasks and measurements for analyzing the memory usage of parallel programs during execution [9, 10]. With our parallel performance analysis we concentrate on applications that are embarrassingly parallel consisting of independent tasks. One example application which is embarrassingly parallel and also has a high resource utilization is the model selection. Here the goal is to find the best machine learning algorithm configuration for building a model for the given data. Therefore one has to search through a huge model space.

Since the gain from parallel execution can be negated if the memory requirements of all parallel processes exceed the capacity of the system, our profiling data can serve as a constraint to determine the degree of parallelism and also to guide distribution of parallel R applications. Our goal is to provide a resource-aware parallelization strategy. To develop such a strategy we first need to analyze the performance of parallel applications. In the following we therefore will describe different parallel example applications and show how traceR is applied to analyze parallel R applications.

2 Parallel Performance Analysis

In the following we will illustrate the gain from analyzing parallel R programs with traceR by presenting three example cases. Subsection 2.1 explains the parallel profiles produced by traceR and shows the outcome of using too many machines for parallel execution. Subsection 2.2 discusses the problem of high runtime variance of parallel processes and load balancing options. An example of inefficient resource utilization caused by using too many parallel processes is presented in Subsection 2.3.

2.1 Inefficient Resource Utilization

In this subsection we will first use traceR to analyze an R example program which runs on multiple cores. To show how traceR can support the detection of inefficient resource utilization for parallel R programs, we will extend this example program for execution on multiple machines. Figure 1 shows the parallel parts of an R program that calculates

```
...
5  library("parallel")
6
7  # resolution and iteration count constants
8  XRES      = 5000
9  YRES      = 5000
10 BLOCKSIZE = 100
11 ITERS     = 600
12 CORES     = 4
13
14 # calculate C value for a given pixel (coordinates between 0 & 1)
15 calcC = function(imag, real) {
16   return((-1 + 2 * imag) * 1i + (-2.3 + 3.0*real))
17 }
18
19 # calculate a single line of the final image
20 calcBlock = function(line) {
21     image = matrix(0+0i, ncol = XRES, nrow = BLOCKSIZE)
22     pixelC = outer(seq(line, line + BLOCKSIZE - 1) / YRES,
23                    (1:XRES) / XRES, calcC)
24     for (i in 1:ITERS) {
25         image = image * image + pixelC
26     }
27     t(image)
28 }
29 lines = do.call(c, mclapply(seq(1,YRES,BLOCKSIZE),
30                     calcBlock, mc.cores = CORES))
...
```

Figure 1: Parallel code parts of an R program for calculating a Mandelbrot fractal on 4 cores on a single machine.

a Mandelbrot fractal. The important lines are marked in green. Here the function *calcBlock* (line 20) is running in parallel on 4 cores (line 12) on a single machine. Each job calculates a block of lines of the final Mandelbrot image. For parallelization the R function *mclapply* (line 29) of the *parallel* R package [3] is used. This function sets up a pool of 4 worker processes with the fork mechanism, one per core. The jobs are prescheduled by default. Therefore *mclapply* first splits the jobs into as many groups as there are cores and sends them to the 4 worker processes, each covering more than one job.

For analyzing the resource consumption of this parallel example code our traceR tool produces a profile that visualizes the relative CPU utilization and memory consumption which is shown in Figure 2. Besides this profile traceR also generates more detailed runtime profiles and memory profiles for each parallel process (for a detailed description see [6]). This data is especially useful to support efficient distribution of parallel processes on heterogeneous machines or cores, a manual for installing and configuring traceR can be found in [5].

The x-axis of Figure 2 represents the runtime in seconds for the R program example. The runtime and the CPU utilization of the parallel processes are illustrated by a horizontal line for each process. The length of the line represents the runtime. The color can vary from blue via purple to orange and indicates the relative CPU utilization of a process, calculated over its entire runtime. The master process is shown on the top while the worker processes are shown below it, sorted by starting time.

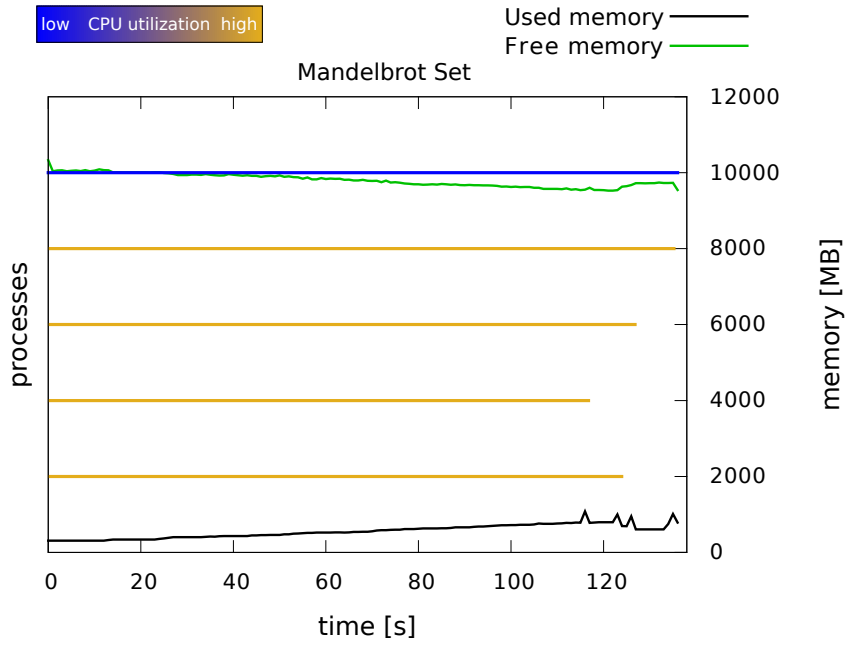

Figure 2: Relative CPU utilization and memory utilization of a parallel R program for calculating a Mandelbrot fractal on 4 cores on a single machine.

Since *mclapply* uses one worker process per core, Figure 2 shows 4 orange lines for the worker processes and one blue line for the master process. The worker processes are orange since there CPU utilization is high, this indicates that they did not have to wait for CPU resources, instead they received the maximum amount of CPU time. If their color would shift to purple or blue this would indicate that they had consumed less CPU time than their wallclock-runtime. This could be caused by other concurrent processes running on the same machine, by running more processes than CPU cores available or just by processing delays that are for example caused by I/O operations. The measure-

ments for the master process shown in the first horizontal line of Figure 2 indicate a low CPU utilization (blue) since the master only waits for the workers to finish and gathers the results at the end without running any calculations.

The y-axis on the right side of Figure 2 shows the memory utilization of the Mandelbrot program. The memory behavior is indicated by two curves. The green curve shows the amount of free main memory during program execution, reported by the Linux kernel. The black curve shows the amount of memory allocated by all parallel processes over time. The values for both curves are sampled once per second. For the Mandelbrot example this profile indicates that we have more memory resources than needed.

A common way to further reduce the effective runtime of a parallel program is to execute it on multiple machines. TraceR is also able to produce profiles for parallelization on multiple machines and the R language supports this way of parallelization via several R packages. We will therefore modify our R program example by using additional mechanisms of the parallel R package. This modification is shown in Figure 3. Again the important parts of the code are marked in green.

```
...
11 HOSTS = c(rep("host1", 4), rep("host2", 4))
12
13 # calculate C value for a given pixel (coordinates between 0 and 1)
14 calcC = function(imag, real) {
15   return((-1 + 2 * imag) * 1i + (-2.3 + 3.0*real))
16 }
17
18 # calculate a block of the final image
19 calcBlock = function(line) {
20     image = matrix(0+0i, ncol = XRES, nrow = BLOCKSIZE)
21     pixelC = outer(seq(line, line + BLOCKSIZE - 1) / YRES,
22                    (1:XRES) / XRES, calcC)
23     for (i in 1:ITERS) {
24         image = image * image + pixelC
25     }
26     t(image)
27 }
28
29 clust <- makePSOCKcluster(HOSTS)
30 clusterExport(clust, c("XRES", "YRES", "BLOCKSIZE", "ITERS", "calcC"))
31 lines = do.call(c, parLapply(clust, seq(1,YRES,BLOCKSIZE), calcBlock))
32 stopCluster(clust)
...
```

Figure 3: Parallel code parts of an R program for calculating a Mandelbrot fractal using 8 cores on 2 machines.

To run this example program the machines and cores that should be used for execution first have to be specified. This is done by defining a list of machines (line 11). Here the number of cores was increased by a factor of two. We are using 4 cores on each machine (*host1* and *host2*). The *makePSOCKcluster* function (line 29) and the *parLapply* function (line 31) are launching further copies of R interpreters by using the *Rscript* front end, which is an alternative front end for using R in scripts. This mechanism creates a cluster of workers based on sockets. Here again the jobs are prescheduled and therefore evenly split into groups and send to the worker processes. If the calculation of a program

was distributed over multiple hosts, traceR generates one sub-profile for each machine. In this case the master process is only shown in the sub-profile for the machine it was running on, here on *host2*.

Compared to the profile in Figure 2 where the program was executed only on 4 cores this profile shows the parallel execution on 8 cores on 2 machines. But by doubling the number of cores only minor runtime savings were realized. This indicates an inefficient resource utilization. One reason for this is the overhead of using the *Rscript* to launch futher copies of the master process. Another reason is indicated by the CPU utilization of the worker processes on *host2* where the master process is executed. In the previous profile where the program was executed on a single machine all worker processes had a high CPU utilization and also a variance in runtimes. Now the single runtimes of the worker processes are very similar to each other and their CPU utilization is lower (purple color). The reason for this is that the worker processes are not able to stop execution when their computation is finished, they have to wait for the master process and the master first waits for all computations to be finished and then shuts down the worker processes.

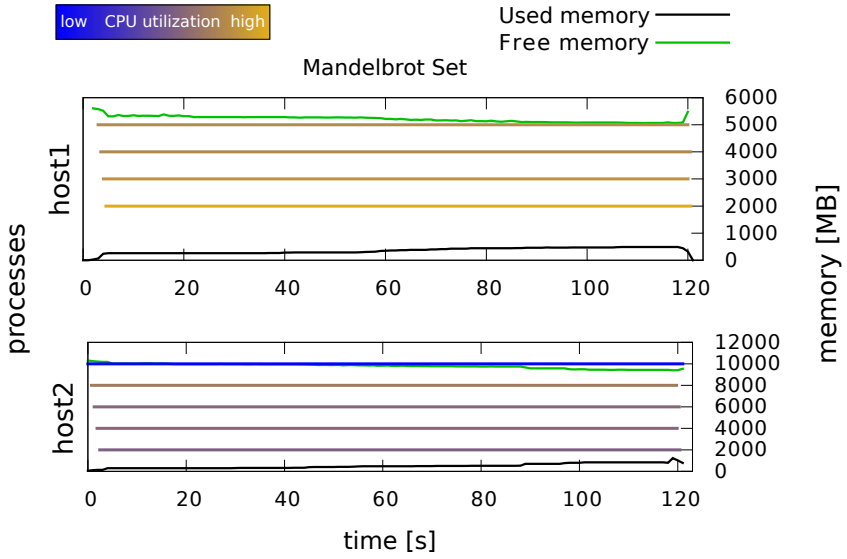

Figure 4: Relative CPU utilization and memory utilization of an R program for calculating a Mandelbrot fractal using 8 cores on 2 machines.

In summary the parallel profile in Figure 4 shows that using two machines for the Mandelbrot example produces overhead and thus does not lead to an efficient runtime reduction. For a runtime reduction the execution on a single machine with more cores would be more efficient and would avoid inefficient resource utilization. In the next subsection we will introduce a real world code example running on a single machine, were the runtime variance between jobs is higher compared to this Mandelbrot program example.

2.2 Runtime Variance and Load Balancing

If parallel executed jobs have a high variance in runtime, load balancing is recommended to reduce inefficient resource utilization. In this subsection we want to discuss the problem of high runtime variance of parallel processes by analyzing a real world code example with traceR. Figure 5 therefore shows the parallel parts of an R program for evaluating different parameter configurations of a SVM classification task based on the *mlr* library [2]. Again the important parts of the program are marked in green. The performance of a SVM classification highly depends on its parameter configurations, therefore it is important to tune those parameters by evaluating the performance of different configurations.

```
...
2  data = libsvm.read(data_file)
3  task = makeClassifTask(data = data, target = "Y")
4
5  # evaluate different parameter configurations for SVM classification
6  library(parallelMap)
7  parallelStartMulticore(cpu_cores = 4, mc.preschedule = TRUE)
8
9  lrn = makeLearner("classif.LiblineaRMultiClass")
10 par.set = makeParamSet(
11   makeNumericParam("cost", lower = -15, upper = 15, trafo = function(x) 2^x),
12   makeNumericParam("epsilon", lower = -15, upper = 15, trafo = function(x) 2^x))
13 rsi = makeResampleDesc(method = "Holdout", stratify = TRUE)
14 tune.ctrl = makeTuneControlGrid(resolution = 10)
15
16 t.res = tuneParams(learner = lrn, task = task, resampling = rsi,
17           measures = list(mmce, timetrain, timepredict, timeboth),
18           par.set = par.set, control = tune.ctrl)
19 op.df = as.data.frame(t.res$opt.path)
20
21 parallelStop()
...
```

Figure 5: Parallel code parts of an R program for evaluating different parameter configurations of a SVM classification using 4 cores on a single machines.

For parallelization of the SVM program in Figure 5 the *parallelStartMulticore* function (line 7) from the *parallelMap* R library [4] is used. This library utilizes the mechanisms of the parallel package. Here the master process is copied via fork and the parameter *mc.preschedule* is passed for splitting the jobs into groups and sending them to the worker processes. Here again 4 cores on a single machine (line 7) are used. Each job evaluates the performance of the SVM with another parameter configuration, this is triggered by the function *tuneParams* (line 16). Figure 6 shows the corresponding traceR profile.

As can be seen from the black curve that indicates the memory allocation, here much more memory is needed for computation compared to our previous Mandelbrot program. Here all worker processes have a high CPU utilization (orange color). But the completion times of the worker processes have a high variance (length of the horizontal lines). This indicates, that runtime could be saved and thus resources by distributing the jobs more evenly on the worker processes. The parallel package supports such mechanisms through its *load balancing* option. This option is recommended when the jobs of a parallel program have widely different computation times or the machines are heterogeneous.

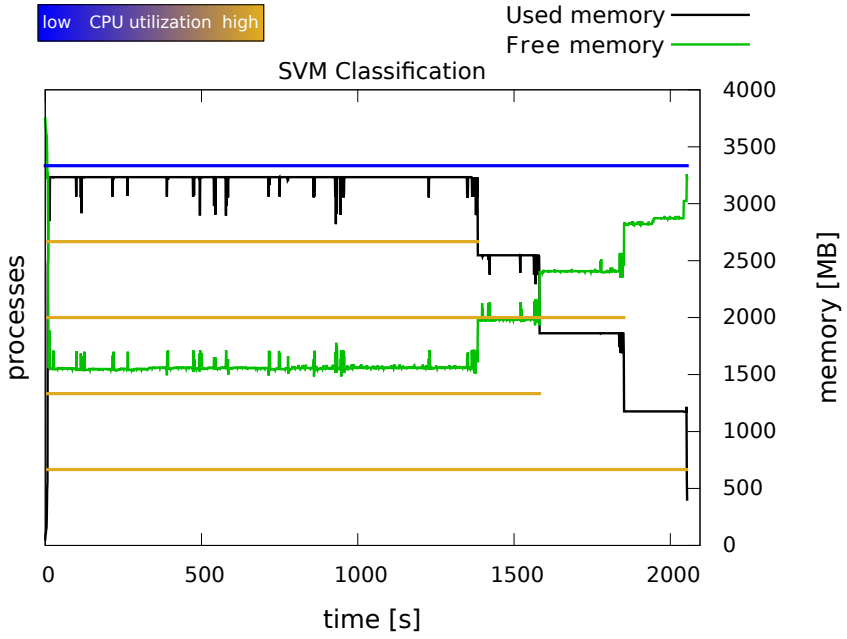

Figure 6: Relative CPU utilization and memory utilization of an R program evaluating different parameter configurations of a SVM classification using 4 cores on a single machines.

Figure 7 shows the modification marked in green (line 7) needed to enable the load balancing option of the parallel package. Therefore the *mc.preschedule* parameter is passed with the *FALSE* argument. Figure 8 presents the corresponding profile for CPU utilization and memory utilization of the SVM program that utilizes the load balancing mechanism.

```
...
2  data = libsvm.read(data_file)
3  task = makeClassifTask(data = data, target = "Y")
4
5  # evaluate different parameter configurations for SVM classification
6  library(parallelMap)
7  parallelStartMulticore(cpu_cores = 4, mc.preschedule = FALSE)
8
9  lrn = makeLearner("classif.LiblineaRMultiClass")
10 par.set = makeParamSet(
11   makeNumericParam("cost", lower = -15, upper = 15, trafo = function(x) 2^x),
12   makeNumericParam("epsilon", lower = -15, upper = 15, trafo = function(x) 2^x))
13 rsi = makeResampleDesc(method = "Holdout", stratify = TRUE)
14 tune.ctrl = makeTuneControlGrid(resolution = 10)
15
16 t.res = tuneParams(learner = lrn, task = task, resampling = rsi,
17           measures = list(mmce, timetrain, timepredict, timeboth),
18           par.set = par.set, control = tune.ctrl)
19 op.df = as.data.frame(t.res$opt.path)
20
21 parallelStop()
...
```

Figure 7: Parallel code parts of an R program for evaluating different parameter configurations of a SVM classification using load balancing on 4 cores on a single machine.

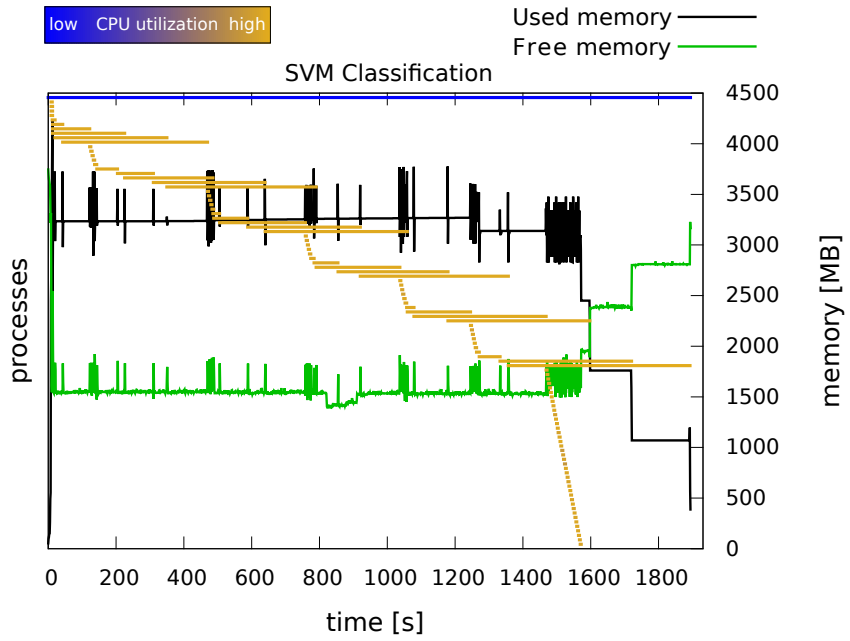

Figure 8: Relative CPU utilization and memory utilization of an R program evaluating different parameter configurations of a SVM classification using load balancing on 4 cores on a single machine.

To visualize the load balancing option the traceR profile shows one horizontal line for the CPU utilization of each job. All jobs are still processed on 4 cores. Load balancing dynamically allocates jobs to the worker processes, it sends jobs one at a time to a core (or machine). Compared to the profile from Figure 6 where load balancing was turned off here we could save runtime and thus resources. However there is still a high variance of computation times that leads to an inefficient resource utilization. This indicates that the load balancing mechanism of the parallel package is not sufficient for the SVM program where most of the jobs have a high variance in computation times. This high variance however is a common case for parameter tuning of machine learning algorithms. Therefore our future work will concentrate on the development of an optimized scheduling strategy for such use cases.

Inefficient resource utilization can not only be caused by insufficient load balancing. In the next subsection we will discuss an example case where inefficient resource utilization is caused by creating to many parallel processes.

2.3 Number of Parallel Processes

The gain from parallel execution can be negated if the memory requirements of all processes exceed the capacity of the RAM. By triggering too many parallel processes the OS starts to swap out memory which leads to inefficient resource utilization. An example profile illustrating this case is shown in Figure 9. Here the same SVM program code from Figure 7 was running, but this time on a system with lower memory resources (2GB of RAM).

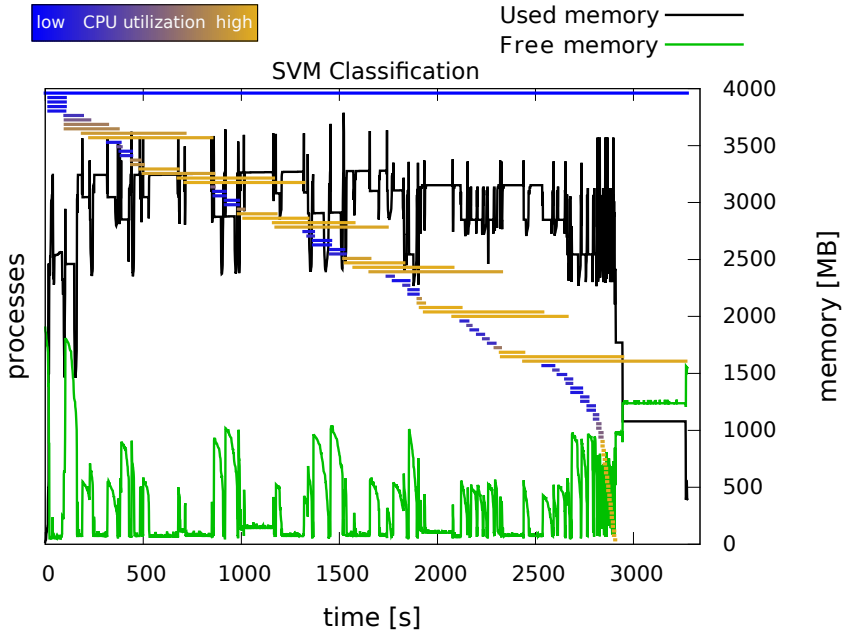

Figure 9: Relative CPU utilization and memory utilization of an R program evaluating different parameter configurations of a SVM classification using load balancing on 4 cores on a single machine with 2 GB of RAM.

Compared to Figure 8 where all processes had a high CPU utilization this profile includes more blue lines indicating a low CPU utilization. This low CPU utilization is caused by processes that are waiting for processing their data because it was swapped out. This is also represented by the green curve that indicates dthe free memory which is most of the time close to zero. If only 2 instead of 4 cores would have been used the CPU utilization would be high for all jobs since using less worker processes also reduces the data that needs to be copied. The parallel processes are all independent. They are not aware of each others memory usage, thus one job cannot trigger garbage collection in another and free memory for it.

This example shows that the development of an optimized scheduling strategy for parallel R applications also has to consider the memory requirements of applications. Here TraceR supports the development by generation those parallel profiles.

3 Conclusion

We presented the gain from analyzing parallel R applications with traceR by showing three example cases. We presented the outcome of using too many machines for parallel execution, we discussed the problem of high runtime variance of parallel processes including load balancing and we presented how inefficient resource utilization can be caused by using too many parallel processes.

Our parallel performance analysis with TraceR can steer the development of parallel R programs aimed at overcoming inefficient resource utilizations we are currently facing. Our long term goal is to realize an optimized scheduling strategy for parallel R programs Therefore the information gathered by traceR could be used to guide the scheduling decisions to allow for efficient resource utilization. Such decisions are especially important if the hardware system is heterogeneous or if the jobs have varying resource requirements depending on the input data.

References

[1] R Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria, 2015. URL `http://www.R-project.org`

[2] Bischl B., Lang M., Richter J., Bossek J., Judt L., Kuehn T., Studerus E., Kotthoff L., Jones Z. mlr: Machine Learning in R. Version 2.4. 2015. URL `http://CRAN.R-project.org/package=mlr`

[3] Ripley B, Tierney L., Urbanek S., parallel: Support for Parallel Computation. R package included in the R-core 2.14.0. 2015. URL `http://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf`

[4] Bischl B., Lang M., parallelMap: Unified Interface to Parallelization Back-Ends. R package version 1.3. 2015. URL `http://CRAN.R-project.org/package=parallelMap`

[5] TraceR: A Profiling Tool for R. TU Dortmund University. 2015. URL `https://github.com/allr/traceR-installer`

[6] R-instrumented: Part of the TraceR Profiling Tool for R. TU Dortmund University. 2015. URL `https://github.com/allr/r-instrumented`

[7] Kotthaus H., Korb I., Lang M., Bischl B., Rahnenführer J., Marwedel P., Runtime and Memory Consumption Analyses for Machine Learning R Programs. Journal of Statistical Computation and Simulation. Volume 85, Issue 1, pp.14-29. 2014.

[8] Kotthaus, H., Korb, I., Engel, M., Marwedel, P., Dynamic Page Sharing Optimization for the R Language. Proceedings of the 10th Symposium on Dynamic Languages (DLS'14). Portland, USA. pp. 79-90. 2014.

[9] Kotthaus, H., Korb, I., Marwedel, P., Performance Analysis for Parallel R Programs: Towards Efficient Resource Utilization. Abstract Booklet of the International R User Conference (UseR!). Aalborg, Denmark, pp. 66. 2015.

[10] Kotthaus, H., Korb, I., Marwedel, P., Distributed Performance Analysis for R. R Implementation, Optimization and Tooling Workshop (RIOT). Prag, Czech, 2015.